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 Standard EA 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
<table>
<thead>
<tr>
<th>Section</th>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.4</td>
<td>Strings...</td>
<td>24</td>
</tr>
<tr>
<td>7.4.1</td>
<td>C++ syntax</td>
<td>24</td>
</tr>
<tr>
<td>7.4.2</td>
<td>Examples</td>
<td>25</td>
</tr>
<tr>
<td>7.5</td>
<td>chandles...</td>
<td>26</td>
</tr>
<tr>
<td>7.6</td>
<td>Struts...</td>
<td>27</td>
</tr>
<tr>
<td>7.6.1</td>
<td>DSL syntax</td>
<td>27</td>
</tr>
<tr>
<td>7.6.2</td>
<td>C++ syntax</td>
<td>27</td>
</tr>
<tr>
<td>7.6.3</td>
<td>Examples</td>
<td>28</td>
</tr>
<tr>
<td>7.7</td>
<td>User-defined data types</td>
<td>28</td>
</tr>
<tr>
<td>7.7.1</td>
<td>DSL syntax</td>
<td>29</td>
</tr>
<tr>
<td>7.7.2</td>
<td>C++ syntax</td>
<td>29</td>
</tr>
<tr>
<td>7.7.3</td>
<td>Examples</td>
<td>29</td>
</tr>
<tr>
<td>7.8</td>
<td>Arrays...</td>
<td>29</td>
</tr>
<tr>
<td>7.8.1</td>
<td>C++ syntax</td>
<td>29</td>
</tr>
<tr>
<td>7.8.2</td>
<td>Examples</td>
<td>34</td>
</tr>
<tr>
<td>7.8.3</td>
<td>Properties</td>
<td>34</td>
</tr>
<tr>
<td>8.1</td>
<td>DSL syntax</td>
<td>36</td>
</tr>
<tr>
<td>8.2</td>
<td>C++ syntax</td>
<td>37</td>
</tr>
<tr>
<td>8.3</td>
<td>Examples</td>
<td>37</td>
</tr>
<tr>
<td>9.1</td>
<td>Buffer objects</td>
<td>38</td>
</tr>
<tr>
<td>9.1.1</td>
<td>DSL syntax</td>
<td>38</td>
</tr>
<tr>
<td>9.1.2</td>
<td>C++ syntax</td>
<td>38</td>
</tr>
<tr>
<td>9.1.3</td>
<td>Examples</td>
<td>39</td>
</tr>
<tr>
<td>9.2</td>
<td>Stream objects</td>
<td>40</td>
</tr>
<tr>
<td>9.2.1</td>
<td>DSL syntax</td>
<td>41</td>
</tr>
<tr>
<td>9.2.2</td>
<td>C++ syntax</td>
<td>41</td>
</tr>
<tr>
<td>9.2.3</td>
<td>Examples</td>
<td>41</td>
</tr>
<tr>
<td>9.3</td>
<td>State objects</td>
<td>43</td>
</tr>
<tr>
<td>9.3.1</td>
<td>DSL syntax</td>
<td>43</td>
</tr>
<tr>
<td>9.3.2</td>
<td>C++ syntax</td>
<td>44</td>
</tr>
<tr>
<td>9.3.3</td>
<td>Examples</td>
<td>44</td>
</tr>
<tr>
<td>9.4</td>
<td>Using flow objects</td>
<td>45</td>
</tr>
<tr>
<td>9.4.1</td>
<td>DSL syntax</td>
<td>45</td>
</tr>
<tr>
<td>9.4.2</td>
<td>C++ syntax</td>
<td>45</td>
</tr>
<tr>
<td>9.4.3</td>
<td>Examples</td>
<td>46</td>
</tr>
<tr>
<td>9.5</td>
<td>Implicitly binding flow objects</td>
<td>46</td>
</tr>
<tr>
<td>10.1</td>
<td>Declaring resource objects</td>
<td>47</td>
</tr>
<tr>
<td>10.1.1</td>
<td>DSL syntax</td>
<td>47</td>
</tr>
<tr>
<td>10.1.2</td>
<td>C++ syntax</td>
<td>47</td>
</tr>
<tr>
<td>10.1.3</td>
<td>Examples</td>
<td>48</td>
</tr>
<tr>
<td>10.2</td>
<td>Claiming resource objects</td>
<td>48</td>
</tr>
<tr>
<td>10.2.1</td>
<td>DSL syntax</td>
<td>48</td>
</tr>
<tr>
<td>10.2.2</td>
<td>C++ syntax</td>
<td>48</td>
</tr>
<tr>
<td>10.2.3</td>
<td>Examples</td>
<td>49</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>13.1.4</td>
<td>Set membership expression</td>
<td>108</td>
</tr>
<tr>
<td>13.1.5</td>
<td>Implication constraint</td>
<td>110</td>
</tr>
<tr>
<td>13.1.6</td>
<td>if-else constraint</td>
<td>111</td>
</tr>
<tr>
<td>13.1.7</td>
<td>foreach constraint</td>
<td>113</td>
</tr>
<tr>
<td>13.1.8</td>
<td>Unique constraint</td>
<td>115</td>
</tr>
<tr>
<td>13.2</td>
<td>Scheduling constraints</td>
<td>116</td>
</tr>
<tr>
<td>13.2.1</td>
<td>DSL syntax</td>
<td>116</td>
</tr>
<tr>
<td>13.2.2</td>
<td>Example</td>
<td>117</td>
</tr>
<tr>
<td>13.3</td>
<td>Randomization process</td>
<td>117</td>
</tr>
<tr>
<td>13.3.1</td>
<td>Random attribute fields</td>
<td>118</td>
</tr>
<tr>
<td>13.3.2</td>
<td>Randomization of flow objects</td>
<td>120</td>
</tr>
<tr>
<td>13.3.3</td>
<td>Randomization of resource objects</td>
<td>122</td>
</tr>
<tr>
<td>13.3.4</td>
<td>Randomization of component assignment</td>
<td>124</td>
</tr>
<tr>
<td>13.3.5</td>
<td>Random value selection order</td>
<td>125</td>
</tr>
<tr>
<td>13.3.6</td>
<td>Loops and random value selection</td>
<td>125</td>
</tr>
<tr>
<td>13.3.7</td>
<td>Relationship lookahead</td>
<td>127</td>
</tr>
<tr>
<td>13.3.8</td>
<td>Lookahead and sub-actions</td>
<td>129</td>
</tr>
<tr>
<td>13.3.9</td>
<td>Lookahead and dynamic constraints</td>
<td>131</td>
</tr>
<tr>
<td>13.3.10</td>
<td>pre_solve and post_solve exec blocks</td>
<td>133</td>
</tr>
<tr>
<td>13.3.11</td>
<td>Body blocks and sampling external data</td>
<td>138</td>
</tr>
<tr>
<td>14.1</td>
<td>coverspec declaration</td>
<td>141</td>
</tr>
<tr>
<td>14.1.1</td>
<td>DSL syntax</td>
<td>142</td>
</tr>
<tr>
<td>14.1.2</td>
<td>Examples</td>
<td>142</td>
</tr>
<tr>
<td>14.2</td>
<td>coverspec instantiation</td>
<td>143</td>
</tr>
<tr>
<td>14.2.1</td>
<td>DSL syntax</td>
<td>143</td>
</tr>
<tr>
<td>14.2.2</td>
<td>Examples</td>
<td>143</td>
</tr>
<tr>
<td>14.3</td>
<td>coverpoint goal</td>
<td>143</td>
</tr>
<tr>
<td>14.4</td>
<td>Referencing existing bin schemes</td>
<td>144</td>
</tr>
<tr>
<td>14.5</td>
<td>cross goal</td>
<td>144</td>
</tr>
<tr>
<td>14.6</td>
<td>coverspec constraints</td>
<td>145</td>
</tr>
<tr>
<td>14.6.1</td>
<td>Ignore constraint</td>
<td>145</td>
</tr>
<tr>
<td>14.6.2</td>
<td>Illegal constraint</td>
<td>146</td>
</tr>
<tr>
<td>14.7</td>
<td>coverspec bins</td>
<td>147</td>
</tr>
<tr>
<td>14.7.1</td>
<td>DSL syntax</td>
<td>148</td>
</tr>
<tr>
<td>14.7.2</td>
<td>Examples</td>
<td>148</td>
</tr>
<tr>
<td>14.7.3</td>
<td>Explicit value and range grouping</td>
<td>148</td>
</tr>
<tr>
<td>14.7.4</td>
<td>Value range divide operator (/)</td>
<td>148</td>
</tr>
<tr>
<td>14.7.5</td>
<td>Value range size operator (;)</td>
<td>149</td>
</tr>
<tr>
<td>14.7.6</td>
<td>Wildcard bin (*)</td>
<td>149</td>
</tr>
<tr>
<td>15.1</td>
<td>Specifying type extensions</td>
<td>150</td>
</tr>
<tr>
<td>15.1.1</td>
<td>DSL syntax</td>
<td>150</td>
</tr>
<tr>
<td>15.1.2</td>
<td>C++ syntax</td>
<td>150</td>
</tr>
<tr>
<td>15.1.3</td>
<td>Examples</td>
<td>151</td>
</tr>
<tr>
<td>15.1.4</td>
<td>Compound type extensions</td>
<td>153</td>
</tr>
<tr>
<td>15.1.5</td>
<td>Enum type extensions</td>
<td>156</td>
</tr>
<tr>
<td>15.1.6</td>
<td>Ordering of type extensions</td>
<td>158</td>
</tr>
</tbody>
</table>
### 15.2 Overriding types

<table>
<thead>
<tr>
<th>Subsections</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.2.1 DSL syntax</td>
<td>158</td>
</tr>
<tr>
<td>15.2.2 C++ syntax</td>
<td>158</td>
</tr>
<tr>
<td>15.2.3 Examples</td>
<td>159</td>
</tr>
</tbody>
</table>

### 16. Packages

<table>
<thead>
<tr>
<th>Subsections</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.1 Package declaration</td>
<td>162</td>
</tr>
<tr>
<td>16.1.1 DSL syntax</td>
<td>163</td>
</tr>
<tr>
<td>16.1.2 C++ syntax</td>
<td>163</td>
</tr>
<tr>
<td>16.1.3 Examples</td>
<td>163</td>
</tr>
<tr>
<td>16.2 Namespaces and name resolution</td>
<td>164</td>
</tr>
<tr>
<td>16.3 Import statement</td>
<td>164</td>
</tr>
<tr>
<td>16.4 Naming rules for members across extensions</td>
<td>164</td>
</tr>
</tbody>
</table>

### 17. Test realization

<table>
<thead>
<tr>
<th>Subsections</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.1 exec blocks</td>
<td>165</td>
</tr>
<tr>
<td>17.1.1 DSL syntax</td>
<td>165</td>
</tr>
<tr>
<td>17.1.2 C++ syntax</td>
<td>166</td>
</tr>
<tr>
<td>17.1.3 Examples</td>
<td>168</td>
</tr>
<tr>
<td>17.2 Implementation using a procedural interface (PI)</td>
<td>168</td>
</tr>
<tr>
<td>17.2.1 Import function declaration</td>
<td>168</td>
</tr>
<tr>
<td>17.2.2 DSL syntax</td>
<td>168</td>
</tr>
<tr>
<td>17.2.3 C++ syntax</td>
<td>168</td>
</tr>
<tr>
<td>17.2.4 Examples</td>
<td>169</td>
</tr>
<tr>
<td>17.2.5 Method result</td>
<td>170</td>
</tr>
<tr>
<td>17.2.6 Method parameters</td>
<td>170</td>
</tr>
<tr>
<td>17.2.7 Parameter direction</td>
<td>170</td>
</tr>
<tr>
<td>17.3 PI PSS layer</td>
<td>171</td>
</tr>
<tr>
<td>17.4 PI function qualifiers</td>
<td>171</td>
</tr>
<tr>
<td>17.4.1 DSL syntax</td>
<td>171</td>
</tr>
<tr>
<td>17.4.2 C++ syntax</td>
<td>171</td>
</tr>
<tr>
<td>17.4.3 Specifying function availability</td>
<td>172</td>
</tr>
<tr>
<td>17.4.4 Specifying an implementation language</td>
<td>173</td>
</tr>
<tr>
<td>17.5 Calling PI methods</td>
<td>174</td>
</tr>
<tr>
<td>17.6 Target-template implementation for import functions</td>
<td>176</td>
</tr>
<tr>
<td>17.6.1 DSL syntax</td>
<td>177</td>
</tr>
<tr>
<td>17.6.2 C++ syntax</td>
<td>177</td>
</tr>
<tr>
<td>17.6.3 Examples</td>
<td>177</td>
</tr>
<tr>
<td>17.7 Import classes</td>
<td>178</td>
</tr>
<tr>
<td>17.7.1 DSL syntax</td>
<td>178</td>
</tr>
<tr>
<td>17.7.2 C++ syntax</td>
<td>179</td>
</tr>
<tr>
<td>17.7.3 Examples</td>
<td>179</td>
</tr>
<tr>
<td>17.8 Implementation using target-template code blocks</td>
<td>180</td>
</tr>
<tr>
<td>17.8.1 Target-template code exec block kinds</td>
<td>180</td>
</tr>
<tr>
<td>17.8.2 Target language</td>
<td>181</td>
</tr>
<tr>
<td>17.8.3 exec file</td>
<td>181</td>
</tr>
<tr>
<td>17.9 C++ in-line solve exec implementation</td>
<td>181</td>
</tr>
<tr>
<td>17.9.1 C++ syntax</td>
<td>181</td>
</tr>
<tr>
<td>17.9.2 Examples</td>
<td>184</td>
</tr>
<tr>
<td>17.10 C++ generative target exec implementation</td>
<td>184</td>
</tr>
<tr>
<td>17.10.1 Generative PI execs</td>
<td>185</td>
</tr>
<tr>
<td>17.10.2 Generative target-template execs</td>
<td>186</td>
</tr>
</tbody>
</table>
# Contents

1. 17.11 Comparison between mapping mechanisms .................................................. 188
2. 17.12 Exported actions ..................................................................................................................... 190
3. 17.12.1 DSL syntax .................................................................................................................. 190
4. 17.12.2 C++ syntax .................................................................................................................. 190
5. 17.12.3 Examples ...................................................................................................................... 191
6. 17.12.4 Export action foreign language binding ................................................................. 192

8. 18. Hardware/Software Interface (HSI) ................................................................................................. 193

Annex A (informative) Bibliography .................................................................... 194

Annex B (normative) Formal syntax ........................................................................ 195

B.1 Package declarations .......................................................................................... 195
B.2 Action declarations ............................................................................................ 195
B.3 Struct declarations ............................................................................................... 197
B.4 Procedural interface (PI) ...................................................................................... 197
B.5 Component declarations ...................................................................................... 198
B.6 Activity statements ............................................................................................... 199
B.7 Overrides ............................................................................................................... 200
B.8 Data declarations .................................................................................................... 200
B.9 Data types .............................................................................................................. 201
B.10 Constraint ............................................................................................................. 202
B.11 Coverspec ............................................................................................................... 202
B.12 Expression ............................................................................................................ 203
B.13 Identifiers and literals .......................................................................................... 205
B.14 Numbers ............................................................................................................... 206
B.15 Comments ............................................................................................................. 206

Annex C (normative) C++ header files .................................................................... 208

C.1 File pss.h ................................................................................................................. 208
C.2 File pss/action_attr.h ............................................................................................... 208
C.3 File pss/action.h ....................................................................................................... 209
C.4 File pss/action_handle.h .......................................................................................... 211
C.5 File pss/attr.h .......................................................................................................... 211
C.6 File pss/bind.h .......................................................................................................... 215
C.7 File pss/bit.h .............................................................................................................. 215
C.8 File pss/buffer.h ....................................................................................................... 215
C.9 File pss/chandle.h ..................................................................................................... 216
C.10 File pss/comp_inst.h ............................................................................................... 216
C.11 File pss/component.h ............................................................................................. 216
<table>
<thead>
<tr>
<th>Section</th>
<th>File</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.12</td>
<td>File pss/constraint.h</td>
<td>217</td>
</tr>
<tr>
<td>C.13</td>
<td>File pss/enumeration.h</td>
<td>218</td>
</tr>
<tr>
<td>C.14</td>
<td>File pss/exec.h</td>
<td>218</td>
</tr>
<tr>
<td>C.15</td>
<td>File pss/export_action.h</td>
<td>219</td>
</tr>
<tr>
<td>C.16</td>
<td>File pss/extend.h</td>
<td>220</td>
</tr>
<tr>
<td>C.17</td>
<td>File pss/import_class.h</td>
<td>221</td>
</tr>
<tr>
<td>C.18</td>
<td>File pss/import_func.h</td>
<td>221</td>
</tr>
<tr>
<td>C.19</td>
<td>File pss/input.h</td>
<td>223</td>
</tr>
<tr>
<td>C.20</td>
<td>File pss/inside.h</td>
<td>224</td>
</tr>
<tr>
<td>C.21</td>
<td>File pss/lock.h</td>
<td>224</td>
</tr>
<tr>
<td>C.22</td>
<td>File pss/output.h</td>
<td>225</td>
</tr>
<tr>
<td>C.23</td>
<td>File pss/override.h</td>
<td>225</td>
</tr>
<tr>
<td>C.24</td>
<td>File pss/package.h</td>
<td>226</td>
</tr>
<tr>
<td>C.25</td>
<td>File pss/pool.h</td>
<td>226</td>
</tr>
<tr>
<td>C.26</td>
<td>File pss/rand_attr.h</td>
<td>226</td>
</tr>
<tr>
<td>C.27</td>
<td>File pss/range.h</td>
<td>230</td>
</tr>
<tr>
<td>C.28</td>
<td>File pss/resource.h</td>
<td>230</td>
</tr>
<tr>
<td>C.29</td>
<td>File pss/scope.h</td>
<td>231</td>
</tr>
<tr>
<td>C.30</td>
<td>File pss/share.h</td>
<td>231</td>
</tr>
<tr>
<td>C.31</td>
<td>File pss/state.h</td>
<td>232</td>
</tr>
<tr>
<td>C.32</td>
<td>File pss/stream.h</td>
<td>232</td>
</tr>
<tr>
<td>C.33</td>
<td>File pss/structure.h</td>
<td>232</td>
</tr>
<tr>
<td>C.34</td>
<td>File pss/symbol.h</td>
<td>233</td>
</tr>
<tr>
<td>C.35</td>
<td>File pss/type_decl.h</td>
<td>233</td>
</tr>
<tr>
<td>C.36</td>
<td>File pss/unique.h</td>
<td>233</td>
</tr>
<tr>
<td>C.37</td>
<td>File pss/vec.h</td>
<td>234</td>
</tr>
<tr>
<td>C.38</td>
<td>File pss/width.h</td>
<td>234</td>
</tr>
<tr>
<td>C.39</td>
<td>File pss/detail/algebExpr.h</td>
<td>234</td>
</tr>
<tr>
<td>C.40</td>
<td>File pss/detail/activityStmt.h</td>
<td>236</td>
</tr>
<tr>
<td>D.1</td>
<td>C primitive types</td>
<td>237</td>
</tr>
<tr>
<td>D.2</td>
<td>C++ composite and user-defined types</td>
<td>237</td>
</tr>
<tr>
<td>D.3</td>
<td>SystemVerilog</td>
<td>240</td>
</tr>
<tr>
<td>E</td>
<td>(informative) Solution space</td>
<td>241</td>
</tr>
<tr>
<td>F</td>
<td>(informative) HSI UART example</td>
<td>244</td>
</tr>
</tbody>
</table>
PSS Early Adopter (EA): A Portable Stimulus and Test Standard

NOTE—Some of the material in this EA version remains under active discussion by the PSS working group; consequently, there may be substantive changes before the PSS 1.0 version is released.

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.

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 Specification 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) as a referent.

1.3 Modeling concepts

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 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

---

1Information on references can be found in Clause 2.
implementation for a target platform is captured in one of three ways: as a sequence of calls to external 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>
<thead>
<tr>
<th>Visual cue</th>
<th>Represents</th>
</tr>
</thead>
</table>
| **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 *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 } } [ ; ] |
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 execution semantic concepts.
- Clause 6 details some specific C++ considerations in using PSS.
- Clause 7 highlights the PSS data types.
- Clause 8 - Clause 17 describe the PSS modeling constructs.
- Clause 18 highlights the Hardware/Software Interface (HSI).
- Annexes. Following Clause 18 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.


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

---

2The IEEE standards or products referred to in this clause are trademarks of the Institute of Electrical and Electronics Engineers, Inc.
3IEEE publications are available from the Institute of Electrical and Electronics Engineers, Inc., 445 Hoes Lane, Piscataway, NJ 08854, USA (http://standards.ieee.org/).
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 **scenarios** 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 Stimulus language Specification

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_identifier, e.g., action_identifier. See also B.13.

4.3 Keywords

PSS reserves the keywords listed in Table 2.

Table 2—PSS keywords

<table>
<thead>
<tr>
<th>abstract</th>
<th>action</th>
<th>activity</th>
<th>bind</th>
<th>bins</th>
<th>bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>buffer</td>
<td>chandle</td>
<td>class</td>
<td>component</td>
<td>constraint</td>
</tr>
<tr>
<td>coverpoint</td>
<td>coverspec</td>
<td>cross</td>
<td>dynamic</td>
<td>else</td>
<td>enum</td>
</tr>
<tr>
<td>exec</td>
<td>export</td>
<td>extend</td>
<td>false</td>
<td>file</td>
<td>foreach</td>
</tr>
<tr>
<td>if</td>
<td>import</td>
<td>inout</td>
<td>input</td>
<td>inside</td>
<td>instance</td>
</tr>
<tr>
<td>int</td>
<td>lock</td>
<td>option</td>
<td>output</td>
<td>override</td>
<td>package</td>
</tr>
<tr>
<td>parallel</td>
<td>pool</td>
<td>rand</td>
<td>repeat</td>
<td>resource</td>
<td>schedule</td>
</tr>
<tr>
<td>select</td>
<td>sequence</td>
<td>share</td>
<td>solve</td>
<td>state</td>
<td>stream</td>
</tr>
<tr>
<td>string</td>
<td>struct</td>
<td>symbol</td>
<td>target</td>
<td>true</td>
<td>type</td>
</tr>
<tr>
<td>typedef</td>
<td>unique</td>
<td>void</td>
<td>while</td>
<td>with</td>
<td></td>
</tr>
</tbody>
</table>
5. Execution semantic concepts

5.1 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 17.8.1) 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.

5.2 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.

5.2.1 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.

5.2.2 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.

5.2.3 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.

5.3 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.

5.3.1 Preliminary definitions

a) An action-execution of an atomic action type is the execution of its exec-body block, with values assigned to all of its parameters (reachable attributes). The execution of a compound action consists in executing the set of atomic actions it contains, directly or indirectly. For more on execution semantics of compound actions and activities, see Clause 12.

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 5.2.1).

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 action-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 5.2.3).

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.

5.3.2 Sequential scheduling

Action-executions a and b are scheduled in sequence if b has a scheduling dependency on a. Two sets of action-executions, S₁ and S₂, are scheduled in sequence if every initial action-execution in S₂ has scheduling

---

6Throughout 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 $S_1 .. S_n$ is the scheduling dependency of every initial action-execution in $S_i$ on every final action-execution in $S_{i-1}$ for every $i <= N$.

For examples of sequential scheduling, see 12.3.2.3.

### 5.3.3 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 12.3.3.3.
6. C++ specifics

All PSS/C++ types are defined in the `pss` namespace and are the only types defined by this specification.

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.

```cpp
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();
};
```

**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.

```cpp
template<class T>
    class type_decl : public detail::TypeDeclBase {
            public:
                type_decl();
                T* operator-> ();
                T& operator* ();
        };
```

**Syntax 2—C++: type declaration**

Example 1 shows an example of this usage.
Example 1—C++: type declaration

class C1 : public component {
    public:
        C1 ( const scope& s ) : component (this) {}
    }
    type_decl<C1> C1_decl;

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.

Example 2—C++: Simplifying class declarations

class C2 : public component {
    PSS_CTOR(C2,component);
    }
    type_decl<C2> C2_decl;
7. Data types

7.1 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>
<thead>
<tr>
<th>Data type</th>
<th>Default domain</th>
<th>Signed/Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>(-2^{31} .. (2^{31}-1))</td>
<td>Signed</td>
</tr>
<tr>
<td>bit</td>
<td>0..1</td>
<td>Unsigned</td>
</tr>
</tbody>
</table>

7.1.1 DSL syntax

The DSL syntax for scalars is shown in Syntax 3.

```
integer_type ::= integer_atom_type [ [ expression [ :
                   expression |
                   , open_range_value { , open_range_value } |
                   .. expression { , open_range_value } ] ] ]
integer_atom_type ::= inte
                   [ bit ]
open_range_value ::= 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 17.2), any X or Z values are converted to 0.

7.1.2 C++ syntax

Contrasting with 7.1.1, 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, Syntax 8, Syntax 9, Syntax 10, Syntax 11, and Syntax 12.
using bit = unsigned int;

Syntax 4---C++: bit declaration

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);
};

Syntax 5---C++: Scalar width declaration

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& 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

Syntax 6---C++: Scalar range declaration
/// Primary template for enums and structs
template <class T>
class rand_attr : public detail::RandAttrTBase {
public:
    /// Constructor
    rand_attr (const scope& name);
    /// Constructor and initial value
    rand_attr (const scope& name, const T& init_val);
    /// Copy constructor
    rand_attr(const rand_attr<T>& other);
    /// Struct access
    T* operator-> ();
    /// Struct access
    T& operator* ();
    /// enum 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 and initial value
rand_attr (const scope& name, const int& init_val);
/// Constructor defining width
rand_attr (const scope& name, const width& a_width);
/// Constructor defining width and initial value
rand_attr (const scope& name, const width& a_width, const int& init_val);
/// Constructor defining range
rand_attr (const scope& name, const range<int>& a_range);
/// Constructor defining range and initial value
rand_attr (const scope& name, const range<int>& a_range, const int& init_val);
/// Constructor defining width and range
rand_attr (const scope& name, const width& a_width, const range<int>& a_range);
/// Constructor defining width and range and initial value
rand_attr (const scope& name, const width& a_width, const range<int>& a_range,
          const int& init_val);
/// 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);

};

Syntax 8—C++: Scalar rand int declaration
/// Template specialization for scalar rand bit

template <>
class rand_attr<bit> : public detail::RandAttrBitBase {

public:

/// Constructor
rand_attr (const scope& name);

/// Constructor and initial value
rand_attr (const scope& name, const bit& init_val);

/// Constructor defining width
rand_attr (const scope& name, const width& a_width);

/// Constructor defining width and initial value
rand_attr (const scope& name, const width& a_width, const bit& init_val);

/// Constructor defining range
rand_attr (const scope& name, const range<bit>& a_range);

/// Constructor defining range and initial value
rand_attr (const scope& name, const range<bit>& a_range, const bit& init_val);

/// Constructor defining width and range
rand_attr (const scope& name, const width& a_width, const range<bit>& a_range);

/// Constructor defining width and range and initial value
rand_attr (const scope& name, const width& a_width, const range<bit>& a_range,
          const bit& init_val);

/// 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);
};
/// 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* ();
  /// enum access
  T& val();
  /// Exec statement assignment
  detail::ExecStmt operator= (const detail::AlgebExpr& value);
};
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);
};
7.1.3 Examples

The DSL and C++ scalar data examples are shown in-line within this section.

Declare a signed variable that is 32-bits wide.

```plaintext
DSL:    int a;
C++:    attr<int> a{"a"};
```
Declare a signed variable that is 5-bits wide.

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

Declare an unsigned variable that is 5-bits wide.

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

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

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

7.2 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).

C++ uses attr<bool> or rand_attr<bool>.

7.3 enums

7.3.1 DSL syntax

The enum declaration is consistent with C/C++ and is a subset of SystemVerilog, as shown in Syntax 13.

| enum_declaration ::= enum enum_identifier { [ enum_item { , enum_item } ] [ ; ]
| enum_item ::= identifier [ = constant_expression ]

Syntax 13—DSL: enum declaration

7.3.2 C++ syntax

The corresponding C++ syntax for Syntax 13 is shown in Syntax 14.

The PSS_ENUM macro is used to encapsulate the PSS_CTOR macro and enum literal value declarations, using C-style enum declaration syntax.
### Syntax 14—C++: enum declaration

```cpp
/// 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);
    
#define PSS_ENUM(class_name, base_class, ...) 
    public:
        
        class_name (const scope& p) : base_class (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;
        }

#undef PSS_ENUM
```

#### 7.3.3 Examples

Examples of enum usage are shown in Example 3 and Example 4.
Example 3—DSL: enum data type

```cpp
class config_modes_e : public enumeration {
    PSS_ENUM(config_modes_e, enumeration, UNKNOWN, MODE_A=10, MODE_B=20);
};//
type_decl<config_modes_e> config_modes_e_decl;
```

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

Example 4—C++: enum data type

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

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

7.4 Strings

The PSS language supports a built-in string type with the type name `string`.

7.4.1 C++ syntax

C++ uses `attr<std::string>` (see Syntax 15) or `rand_attr<std::string>` (see Syntax 16) to represent strings.
7.4.2 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.
The corresponding C++ example for Example 5 is shown in 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**

```
struct string_s : public structure {
    PSS_CTOR(string_s, structure)
    rand_attr<bit> a {a};
    rand_attr<std::string> s {"s"};

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

    type_decl<string_s> string_s_decl;
}
```

**Example 6—C++: String data type**

### 7.5 chandles

The chandle type (pronounced “see-handle”) represents an opaque handle to a foreign-language pointer. A chandle is used with the PI (see 17.2) 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.

Example 7 shows a struct containing a chandle field that is initialized by the return of a foreign-language function.

```
import chandle do_init();

struct info_s {
    chandle     ptr;

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

**Example 7—DSL: chandle data type**
7.6 Structs

A struct declares a collection of data items and constraints that relate the values of the data items, as shown in Syntax 17 or Syntax 18.

7.6.1 DSL syntax

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

Syntax 17—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 constraints (see 13.1) or bins (see 14.7);
- represent data flow objects (see Clause 9) and resources (see Clause 10).

The following also apply.

a) Data elements within a struct may be declared to be a specific 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.

7.6.2 C++ syntax

In C++, structures shall derive from the structure class.
The corresponding C++ syntax for Syntax 17 is shown in Syntax 18.

```cpp
/// 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();
};

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;
};

struct axi4_trans_req : public structure {
PSS_CTOR(axi4_trans_req, 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} };
};
type_decl<axi4_trans_req> axi4_trans_req_decl;
```

Syntax 18—C++: struct declaration

7.6.3 Examples

Struct examples are shown in Example 8 and Example 9.

```cpp
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

```cpp
struct axi4_trans_req : public structure {
PSS_CTOR(axi4_trans_req, 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} };
};
type_decl<axi4_trans_req> axi4_trans_req_decl;
```

Example 9—C++: Struct with rand modifier

7.7 User-defined data types

The typedef statement declares a user-defined type name in terms of an existing data type, as shown in Syntax 19.
7.7.1 DSL syntax

typedef declaration ::= typedef data_type identifier ;

Syntax 19—DSL: User-defined type declaration

7.7.2 C++ syntax

C++ uses the built-in typedef construct.

7.7.3 Examples

typedef examples are shown in Example 10 and Example 11.

Example 10—DSL: typedef

typedef bit[31:0] uint32_t;

Example 11—C++: typedef

typedef unsigned int uint32_t;

7.8 Arrays

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

7.8.1 C++ syntax

The corresponding C++ syntax for arrays is shown in Syntax 20, Syntax 21, Syntax 22, Syntax 23, Syntax 24, Syntax 25, and Syntax 26.

/// Declare an array
namespace pss {
  template < class T>
  using vec = std::vector <T>;
}

Syntax 20—C++: array declaration
/** 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);

/// 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);
        /// 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> >;

Syntax 22—C++: Arrays of rand bits

Syntax 23—C++: Arrays of rand enums and rand structs
/// 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);
    /// 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);
/// 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;
};

Syntax 25—C++: Arrays of bits

/// 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> >;

Syntax 26—C++: Arrays of enums and structs
7.8.2 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

```
// 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

7.8.3 Properties

Arrays of scalar quantities provide properties, such as sum and size (see 7.8.3.1 and 7.8.3.2), that may be used in constraint expressions.

7.8.3.1 Sum

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

7.8.3.2 Size

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

7.8.3.3 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.
Example 16—DSL: size property of an array

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

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

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

Actions may be *atomic*, in which case their implementation is supplied via an *exec block* (see 17.1) or they may be *compound*, in which case they contain an *activity* (see Clause 12) 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 27. See also Syntax 28.

8.1 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
   | bins_declaration
   | symbol_declaration
   | coverspec_declaration
   | exec_block_stmt
```

*Syntax 27—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, 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 be extended. The extended actions may be instantiated like any other action. Abstract actions shall not be instantiated directly.

### 8.2 C++ syntax

Actions are declared using the `action` class.

The corresponding C++ syntax for Syntax 27 is shown in Syntax 28.

```cpp
/// Declare an action
class action : public detail::ActionBase {
protected:
  /// Constructor
  action ( const scope& s );
  /// Destructor
  ~action();
public:
  rand_attr<component*>& comp();
}; // class action
```

**Syntax 28—C++: action declaration**

### 8.3 Examples

For an example of using an `action`, see 12.2.3.
9. 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.

9.1 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 29 or Syntax 30.

9.1.1 DSL syntax

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) An action that inputs a buffer object shall be bound (connected) to an action that outputs a buffer object of the same type. The connected action can be explicitly created and connected by the user or inferred by the PSS processing tool.

d) An action that outputs a buffer object may be bound to one or more actions that input a buffer object of the same type. An action that outputs a buffer object is not required to be bound to an action that inputs a buffer object of the same type.

e) Execution of the producing action shall complete before the execution of the inputting action begins. The execution of the outputting action, and inputting action(s), if any, are sequential. See also Figure 1 (relative to Example 18 and Example 19).

9.1.2 C++ syntax

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

buffer identifier [ : struct_super_spec ] { { struct_body_item } } [ ; ]
9.1.3 Examples

Examples of buffer objects are shown in Example 18 and Example 19.

```cpp
// 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();
};

struct mem_segment_s {
    rand int[4..1024] size;
    rand bit[63:0] addr;
};

buffer data_buff_s {
    rand mem_segment_s seg;
};

component top {
    action cons_mem_a {
        input data_buff_s in_data;
    };

    action prod_mem_a {
        output data_buff_s out_data;
    };
}
```

Example 18—DSL: buffer object
9.2 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 31 or Syntax 32.
9.2.1 DSL syntax

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

Syntax 31—DSL: stream declaration

The following also apply.

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

b) An action that inputs a stream object shall be bound to a single action that outputs a stream object of the same type.

c) An action that outputs a stream object shall be bound to a single action that inputs a stream object of the same type.

d) The outputting and inputting actions are executed in parallel. The semantics of parallel execution are discussed further in 12.3.3. See also Figure 2 (relative to Example 20 and Example 21).

9.2.2 C++ syntax

The corresponding C++ syntax for Syntax 31 is shown in Syntax 32.

```
/// 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();
};
```

Syntax 32—C++: stream declaration

9.2.3 Examples

Examples of stream objects are show in Example 20 and Example 21.
Example 20—DSL: stream object

```markdown
struct mem_segment_s {
    rand int[4..1024] size;
    rand bit[63:0] addr;
}

stream data_buff_s {
    rand mem_segment_s seg;
}

component top {
    action cons_mem_a {
        input data_buff_s in_data;
    }
    action prod_mem_a {
        output data_buff_s out_data;
    }
}
```

Example 21—C++: stream object

```markdown
struct mem_segment_s : public structure {
    PSS_CTOR(mem_segment_s, structure);
    rand_attr<int> size {"size", range<>(4,1024)};
    rand_attr<bit> addr {"addr", width(63,0)};
};
type_decl<mem_segment_s> mem_segment_s_decl;
struct data_buff_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;
struct top : public component{
    PSS_CTOR(top, component);
    struct cons_mem_a : public action {
        PSS_CTOR (cons_mem_a, action);
        input<data_buff_s> in_data {"in_data"};
    }; 
type_decl<cons_mem_a> cons_mem_a_decl;
    struct prod_mem_a : public action {
        PSS_CTOR (prod_mem_a, action);
        output<data_buff_s> out_data {"out_data"};
    }; 
type_decl<prod_mem_a> prod_mem_a_decl;
}; // struct top
```
9.3 State objects

State objects represent the state of some entity in the execution environment at a given time. See Syntax 33 or Syntax 34.

9.3.1 DSL syntax

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

Syntax 33—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 structs, 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 11.7.6 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. bind directives are used to associate the action with the appropriate state-object pool (see 11.7.4).
- 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 on 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 before the execution of any inputting action begins. Execution of actions that produce a state object shall be sequential.
9.3.2 C++ syntax

The corresponding C++ syntax for Syntax 33 is shown in Syntax 34.

```cpp
/// 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();
};
```

Syntax 34—C++: state declaration

9.3.3 Examples

Examples of state objects are show in Example 22 and Example 23.

```cpp
component IOdev_c {
    enum speed_e {SLOW, FAST};
    state config_s {
        rand speed_e speed;
        constraint initial -> speed == SLOW;
    };
    pool config_s config_var;
    bind config_var *;

    action setup {
        output config_s next_cfg;
    };

    action traffic {
        rand int[1,2,4,8] rate;
        input config_s curr_cfg;
        constraint rate == 8 -> curr_cfg.speed == FAST;
    };
};
```

Example 22—DSL: state object
Example 23—C++: state object

```cpp
class IOdev_c : public component {
public:
    PSS_CTOR(IOdev_c, component);
    class speed_e : public enumeration {
        PSS_ENUM(speed_e, enumeration, SLOW, FAST);
    };
    struct config_s : public state {
        PSS_CTOR(config_s, state);
        rand_attr<speed_e> speed("speed");
        constraint init { if_then {initial(), speed==speed_e::SLOW}};
    };
    type_decl<config_s> config_s_decl;
    pool<config_s> config_var("config_var");
    bind b {config_var};
    class setup : public action {
        public:
            PSS_CTOR(setup, action);
            output<config_s> next_cfg("next_cfg");
    };
    type_decl<setup> setup_decl;
    class traffic : public action {
        public:
            PSS_CTOR(traffic, action);
            rand_attr<int> rate("rate", range<>(1)(2)(4)(8));
            input<config_s> curr_cfg;
            constraint c {if_then {rate==8, curr_cfg->speed==speed_e::FAST}};
    };
    type_decl<traffic> traffic_decl;
};
type_decl<IOdev_c> IOdev_c_decl;
```

9.4 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 35 or Syntax 36 and Syntax 37.

9.4.1 DSL syntax

```
input | output action_data_declaration
```

Syntax 35—DSL: Flow object reference

9.4.2 C++ syntax

Action input and outputs are defined using the input (see Syntax 36) and output (see Syntax 36) classes respectively.

The corresponding C++ syntax for Syntax 35 is shown in Syntax 36 and Syntax 37.
9.4.3 Examples

For examples of how to use buffer or stream objects, see 9.1.3 or 9.2.3, respectively.

9.5 Implicitly binding flow objects

Input and output object bindings may be inferred from the context of the activity description (see Annex E). If an action is traversed in an activity that does not explicitly bind its input(s) or output(s), binding needs to be inferred to satisfy the rules in 9.4. This may involve executing actions that are not explicitly traversed in the activity or binding to other actions that are traversed. In all cases, binding two actions shall be such that the output of one action is type-compatible with the input of another, scheduling restrictions are accommodated, and any constraints are satisfied. Inferred binding behaves as if the binding was specified explicitly using the `bind` statement (see 11.7.4).
10. Resource objects

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

10.1 Declaring resource objects

Resource struct types can inherit from previously defined unqualified structs or resource structs. See Syntax 38 or Syntax 39.

10.1.1 DSL syntax

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

Syntax 38—DSL: resource declaration

The following also apply.

a) Resources have a built-in numeric non-negative attribute called instance_id (see 11.7.5). 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 11.7.

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.

10.1.2 C++ syntax

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

```cpp
/// 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();
};
```

Syntax 39—C++: resource declaration
10.1.3 Examples

For example of how to declare a resource, see 10.2.3.

10.2 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 40 or Syntax 41 and Syntax 42.

10.2.1 DSL syntax

For example of how to declare a resource, see 10.2.3.

10.2.2 C++ syntax

The corresponding C++ syntax for Syntax 40 is shown in Syntax 41 and Syntax 42.

```cpp
/// 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* ();
};
```

Syntax 41—C++: Claim a locked resource

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 it.
10.2.3 Examples

Example 24 and Example 25 demonstrate resource claims in lock and share mode. Action `mem_copy` claims exclusive access to one `CPU_core_s` instance out of a pool of four. Action `two_DMA_chan_transfer` claims exclusive access to two different `DMA_channel_s` instances out of a pool of 32. It also claims one `CPU_core_s` instance, but in share mode, i.e., not excluding its assignment to other concurrent actions, given that it too is in share mode.

```cpp
component sys_c {
    resource DMA_channel_s {}
    pool[32] DMA_channel_s Chan_pool;
    bind Chan_pool *;
    resource CPU_core_s {}
    pool[4] CPU_core_s core_pool;
    bind core_pool *;
    action mem_copy {
        lock CPU_core_s core;
    };
    action two_chan_transfer {
        lock DMA_channel_s chan_A;
        lock DMA_channel_s chan_B;
        share CPU_core_s ctrl_core;
    };
}
```

Example 24—DSL: Resource object
Example 25—C++: Resource object

class sys_c : public component {
public:
    PSS_CTOR(sys_c, component);
struct DMA_channel_s : public resource {
    PSS_CTOR(DMA_channel_s, resource);
};
type_decl<DMA_channel_s> DMA_channel_s_decl;
pool<DMA_channel_s> chan_pool {"chan_pool", 32};
bind b1 { chan_pool };
struct CPU_core_s : public resource {
    PSS_CTOR(CPU_core_s, resource);
};
type_decl<CPU_core_s> CPU_core_s_decl;
pool<CPU_core_s> core_pool {"core_pool", 4};
bind b2 { core_pool };
class mem_copy : public action {
public:
    PSS_CTOR(mem_copy, action);
    lock<CPU_core_s> core {"core"};
};
type_decl<mem_copy> mem_copy_decl;
class two_chan_transfer : public action {
public:
    PSS_CTOR(two_chan_transfer, action);
    lock<DMA_channel_s> chan_A {"chan_A"};
    lock<DMA_channel_s> chan_B {"chan_B"};
    share<CPU_core_s> ctrl_core {"core"};
};
type_decl<two_chan_transfer> two_chan_transfer_decl;
};
type_decl<sys_c> sys_c_decl;
11. Components and pools

Components and pools 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. Pools represent collections of resources, state variables, and connectivity for data-flow purposes.

Components are structural entities, defined per type and instantiated under other components (see Syntax 43 or Syntax 44, Syntax 45, and Syntax 46). Component instances constitute a hierarchy (tree structure), beginning with the top or root component, called pss_top. Components have unique identities corresponding to their hierarchical path, but no data-attributes or constraints of their own. Components may also encapsulate imported functions (see 17.2.1) and imported class instances (see 17.7).

Pools, too, are structural entities instantiated under components. They are used to determine the accessibility actions 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 11.7.4).

11.1 DSL syntax

```plaintext
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 43—DSL: component declaration

11.2 C++ syntax

The corresponding C++ syntax for Syntax 43 is shown in Syntax 44, Syntax 45, and Syntax 46.

Components are declared using the component class (see Syntax 44).
Components are instantiated using the `comp_inst<>` class (see Syntax 45).

Arrays of components are instantiated using the `comp_inst_vec<>` class (see Syntax 46).
11.3 Examples

For examples of how to use a component, see 11.5.2.

11.4 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.

11.5 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.

11.5.1 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, pss_top. Other components or actions are instantiated (directly or indirectly) under pss_top. See also Example 26 and Example 27.

c) 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 function calls. The component tree is elaborated to instantiate each component and then the exec init blocks are evaluated bottom-up. See also Example 28 and Example 29.
d) 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 30 and Example 31.

11.5.2 Examples

Example 26 and Example 27 depict a component tree definition. In total, there is one instance of multimedia_ss_c, four instances of codec_c, and eight instances of vid_pipe_c.

```c
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 26—DSL: Component instantiation**

```c
class vid_pipe_c : public component {PSSCTOR(vid_pipe_c, component);};
type_decl<vid_pipe_c> vid_pipe_c_decl;
class codec_c : public component { 
    PSSCTOR(codec_c, component);
    comp_inst<vid_pipe_c> pipeA("pipeA"), pipeB("pipeB");
    class decode : public action { PSSCTOR(decode, action); };
    type_decl<decode> decode_decl;
};
type_decl<codec_c> codec_c_decl;
class multimedia_ss_c : public component { 
    PSSCTOR(multimedia_ss_c, component);
    comp_inst_vec<codec_c> codecs{ "codecs", 4};
};
type_decl<multimedia_ss_c> multimedia_ss_c_decl;
class pss_top : public component { 
    PSSCTOR(pss_top, component);
    comp_inst<multimedia_ss_c> multimedia_ss{"multimedia_ss"};
};
type_decl<pss_top> pss_top_decl;
```

**Example 27—C++: Component instantiation**

In Example 28 and Example 29, 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.

```plaintext
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)
```

```plaintext
class sub_c : public component {
    PSS_CTOR(sub_c, component);
    attr<int> base_addr {"base_addr"};
    exec e { exec::init,
        base_addr = 0x1000
    };
};
type_decl<sub_c> sub_c_decl;

class pss_top : public component {
    PSS_CTOR(pss_top, component);
    comp_inst<sub_c> s1{"s1"}, s2{"s2"};
    exec e { exec::init,
        s1->base_addr = 0x2000
    };
};
type_decl<pss_top> pss_top_decl;
```

**Example 28—DSL: Data initialization in a component**

**Example 29—C++: Data initialization in a component**

In **Example 30** and **Example 31**, component `pss_top` contains two instances of component `sub_c`. Component `sub_c` contains a data field named `base_addr` that controls offset `addr` when action `sub_c::B` traverses action `A`.

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::B` twice. Depending on which component instance `sub_c::B` is associated with during traversal, it will cause `sub_c::A` to be associated with a different `base_addr`. 
— If sub_c::B executes in the context of top_c.s1, sub_c::A uses 0x1000.
— If sub_c::B executes in the context of top_c.s2, sub_c::A uses 0x2000.

```plaintext
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 30—DSL: Accessing component data field from an action*
11.6 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.

11.6.1 Semantics

A compound action can only create 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.
other words, compound actions cannot instantiate actions that are defined in components outside their context component hierarchy.

11.6.2 Examples

Example 32 and Example 33 demonstrate the use of the comp attribute. The first constraint compares the action’s component instance using a global static path. The constraint within the activity forces the action to be associated with a specific sub-component. It uses a static path relative to the component instance of its containing action.

For action C1::A1 to contain action C2::A1, component C2 needs to be instantiated somewhere under C1.

```plaintext
component codec_c {
    vid_pipe_c pipeA, pipeB;

    action decode {
        constraint {
            mode == AX -> comp != pss_top.multimedia_ss.codecs[0];
        }

        vid_pipe_c::program pipe_prog_a;

        activity {
            pipe_prog_a with {comp == this.comp.pipeA;};
        }
    }
}
```

Example 32—DSL: Constraining a comp attribute
Example 33—C++: Constraining a comp attribute

Consider the code in Example 34 and Example 35. It instantiates four instances of codec_c and, therefore, four instances of vid_pipe_c. Action multi_activate expands to multiple activate actions. These are all associated with the same vid_pipe_c instance that is instantiated under the codec_c instance with which their parent compound action is associated.

Example 34—DSL: Sub-action component assignment

```cpp
class codec_c : public component {
  PSS_CTOR(codec_c, component);
  comp_inst<vid_pipe_c> pipeA("pipeA"), pipeB("pipeB");
  class decode : public action {
    PSS_CTOR(decode, action);
    rand_attr<modes_e> mode {"mode"};
    // TODO: we need a way to access pss_top globally
    // constraint cl {
    //   if_then {
    //     mode == modes_e::AX,
    //     comp() != pss_top->multimedia_ss->codecs[0];
    //   }
    // }
    // }
    action_handle<vid_pipe_c::program> pipe_prog_a("pipe_prog_a");
    activity act {
      pipe_prog_a.with(
        pipe_prog_a->comp()==static_cast<codec_c*>(comp().val())->pipeA
      );
    };
    type_decl<decode> decode_decl;
  };
  type_decl<codec_c> codec_c_decl;
}
```
11.7 Pool instantiation and static binding

Pools are used to determine possible assignment of objects to actions, and, thus, shape the space of legal test scenarios. 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.

11.7.1 DSL syntax

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.
11.7.2 C++ syntax

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

```cpp
// Declare a pool
template <class T>
class pool : public detail::PoolBase {
public:
    pool (const scope& name, std::size_t count = 1);
};
```

Syntax 48—C++: Pool instantiation

11.7.3 Examples

For an example of pool usage, see 11.7.4.3.

11.7.4 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 49 or Syntax 50). Binding is done relative to the component sub-tree of the component type in which the `bind` directive occurs.

11.7.4.1 DSL syntax

```plaintext
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 49—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).

11.7.4.2 C++ syntax

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

```cpp
/// Declare a bind
class bind : public detail::BindBase {
    public:
        /// Bind a resource to multiple targets
        template <class R /*resource*/, typename... T
                   /*comp_inst/input/output/lock/share*/>
            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();
};
```

Syntax 50—C++: Static bind directives

11.7.4.3 Examples

Example 36 and Example 37 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.
Example 36—DSL: Pool binding

```plaintext
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 prev;
    output power_state_s next;
    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;
}
```
11.7.5 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.

For example, in Example 38 and Example 39, 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 \texttt{par_dma_xfers}, the two transfer actions can be assigned the same channel \texttt{instance_id} because they are associated with different \texttt{dma_c} instances. However, these same two actions need to be assigned a different \texttt{cpu_core_s} object, with a different \texttt{instance_id}, because both \texttt{dma_c} instances are bound to the same resource pool of \texttt{cpu_core_s} objects defined under \texttt{pss_top} and they are scheduled in parallel. The \texttt{bind} directive designates the pool of \texttt{cpu_core_s} resources is to be utilized by both instances of the \texttt{dma_c} component.

```dsl
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;

        activity {
            parallel {
                xfer_a;
                xfer_b;
                constraint xfer_a.comp != xfer_b.comp;
                constraint xfer_a.chan.instance_id ==
                xfer_b.chan.instance_id; // OK
                constraint xfer_a.core.instance_id ==
                xfer_b.core.instance_id; // conflict!
            }
        }
    }
}
```

Example 38—DSL: Resource object assignment
Example 39—C++: Resource object assignment

11.7.6 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.
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 40 and Example 41. 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 inside [A, B];
  }
}

component pss_top {
  codec_c codec0, codec1;
  action sys_configure {
    activity {
      do codec_c::configure with {next_conf.mode == A;};
      do codec_c::configure with {next_conf.mode == B;};
      // OK, but only on a different codec instance
    }
  }
}
```

Example 40—DSL: State object binding
11.7.7 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 11.7.4). 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

```cpp
class codec_config_mode_e : public enumeration {
    PSS_ENUM(codec_config_mode_e, enumeration, UNKNOWN, A, B);
};

type_decl<codec_config_mode_e> codec_config_mode_e_decl;

class codec_c : public component {
    PSS_CTOR(codec_c, component);
    struct configuration_s : public state {
        PSS_CTOR(configuration_s, state);
        rand_attr<codec_config_mode_e> mode ("mode");
        constraint c1 {
            if_then {
                initial(),
                mode == codec_config_mode_e::UNKNOWN
            }
        }
    };
    type_decl<configuration_s> configuration_s_decl;
    pool <configuration_s> config_var ( "config_var" );
    bind b1 { config_var ];
    class configure_a : public action {
        PSS_CTOR( configure_a, 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 &&
            inside ( next_conf->mode,
                range<codec_config_mode_e>
                    (codec_config_mode_e::A)
                    (codec_config_mode_e::B) )
        };
    };
    type_decl<configuration_s> configuration_s_decl;
    class configure_a : public action {
        PSS_CTOR( configure_a, 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 &&
            inside ( next_conf->mode,
                range<codec_config_mode_e>
                    (codec_config_mode_e::A)
                    (codec_config_mode_e::B) )
        };
    };
    type_decl<configuration_s> configuration_s_decl;
    class codec_c : public component {
        PSS_CTOR(codec_c, component);
        comp_inst <codec_c> codec0 ("codec0"), codec1("codec1");
    class sys_configure_a : public action {
        PSS_CTOR(sys_configure_a, action);
        action_handle<codec_c::configure_a> config_A ("config_A");
        action_handle<codec_c::configure_a> config_B ("config_B");
        activity act {
            config_A.with(config_A->next_conf->mode == codec_config_mode_e::A),
            config_B.with(config_B->next_conf->mode == codec_config_mode_e::B)
                // OK, but only on a different codec instance
        };
    };
    type_decl<sys_configure_a> sys_configure_a_decl;
};

class pss_top : public component {
    PSS_CTOR(pss_top, component);
    comp_inst <codec_c> codec0 ("codec0"), codec1("codec1");
    class sys_configure_a : public action {
        PSS_CTOR(sys_configure_a, action);
        action_handle<codec_c::configure_a> config_A ("config_A");
        action_handle<codec_c::configure_a> config_B ("config_B");
        activity act {
            config_A.with(config_A->next_conf->mode == codec_config_mode_e::A),
            config_B.with(config_B->next_conf->mode == codec_config_mode_e::B)
                // OK, but only on a different codec instance
        };
    };
    type_decl<pss_top> pss_top_decl;
```

Example 41—C++: State object binding

11.7.7 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 11.7.4). 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 9.3).

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

In Example 42, the first constraint inside 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 inside { prev.domain_B - 1,
        prev.domain_B,
        prev.domain_B + 1};

    constraint prev.domain_C==0 -> domain_C inside{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 42—DSL: Sequencing constraints
12. Activities

When an action includes multiple operations, these behaviors are described within the action using an activity.

12.1 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.

12.2 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 51 or Syntax 53.

12.2.1 DSL syntax

Named sub-activities, introduced through statement labels, allow referencing action-handles using hierarchical paths. Reference can be made to an action-handle from within the same activity, from the context 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 lexical scope. Action-handles declared in unnamed activity scope cannot be accessed.

Note that the top activity scope is unnamed. For an action-handle to be accessible directly in the top-level action scope or from outside, it needs to be declared at the top-level action scope.

```
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 51—DSL: activity statement

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 52 or Syntax 54). The symbol may be used as a node in the activity.

A symbol may activate another symbol, but symbols may not activate themselves (no recursion).
12.2.2 C++ syntax

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

The corresponding C++ syntax for Syntax 51 is shown in Syntax 53.

```cpp
// Declare an activity
class activity : public detail::ActivityBase {
public:
  // Constructor
  template < class... R >
  activity(R&&... /* detail::ActivityStmt */ r);
  // Constructor
  activity(const std :: vector<detail::ActivityStmt*>& stmts );
  // Destructor
  ~activity();
};
```

Syntax 53—C++: activity statement

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

The corresponding C++ syntax for Syntax 52 is shown in Syntax 54.

```cpp
using symbol = detail::ActivityStmt;
symbol symbolName (parameters...) { return /* subactivity */ ;}
```

Syntax 54—C++: symbol declaration

12.2.3 Examples

Example 43 and Example 44 depict using a symbol. In this case, the desired activity is a sequence of choices between $aN$ and $bN$, followed by a sequence of $cN$ 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 $aN$ and $bN$ selections. The symbol is then referenced in the top-level activity, which has the same effect as specifying the $aN/bN$ 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 43—DSL: Using a symbol

class A : public action { PSS_CTOR(A,action); };
type_decl<A> A_decl;
class B : public action { PSS_CTOR(B,action); };
type_decl<B> B_decl;
class C : public action { PSS_CTOR(C,action); };
type_decl<C> C_decl;
class top : public action {
    PSS_CTOR(top,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
    };
    type_decl<top> top_decl;
}

Example 44—C++: Using a symbol
12.3 Action scheduling statements

By default, action 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.

12.3.1 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 55 or Syntax 56). The action being traversed may be an action-type field that was previously declared. The action being traversed may also be specified by type, in which case the action instance is anonymous.

12.3.1.1 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 55—DSL: Variable traversal statement
```

identifier names a unique new variable in the context of the containing action type (in the first syntactic variant) or a declared non-rand field of the containing action (in the second variant).

The following also apply.

a) Intuitively, 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 with the action modifier. In the latter case, it is randomized, but has no observed execution or duration.

b) An action instance may be traversed without explicitly creating an action handle by using the anonymous action traversal variant, specifying the keyword do followed by the action-type specifier and an optional in-line constraint. 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 (e.g., sequential or parallel) of the containing scope.

d) Other aspects that impact action-evaluation scheduling, are covered via binding inputs or outputs (see Clause 9), resource claims (see Clause 10), or attribute value assignment (see Clause 8).

12.3.1.2 C++ syntax

The corresponding C++ syntax for Syntax 55 is shown in Syntax 56.
12.3.1.3 Examples

Example 45 and Example 46 show an example of traversing an atomic action variable. Action A is an atomic action, whose exec body block calls a PI function to set the value selected for 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 calls to the set_val import function.
Example 46—C++: Action traversal

Example 47 and Example 48 show an example of traversing a compound action as well as a non-random non-action field. The activity for action C traverses the non-random, non-action field max, then traverses the action-type field b1. Evaluating this activity results in a 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.
Example 47—DSL: Compound action traversal

```plaintext
action A {
    rand bit[3:0] f1;
    exec body {
        set_val(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;
        }
    }
}
```
12.3.2 Sequential block

An activity sequence block statement specifies sequential scheduling between sub-activities (see Syntax 57 or Syntax 58).

12.3.2.1 DSL syntax

```c++
import_func set_val { "set_val",
    { import_func::in<bit>("p1", width(3, 0)) }
};
class A : public action {
    PSS_CTOR(A, action);
    rand_attr<bit> f1 {"f1", width(3, 0)};
    exec e { exec::body,
        set_val (f1)
    };
};
type_decl<A> A_decl;
class B : public action {
    PSS_CTOR(B, action);
    action_handle<A> a1{"a1"}, a2{"a2"};
    activity a {
        a1,
        a2.with(a2->f1 < 10)
    };
};
type_decl<B> B_decl;
class C : public action {
    PSS_CTOR(C, action);
    action_attr<bit> max {"max", width(3, 0)};
    action_handle<B> bl{"bl"};
    activity a {
        sequence {
            max,
            bl.with(bl->a2->f1 <= max)
        }
    };
};
type_decl<C> C_decl;
```

Example 48—C++: Compound action traversal

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 \(\text{activity}_1 \ldots \text{activity}_n\), keeping all scheduling dependencies within the sets and introducing additional dependencies between them to obtain sequential scheduling (see 5.3.2).
c) Sequential scheduling does not rule out other inferred dependencies affecting the nodes in these sub-activities. In particular, there may be cases where additional action-executions need to be scheduled in between sub-activities of subsequent statements.

12.3.2.2 C++ syntax

The corresponding C++ syntax for Syntax 57 is shown in Syntax 58.

```
/// Declare a sequence block
class sequence : public detail::ActivityStmt {
public:
  // Constructor
  template < class... R >
  sequence(R&&... /* detail::ActivityStmt */ r);
  sequence(const std::vector<detail::ActivityStmt*>& stmts);
};
```

`Syntax 58—C++: Activity sequence block`

12.3.2.3 Examples

Assume \(A\) and \(B\) are action types that have no rules or nested activity (see Example 49 and Example 50).

Action `my_test` specifies one execution of action \(A\) and one of action \(B\) with the scheduling dependency \((A) \rightarrow (B)\); the corresponding observed behavior is \(\{\text{start } A, \text{ end } A, \text{ start } B, \text{ end } B\}\).

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 \((A) \rightarrow (B)\) and \((C) \rightarrow (B)\) and multiple behaviors are possible, such as \(\{\text{start } C, \text{ start } A, \text{ end } A, \text{ end } C, \text{ start } B, \text{ end } B\}\).

Finally, assume also \(C\) has a state precondition which only \(A\) can establish. Dependencies in this case are \((A) \rightarrow (B), (A) \rightarrow (C)\) and \((C) \rightarrow (B)\) (note that the first pair can be reduced) and, consequently, the only possible behavior is \(\{\text{start } A, \text{ end } A, \text{ start } C, \text{ end } C, \text{ start } B, \text{ end } B\}\).
12.3.3 parallel

The *parallel* statement specifies sub-activities that execute concurrently (see Syntax 59 or Syntax 60).

12.3.3.1 DSL syntax

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

Syntax 59—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 activity branch does not wait for the completion of any action in another.

b) Formally, the *parallel* statement specifies parallel scheduling between the sets of action-executions per the evaluation of `activity_stmt1 .. activity_stmtn`, 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 5.3.2).

12.3.3.2 C++ syntax

The corresponding C++ syntax for Syntax 59 is shown in Syntax 60.

```
class my_test : public action {
  PSS_CTOR(my_test, action);
  action_handle<A> a("a");
  action_handle<B> b("b");
  activity act {
    a,
    b
  };
  type_decl<my_test> my_test_decl;
  
  Example 50—C++: Sequential block

  class my_test : public action {
    PSS_CTOR(my_test, action);
    action_handle<A> a("a");
    action_handle<B> b("b");
    activity act {
      a,
      b
    };
    type_decl<my_test> my_test_decl;
  }
```

Syntax 60—C++: Parallel statement
12.3.3.3 Examples

Assume A, B, and C are action types that have no rules or nested activity (see Example 51 and Example 52).

The activity in action my_test specifies two dependencies (A) -> (B) and (A) -> (C). Since the executions of both B and C have the exact same scheduling dependencies, their invocation is synchronized.

Now assume action type C inputs a buffer object and action 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, this activity shall be illegal.

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

Example 51—DSL: Parallel statement

```
class my_test : public action {
    PSS_CTOR(my_test, action);
    action_handle<A> a("a");
    action_handle<B> b("b");
    action_handle<C> c("c");
    activity act {
        a,
        parallel {
            b,
            c
        }
    };
    type_decl<my_test> my_test_decl;
};
```

Example 52—C++: Parallel statement

The semantics of the parallel construct require the sequences \{a, b\} and \{c, d\} to start execution at the same time (see Example 53 and Example 54). The semantics of the sequential block require the execution of \b\ follows \a\ and \d\ follows \c\. It shall be illegal for \a\ and \d\ to be assigned the same instance of the resource R, since they are executed in separate sub-blocks of the parallel statement and there may be no scheduling dependencies between sub-blocks. Thus, if resource type R had one instance instead of four in the code snippet, the activity specified in my_test would be illegal.
component top {
    resource R {};
    pool[4] R R_pool;
    bind R_pool *;

    action A { lock R r; }
    action B {}
    action C {}
    action D { lock R r; }

    action my_test {
        activity {
            parallel {
                {do A; do B;}
                {do C; do D;}
            }
        }
    }
}
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 61 or Syntax 62.

12.3.4.1 DSL syntax

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

Syntax 61—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_stmt1 .. activity_stmtn`, 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.

### 12.3.4.2 C++ syntax

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

```cpp
class schedule : public detail::ActivityStmt {
    public:
        // Constructor
        template <class... R>
        schedule(R&&... /* detail::ActivityStmt */ r);
        schedule(const std::vector<detail::ActivityStmt*>& stmts);
    }
```

**Syntax 62—C++: Schedule statement**

### 12.3.4.3 Examples

Consider the code in Example 55 and Example 56, which are similar to Example 51 and Example 52, 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.

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

**Example 55—DSL: Schedule statement**
In contrast, consider the code in Example 57 and Example 58. 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 there were only a single instance of the \( R \) resource, \( a \) and \( d \) 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.
12.4 Activity control-flow constructs

The simplest activity procedural constructs are action instances listed sequentially in the activity clause. These action instances are traversed sequentially. In addition to simple sequences, repetition and branching statements can be used inside the activity clause.

12.4.1 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 63 or Syntax 64) and 12.4.2 describes the while-expression variant.
12.4.1.1 DSL syntax

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

Syntax 63—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 5.3.2).

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

12.4.1.2 C++ syntax

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

Syntax 64—C++: repeat-count statement

In Example 59 and Example 60, the resulting execution is six sequential action executions, alternating A’s and B’s, with five scheduling dependencies: \((A_{i0}) \rightarrow (B_{i0}), (B_{i0}) \rightarrow (A_{i1}), (A_{i1}) \rightarrow (B_{i2}), (B_{i2}) \rightarrow (A_{i2}), (B_{i3}) \rightarrow (A_{i3})\).
Example 59—DSL: repeat statement

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

Example 60—C++: repeat statement

```plaintext
class my_test : public action {
    PSS_CTOR(my_test, action);
    action_handle<A> a("a");
    action_handle<B> b("b");
    activity act {
        repeat { 3,
            sequence {
                a,
                b
            }
        }
    };
    type_decl<my_test> my_test_decl;
};
```

Example 61 and Example 62 show additional example of using `repeat`-count.

Example 61—DSL: Another repeat statement

```plaintext
action my_test {
    my_action1    action1;
    my_action2    action2;
    activity {
        repeat (i : 10) {
            if ((i % 4) == 0) {
                action1;
            } else {
                action2;
            }
        }
    }
};
```
12.4.2 repeat while

In the repeat while and repeat … while forms, iteration continues while the expression evaluates to true (see Syntax 65 or Syntax 66). See also Example 63 and Example 64.

12.4.2.1 DSL syntax

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

Syntax 65—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.

12.4.2.2 C++ syntax

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

---

Example 62—C++: Another repeat statement

```c++
class my_test : public action {
    PSS_CTOR(my_test, 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
            }
        }
    }
};
type_decl<my_test> my_test_decl;
```
12.4.2.3 Examples

```plaintext
class repeat_while : public detail::ActivityStmt {
    public:
        // Declare a repeat while statement
        repeat_while(const detail::AlgebExpr& cond,
                     const detail::ActivityStmt& activity);
    }

class do_while : public detail::ActivityStmt {
    public:
        // Declare a do while statement
        do_while(const detail::ActivityStmt& activity,
                 const detail::AlgebExpr& cond);
    }

component top {
    import 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);
        }
    }
}
```

`Syntax 66—C++: repeat-while statement`

`Example 63—DSL: repeat while statement`
12.4.3 foreach

The **foreach** construct iterates across the elements of an array (see Syntax 67 or Syntax 68). See also Example 65 and Example 66.

### 12.4.3.1 DSL syntax

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

**Syntax 67—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.

### 12.4.3.2 C++ syntax

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

```cpp
/// Declare a foreach statement
class foreach : public detail::SharedExpr {

public:

    /// Declare a foreach activity statement
    foreach( const attr<int>& iter,
             const rand_attr<vec<int>>& array,
             const detail::ActivityStmt& activity );

    /// Declare a foreach activity statement
    foreach( const attr<int>& iter,
             const rand_attr<vec<bit>>& array,
             const detail::ActivityStmt& activity );

    /// Declare a foreach activity statement
    foreach( const attr<int>& iter,
             const attr<vec<int>>& array,
             const detail::ActivityStmt& activity );

    /// Declare a foreach activity statement
    foreach( const attr<int>& iter,
             const attr<vec<bit>>& array,
             const detail::ActivityStmt& activity );

};
```

*Syntax 68—C++: foreach statement*
12.4.3.3 Examples

Example 65—DSL: foreach statement

```cpp
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 66—C++: foreach statement

```cpp
class my_action1 : public action {
    PSS_CTOR(my_action1, action);
    rand_attr < bit > val ("val", range<bit> {0, 3});
    // ...
};
type_decl<my_action1> my_action1_decl;

class my_test : public action {
    PSS_CTOR(my_test, 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] )
    }
}
type_decl<my_test> my_test_decl;
```

12.4.4 select

The `select` statement specifies a branch point in the traversal of the activity (see Syntax 69 or Syntax 70).
12.4.4.1 DSL syntax

```
activity_select_stmt ::= select { activity_labeled_stmt activity_labeled_stmt
                                { activity_labeled_stmt } }
```

**Syntax 69—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_labeled_stmts. 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.

12.4.4.2 C++ syntax

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

```
// Declare a select statement
class select : public detail::ActivityStmt {
    public:
        template <class... R>
        select(R&&... /* detail::ActivityStmt */ r);
        select(const std::vector<detail::ActivityStmt*>& stmts);
};
```

**Syntax 70—C++: select statement**

12.4.4.3 Examples

In Example 67 and Example 68, 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 67—DSL: Select statement**
12.4.5 if-else

The if-else statement introduces a branch point in the traversal of the activity (see Syntax 71 or Syntax 72).

12.4.5.1 DSL syntax

The following also apply.

a) \textit{expression} shall be of type \texttt{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 \texttt{activity\_stmt} if \texttt{expression} evaluates to \texttt{true} or the second \texttt{activity\_stmt} (following else) if present and \texttt{expression} evaluates to \texttt{false}.

d) The scheduling relationships need only be met for one branch for each evaluation of the activity.

e) The choice of values to \texttt{rand} attributes figuring in the \texttt{expression} needs to be such that it yields legal execution scheduling.

12.4.5.2 C++ syntax

The corresponding C++ syntax for Syntax 71 is shown in Syntax 72.

```cpp
class my_test : public action {
    PSS_CTOR(my_test, action);
    action_handle<my_action1> action1("action1");
    action_handle<my_action2> action2("action2");

    activity act {
        select {
            action1,
            action2
        }
    };
};
type_decl<my_test> my_test_decl;
```

\textit{Example 68—C++: Select statement}
12.4.5.3 Examples

If the scheduling requirements for Example 69 and Example 70 required selection of the $b$ branch, then the value selected for $x$ needs to be $\leq 5$. 

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

Example 69—DSL: if-else statement
12.5 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.

12.5.1 DSL syntax

A named sub-activity is declared by labeling an activity statement, see Syntax 73.

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

Syntax 73—DSL: Labeled activity statement

12.5.2 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 71, some activity statements are labeled while others are not. The second occurrence of label 12 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.
12.5.3 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 72, 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.
12.6 Explicitly binding flow objects

Input and output objects may be explicitly connected to actions using the `bind` statement (see Syntax 74 or Syntax 75).

12.6.1 DSL syntax

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

Syntax 74—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 9.4, the connection defines the data flow between actions and the type of the flow object defines the scheduling and semantics of the connection.

12.6.2 C++ syntax

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

Syntax 75—C++: bind statement

```cpp
/// Declare a bind
class bind : public detail::BindBase {
  public:
    /// Bind a resource to multiple targets
    template <class R /*resource*/, 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();
};

Syntax 75—C++: bind statement
```

12.6.3 Examples

Examples of binding are shown in Example 73 and Example 74.

Example 73—DSL: bind statement

```cpp
struct S {};
action P {
  output S out;
};
action C {
  input S in;
};
action T {
  P p;
  C c;
  bind p.out c.in;
  activity {
    p,
    c
  };
};
```

Example 73—DSL: bind statement
Example 74—C++: bind statement

```c++
class S : public std::structure {
    PSS_CTOR(S, std::structure);
};
type_decl<S> S_decl;

class P : public std::action {
    PSS_CTOR(P, std::action);
    std::output<S> out {"out"};
};
type_decl<P> P_decl;

class C : public std::action {
    PSS_CTOR(C, std::action);
    std::input<S> in {"in"};
};
type_decl<C> C_decl;

class T : public std::action {
    PSS_CTOR(T, std::action);
    std::action_handle<P> p {"p"};
    std::action_handle<C> c {"c"};
    std::bind b1 { p->out, c->in };
    activity act {
        p,
        c
    };
};
type_decl<T> T_decl;
```
13. 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.

13.1 Algebraic constraints

13.1.1 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 76 or Syntax 77).

NOTE—As shown in 13.3.9, named dynamic constraints may be referenced as a node inside an activity.
13.1.1.1 DSL syntax

```plaintext
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 76—DSL: Member constraint declaration

13.1.1.2 C++ syntax

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

```plaintext
/// Declare a member constraint
class constraint : public detail::ConstraintBase {
    public:
        /// Declare an unnamed member constraint
        template <class... R> constraint ( 
            const R&... /*detail::AlgebExpr*/ expr
        );
        /// Declare a named member constraint
        template <class... R> constraint ( const std::string& name, 
            const R&... /*detail::AlgebExpr*/ expr
        );
    }
    /// Declare a dynamic member constraint
class dynamic_constraint : public detail::DynamicConstraintBase {
        public:
            /// Declare a named dynamic member constraint
            template <class... R> dynamic_constraint ( 
                const std::string& name, 
                const R&... /*detail::AlgebExpr*/ expr
            );
        }
```

Syntax 77—C++: Member constraint declaration
13.1.1.3 Examples

Example 75 and Example 76 declare a static constraint block, while Example 77 and Example 78 declare a dynamic constraint block. In the case of the static constraint, the name is optional.

```markdown
Example 75—DSL: Declaring a static constraint

```action A {
    rand bit[31:0]    addr;

    constraint addr_c {
        addr == 0x1000;
    }
}

Example 76—C++: Declaring a static constraint

class A : public action {
    public:
        PSS_CTOR(A,action);

        rand_attr < bit > addr {"addr", width {31, 0}};
        constraint addr_c { "addr_c", addr == 0x1000 };
    };

type_decl<A> A_decl;

Example 77—DSL: Declaring a dynamic constraint

```action B {
    action bit[31:0]    addr;

    dynamic constraint dyn_addr1_c {
        addr inside [0x1000..0x1FFF];
    }

    dynamic constraint dyn_addr2_c {
        addr inside [0x2000..0x2FFF];
    }
}
```
13.1.2 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 79 and Example 80 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 inside 1..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.
13.1.3 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 12.3.1).

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 83 and Example 84).

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 81 and Example 82 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.
Example 81—DSL: Action traversal in-line constraint

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

action B {
    A a1, a2, a3, a4;

    constraint c1 { a1.f inside [8..15]; }
    constraint c2 { a1.f == a4.f; }

    activity {
        a1;
        a2 with {
            f inside [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 82—C++: Action traversal in-line constraint

```c++
class A : public action {
    PSS_CTOR(A, action);
    rand_attr< bit > f {"f", width(3, 0)};
};
type_decl<A> A_decl;

class B : public action {
    PSS_CTOR(B, action);
    action_handle<A> a1{"a1"}, a2{"a2"}, a3{"a3"}, a4{"a4"};

    constraint c1 { "c1", inside (a1->f, range<bit>({8, 15})) }
    constraint c2 { "c2", a1->f == a4->f }

    activity a {
        a1,
        a2.with {
            inside { a2->f, range<bit>({8,15}) }
        },
        a3.with {
            a3->f == a4->f
        },
        a4;
    }

type_decl<B> B_decl;
```

Example 83 and Example 84 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
            }
        }
    }
}
13.1.4 Set membership expression

The **inside** expression defines the value of the referenced attribute field to be a member of the specified set. Syntax 78 or Syntax 79 shows the syntax for a set membership (**inside**) expression.

### 13.1.4.1 DSL syntax

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

**Syntax 78—DSL: Set membership expression**

### 13.1.4.2 C++ syntax

The corresponding C++ syntax for Syntax 78 is shown in Syntax 79.
// Declare a set membership
class inside : public detail::AlgebExpr {
    public:
        inside ( const attr<int>& a_var,
            const range<int>& a_range
        );
        inside ( const attr<bit>& a_var,
            const range<bit>& a_range
        );
        inside ( const rand_attr<int>& a_var,
            const range<int>& a_range
        );
        inside ( const rand_attr<bit>& a_var,
            const range<bit>& a_range
        );
        template < class T>
        inside ( const rand_attr<T>& a_var,
            const range<T>& a_range
        );
        template < class T>
        inside ( const attr<T>& a_var,
            const range<T>& a_range
        );
    }

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

    Example 85—DSL: inside constraint

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

    Example 86—C++: inside constraint

13.1.4.3 Examples

Example 85 and Example 86 constrain the addr attribute field to the range 0x0..0xFFFF.
13.1.5 Implication constraint

Conditional constraints can be specified using the implication operator (\(\Rightarrow\)). Syntax 80 shows the syntax for an implication constraint.

13.1.5.1 DSL syntax

```
expression_constraint_item ::= expression
    implicand_constraint_item
| ;
implicand_constraint_item ::= -> constraint_set
```

Syntax 80—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 \(a \Rightarrow b\) is \(!a \lor b\). 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.

13.1.5.2 C++ syntax

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

The Boolean equivalent of if_then(a, b) is \(!a \lor b\).

13.1.5.3 Examples

Consider Example 87 and Example 88. 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 \(!((a>5) \lor (b==1))\) is true, so the value of \(a\) is unconstrained. Similarly, if \(b\) has a value other than 1, \(a\) is \(\leq\) 5.

```
struct impl_s {
    rand bit[7:0]    a, b;

    constraint ab_c {
        (a > 5) -> b == 1;
    }
}
```

Example 87—DSL: Implication constraint
13.1.6 if-else constraint

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

Syntax 81 or Syntax 82 shows the syntax for an if-else constraint.

13.1.6.1 DSL syntax

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

Syntax 81—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 13.1.5), if-else style constraints are bidirectional.

13.1.6.2 C++ syntax

The corresponding C++ syntax for Syntax 81 is shown in Syntax 82.
13.1.6.3 Examples

In Example 89 and Example 90, 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 \( \leq 5 \).

In evaluating the constraint, the if-clause evaluates to \(! (a>5) \lor (b==1)\). 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, \ldots 255\}.
13.1.7 foreach constraint

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

Syntax 83 or Syntax 84 shows the syntax for a `foreach` constraint.

13.1.7.1 DSL syntax

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

`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.

13.1.7.2 C++ syntax

The corresponding C++ syntax for Syntax 83 is shown in Syntax 84.
13.1.7.3 Examples

Example 91 and Example 92 show an iterative constraint that ensures that the values of the elements of a fixed-size array increment.
13.1.8 Unique constraint

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

Syntax 85 or Syntax 86 shows the syntax for a **unique** constraint.

### 13.1.8.1 DSL syntax

```%
unique_constraint_item ::= unique { hierarchical_id { , hierarchical_id } } ;
```

**Syntax 85—DSL: unique constraint**

### 13.1.8.2 C++ syntax

The corresponding C++ syntax for Syntax 85 is shown in Syntax 86.

```cpp
// Declare an unique constraint
class unique : public detail::AlgebExpr {
    public:
        // Declare unique constraint
        template < class ... R >
        unique ( const R&& ... /* rand_attr <T> */ r );
};
```

**Syntax 86—C++: unique constraint**
13.1.8.3 Examples

Example 93 and Example 94 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: \((A \neq B) \&\& (A \neq C) \&\& (B \neq C)\).

```
struct my_struct {
    rand bit[0..15] A, B, C;
    constraint unique_abc_c {
        unique {A, B, C};
    }
}
```

**Example 93—DSL: Unique constraint**

```
class my_struct : public structure {
    PSS_CTOR(my_struct, 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};
    );
};
type_decl<my_struct> my_action_decl;
```

**Example 94—C++: Unique constraint**

13.2 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 12.3.4). Similarly, scheduling constraints can be applied to named sub-activities, see Syntax 87.

13.2.1 DSL syntax

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

**Syntax 87—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 12.3.2).

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 12.3.3).

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 12.4). 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 12.4), hold vacuously.

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

13.2.2 Example

Example 95 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.

```plaintext
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 95—DSL: Scheduling constraints

13.3 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.

### 13.3.1 Random attribute fields

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

#### 13.3.1.1 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, random attribute fields are randomized prior to the execution of the activity and, in all cases, prior to the execution of the action’s `exec blocks` (except `pre_solve`, see 13.3.10).

**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.

#### 13.3.1.2 Examples

In **Example 96** and **Example 97**, 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.a` 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.

```c
struct S1 {  
    rand bit[3:0]   a;  
    bit[3:0]        b;  
}  

struct S2 {  
    rand S1         s1_1;  
    S1              s1_2;  
}  

class S1 : public structure {  
    PSS_CTOR(S1,structure);  
    rand_attr<bit> a { "a", width(3,0) };  
    attr<bit> b { "b", width (3,0) };  
};  

type_decl<S1> S1_decl;  
class S2 : public structure {  
    PSS_CTOR(S2,structure);  
    rand_attr<S1> s1_1 {"s1_1"};  
    attr<S1> s1_2 {"s1_2"};  
};  

type_decl<S2> S2_decl;  
```

**Example 96**—DSL: Struct rand and non-rand fields

```c

class S1 : public structure {  
    PSS_CTOR(S1,structure);  
    rand_attr<bit> a { "a", width(3,0) };  
    attr<bit> b { "b", width (3,0) };  
};  

type_decl<S1> S1_decl;  
class S2 : public structure {  
    PSS_CTOR(S2,structure);  
    rand_attr<S1> s1_1 {"s1_1"};  
    attr<S1> s1_2 {"s1_2"};  
};  

type_decl<S2> S2_decl;  
```

**Example 97**—C++: Struct rand and non-rand fields
**Example 98** and **Example 99** show two actions, 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.

```c
class A : public action {
    PSS_CTOR(A, action);
    rand_attr<bit> a {"a", width(3, 0) };  
};
type_decl<A> A_decl;

class B : public action {
    PSS_CTOR(B, 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 
    };
};
type_decl<B> B_decl;
```

**Example 99**—C++: Action rand-qualified fields

**Example 100** and **Example 101** 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.

```c
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 98**—DSL: Action rand-qualified fields
13.3.2 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 17.5), 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 binding rules (see 9.5). 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 102 and Example 103. Assume a scenario is generated starting from action test. Action wr of type writel 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..5, due to a constraint in action write1. But, val of the mem_obj instance rd inputs need to be in the range 8..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 inside 6..10, due to a constraint in write2.
— val is inside 8..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 inside [1..5];
  }

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

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

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

*Example 102—DSL: Randomizing flow object attributes*
13.3.3 Randomization of resource objects

When an action is randomized, its resource-claim fields (of resource type declared with lock / share modifiers, see 10.1) 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 17.5) 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 10.2). 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.

Example 103—C++: Randomizing flow object attributes

class mem_obj : public buffer {
    public:
        PSS_CTOR(mem_obj, buffer);
        attr<int> val ("val");
        constraint c {
            val%2 == 0 // val must be even
        };  
};
type_decl<mem_obj> mem_obj_decl;
class write1 : public action {
    public:
        PSS_CTOR(write1, action);
        output<mem_obj> out_obj ("out_obj");
        constraint c {
            inside(out_obj->val, range<>(1,5) )
        };
};
type_decl<write1> write1_decl;
class write2 : public action {
    public:
        PSS_CTOR(write2, action);
        output<mem_obj> out_obj ("out_obj");
        constraint c {
            inside(out_obj->val, range<>(6,10) )
        };
};
type_decl<write2> write2_decl;
class read : public action {
    public:
        PSS_CTOR(read, action);
        input<mem_obj> in_obj ("in_obj");
        constraint c {
            inside(in_obj->val, range<>(8,12) )
        };
};
type_decl<read> read_decl;
class test : public action {
    PSS_CTOR(test, action);
    activity _activity {
        action_handle<write1>().
        action_handle<read>().
    };
};
type_decl<test> test_decl;
Consider the model in Example 104 and Example 105. 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 104—DSL: Randomizing resource object attributes**
Example 105—C++: Randomizing resource object attributes

```cpp
class top : public component {
    PSS_CTOR(top, component);
    class rsrc_kind_e : public enumeration
        PSS_ENUM(rsrc_kind_e, enumeration, A, B, C, D);
};
type_decl<rsrc_kind_e> rsrc_kind_e_decl;
class rsrc_obj : public resource {
    PSS_CTOR(rsrc_obj, resource);
    rand_attr<rsrc_kind_e> kind {"kind"};
};
type_decl<rsrc_obj> rsrc_obj_decl;
pool<rsrc_obj> rsrc_pool {"rsrc_pool", 2};
bind b1 {rsrc_pool};
class do_something : public action {
    PSS_CTOR(do_something, 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 {
    PSS_CTOR(do_something_else, action);
    lock<rsrc_obj> my_rsrc_inst {"my_rsrc_inst"};
};
type_decl<do_something_else> do_something_else_decl;
class test : public action {
    PSS_CTOR(test, 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;
}
type_decl<top> top_decl;
```

13.3.4 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 11.6). 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 11.7 for more on pool binding.
13.3.5 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 106 and Example 107 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.

```plaintext
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 106—DSL: Activity with random fields*

```plaintext
class A : public action {
    PSS_CTOR(A, action);
    rand_attr<bit> val ("val", width(3,0));
};
type_decl<A> A_decl;
class my_action : public action {
    PSS_CTOR(my_action, 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<my_action> my_action_decl;
```

*Example 107—C++: Activity with random fields*

13.3.6 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 108 and Example 109 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 108—DSL: Activity with random fields in a loop*
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 110 and Example 111.

13.3.7.1 Example 1

Example 110 and Example 111 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.
Example 110—DSL: Struct with random fields

```
struct abc_s {
    rand bit [0..15] a_val, b_val, c_val;

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

Example 111—C++: Struct with random fields

```
class abc_s : public structure {
    PSS_CTOR(abc_s,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
    };
    type_decl<abc_s> abc_s_decl;
}
```

13.3.7.2 Example 2

Example 112 and Example 113 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 110 and Example 111 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 inside 0..15, due to its domain.
2) b.val is inside 0..15, due to its domain.
3) c.val is inside 0..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..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 inside 0..15, due to its domain.
3) c.val is inside 0..15, due to its domain.
4) a.val < b.val.
5) b.val < c.val.
13.3.8 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 114 and Example 115 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 \( \text{top.v.val} \) needs to be less than 14 to satisfy the \( \text{top_action.s1.a.val < top_action.s1.b.val} \) constraint.

```plaintext
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 114—DSL: Sub-activity traversal
13.3.9 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 115—C++: Sub-activity traversal

```c++
class top : public component {
    PSS_CTOR(top, component);
    class A : public action {
        PSS_CTOR(A, action);
        rand_attr<bit> val {"val", width(3,0)};
    };
    type_decl<A> A_decl;
    class sub : public action {
        PSS_CTOR(sub, 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 {
        PSS_CTOR(top_action,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 116 and Example 117 show an activity with two dynamic constraints which are mutually exclusive. If the first branch is selected, \( b.\text{val} \leq 5 \) and \( b.\text{val} < a.\text{val} \). If the second branch is selected, \( b.\text{val} \leq 7 \) and \( b.\text{val} > a.\text{val} \). A PSS processing tool needs to select a value for \( a.\text{val} \) such that a legal value for \( b.\text{val} \) also exists (presuming this is possible).

Given the dynamic constraints, legal value ranges for \( a.\text{val} \) are \( 1..15 \) for the first branch and \( 0..6 \) for the second branch.
Example 116—DSL: Activity with dynamic constraints

```
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;
}
```
13.3.10 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 §17.1 for more information on the *exec block* construct.

### 13.3.10.1 Example 1

Example 118 and Example 119 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`
import bit[5:0] get_init_val();
import 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);
  }
}
13.3.10.2 Example 2

Example 120 and Example 121 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 rd;
    }
    exec post_solve { ... }
}
```

Example 120—DSL: post_solve ordering between action and flow-objects
13.3.11 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 122 and Example 123 show an _exec body_ block that assigns to 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 \( y_1 \) and \( y_2 \) conflict with other attribute field values and constraints. Backtracking is not performed.

```python
import bit[3:0] compute_val1(bit[3:0] v);
import 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 122—DSL: exec body block sampling external data**
import_func compute_val1{"compute_val1",
  import_func::result<bit>({width(3,0)},
  {import_func::in<bit>\{"v\}, width(3,0)})
};
import_func compute_val2{"compute_val2",
  import_func::result<bit>({width(3,0)},
  {import_func::in<bit>\{"v\}, width(3,0)})
};
class pss_top : public component {
public:
  PSS_CTOR(pss_top, component);
  class A : public action {
    public:
      PSS_CTOR(A, 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;
    };
  type_decl<pss_top> pss_top_decl;
};

Example 123—C++: exec body block sampling external data
14. 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.

14.1 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 88). An instance of a coverspec is created to apply the coverage goals to specific data items (see 14.2).
14.1.1 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 
  | ignore_constraint 
  | illegal_constraint
coverspec_coverpoint_binspec ::= bins identifier 
  bin_specification 
  | hierarchical_id ;
ignore_constraint ::= ignore expression ;
illegal_constraint ::= illegal expression ;
coverspec_cross ::= ID : cross coverpoint_identifier { , coverpoint_identifier }
  { { coverspec_cross_body_item } } 
  | ;
coverspec_cross_body_item ::= 
  coverspec_option 
  | ignore_constraint 
  | illegal_constraint
```

Syntax 88—DSL: coverspec declaration

The following also apply.

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

14.1.2 Examples

For examples of how to use a coverspec, see 14.2.2.
## 14.2 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 89).

### 14.2.1 DSL syntax

```plaintext
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 89—DSL: coverspec instantiation**

### 14.2.2 Examples

**Example 124** 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.

```plaintext
class 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 124—DSL: coverspec declaration and instantiation**

### 14.3 coverpoint goal

A coverpoint goal specifies a coverage goal on a scalar data item. Named bins (see 14.7) are used to identify key values and value ranges.

**Example 125** shows a coverpoint goal specified on the addr field. bins are used to specify 64 even bins across the range $0x00000000-0x0000FFFF$. 

14.4 Referencing existing bin schemes

Bins and bin schemes (see 14.7) can be defined inside structs and activities. These bins and bin schemes can be referenced from a coverpoint goal.

Example 126 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 rand 1-'hfff.

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

Example 126—DSL: coverpoint goal

14.5 cross goal

A cross goal specifies a coverage goal on two or more coverpoints that encompasses all combinations of the bins (see 14.7) of the two coverpoints.

Example 127 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 127—DSL: cross goal
14.6 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 128 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 inside [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 128—DSL: coverage constraint

14.6.1 Ignore constraint

Ignore constraints 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 expression true are placed in the ignore bucket. Coverpoint identifiers have the type of the target variable that they monitor.

Ignore expressions can be added to coverpoints or crosses. Coverpoint ignore expressions place samples for that coverpoint into an ignore bucket. Any crosses using the coverpoint also result in those samples being placed in an ignore bucket. Ignore in a cross places the relevant samples to the cross in the crosses ignore bucket and does not change the ignore buckets of the other crosses.

Example 1

```
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 {
    ignore burst_type ? (burst_len < 2) : 1;
}

The following samples are placed in the ignore bucket.

<table>
<thead>
<tr>
<th>burst_type</th>
<th>burst_len</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Example 2

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;
}
}

The following samples are placed in the ignore bucket.

<table>
<thead>
<tr>
<th>burst_type</th>
<th>burst_len</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

14.6.2 Illegal constraint

Illegal constraints bucket coverage samples into an illegal bucket. An illegal constraint is 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 expression true are placed in the illegal bucket. Coverpoint identifiers have the type of the target variable that they monitor.

Illegal expressions can be added to coverpoints or crosses. Coverpoint illegal expressions place samples for that coverpoint into an illegal bucket. Any crosses using the coverpoint also result in those samples being placed in an illegal bucket. Illegal in a cross will place the relevant samples to the cross in the crosses illegal bucket and does not change the illegal buckets of the other crosses.
Example 1

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 {
        illegal !burst_type ? (burst_len > 2) : 1;
    }
}

The following samples are placed in the illegal bucket.

<table>
<thead>
<tr>
<th>burst_type</th>
<th>burst_len</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Example 2

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;
    }
}

The following samples are placed in the illegal bucket.

<table>
<thead>
<tr>
<th>burst_type</th>
<th>burst_len</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

14.7 coverspec bins

The bins construct provides a way to declare a named set of values and value ranges associated with a variable (see Syntax 90).
### 14.7.1 DSL syntax

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td><strong>bins_declaration ::=</strong> <code>bins</code> <code>identifier</code> <code>[variable_identifier]</code> <code>bin_specification</code> <code>;</code></td>
</tr>
<tr>
<td>10</td>
<td><code>bin_specification ::=</code> <code>bin_specifier</code> <code>{ bin_specifier }</code> <code>[ bin_wildcard ]</code></td>
</tr>
<tr>
<td></td>
<td><code>bin_specifier ::=</code> <code>explicit_bin_value</code></td>
</tr>
<tr>
<td></td>
<td>`</td>
</tr>
<tr>
<td></td>
<td>`</td>
</tr>
<tr>
<td></td>
<td>`</td>
</tr>
<tr>
<td></td>
<td><code>explicit_bin_value ::=</code> <code>[ constant ]</code></td>
</tr>
<tr>
<td></td>
<td><code>explicit_bin_range ::=</code> <code>[ constant .. constant ]</code></td>
</tr>
<tr>
<td></td>
<td><code>bin_range_divide ::=</code> <code>explicit_bin_range</code> <code>/</code> <code>constant</code></td>
</tr>
<tr>
<td></td>
<td><code>bin_range_size ::=</code> <code>explicit_bin_range</code> <code>:</code> <code>constant</code></td>
</tr>
<tr>
<td></td>
<td><code>bin_wildcard ::=</code> <code>[ * ]</code></td>
</tr>
</tbody>
</table>

**Syntax 90—DSL: bins declaration**

### 14.7.2 Examples

**Example 129** 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.

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

**Example 129—DSL: bins declaration**

### 14.7.3 Explicit value and range grouping

**Example 130** 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.

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

**Example 130—DSL: Explicit value and range grouping**

### 14.7.4 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 131** 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.

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

**Example 131**—DSL: Defining bins with the divide operator

### 14.7.5 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 132** 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.

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

**Example 132**—DSL: Defining bins with the size operator

### 14.7.6 Wildcard bin (`*`)

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

**Example 133** 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.

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

**Example 133**—DSL: Using the wildcard bin
15. 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 16).

15.1 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 91 or Syntax 92). It does not reiterate modifiers of the type declaration, such as the object kind or base type. See also 16.1.

15.1.1 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 91—DSL: type extension

15.1.2 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 91 is shown in Syntax 92.
15.1.3 Examples

Examples of type extension are shown in Example 134 and Example 135.
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 134—DSL: Type extension
15.1.4 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 136 and Example 137, 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.

Example 136—DSL: Action type extension

```dls
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 inside [SYS_MEM, A_MEM, DDR];
      dst_buff.mem_block inside [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; };
      }
    }
  }
}
```
class mem_ops_c : public component {
  public:
    PSS_CTOR(mem_ops_c, component);
    struct mem_block_tag_e : public enumeration {
      PSS_ENUM(mem_block_tag_e, enumeration, SYS_MEM, A_MEM, B_MEM, DDR);
    };
    type_decl<mem_block_tag_e> mem_block_tag_e_decl;
    struct mem_buff_s : public buffer {
      PSS_CTOR(mem_buff_s, buffer);
      rand_attr<mem_block_tag_e> mem_block {"mem_block"};
    };
    type_decl<mem_buff_s> mem_buff_s_decl;
    class memcpy : public action {
      public:
        PSS_CTOR(memcpy, action);
        input<mem_buff_s> src_buff {"src_buff"};
        output<mem_buff_s> dst_buff {"dst_buff"};
      };
      type_decl<memcpy> memcpy_decl;
    };
    type_decl<mem_ops_c> mem_ops_c_decl;
  };
}

class soc_config_pkg : public package {
  public:
    PSS_CTOR(soc_config_pkg, package);
    class memcpy_ext : public mem_ops_c::memcpy {
      public:
        PSS_CTOR(memcpy_ext, mem_ops_c::memcpy);
        using mem_block_tag_e = mem_ops_c::mem_block_tag_e;
        // introducing new attribute
        rand_attr<int> ta_width {"ta_width", range<>(1)(2)(4)(8)};
        constraint c { // layering additional constraint
          inside { src_buff->mem_block,
            range<mem_block_tag_e>(mem_block_tag_e::SYS_MEM)
          },
          inside { dst_buff->mem_block,
            range<mem_block_tag_e>(mem_block_tag_e::SYS_MEM)
          },
          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 {
  public:
    PSS_CTOR(pss_top, component);
    comp_inst<mem_ops_c> mem_ops {"mem_ops"};
    class test : public action {
      public:
        PSS_CTOR(test, action);
        action_handle<soc_config_pkg::memcpy_ext> cpyl {"cpy1"},
          cpyp2 {"cpy2"};
        constraint c { cpyl->ta_width == cpyp2->ta_width };
        activity a {
          repeat { 3,
            parallel { cpyl, cpyp2 };
          };
        };
        type_decl<test> test_decl;
    };
    type_decl<pss_top> pss_top_decl;
    Example 137—C++: Action type extension

15.1.5 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 138 and Example 139, 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.

```dls
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};
}

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 138—DSL: Enum type extensions
class mem_defs_pkg : public package { // reusable definitions
  public:
    PSS_CTOR(mem_defs_pkg, package);
  class mem_block_tag_e : public enumeration {
    public:
      PSS_ENUM(mem_block_tag_e, enumeration); // initially empty }
    type_decl<mem_block_tag_e> mem_block_tag_e_decl;
  class mem_buff_s : public buffer {
    public:
      PSS_CTOR(mem_buff_s, buffer);
    rand_attr<mem_block_tag_e> mem_block {"mem_block"};
    type_decl<mem_buff_s> mem_buff_s_decl;
  } type_decl<mem_defs_pkg> mem_defs_pkg_decl;
  class dma_c : public component {
    public:
      PSS_CTOR(dma_c, component);
    class mem2mem_xfer : public action {
      public:
        PSS_CTOR(mem2mem_xfer, action);
        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;
    } type_decl<dma_c> dma_c_decl;
    class AB_subsystem_pkg : public package {
      public:
        PSS_CTOR(AB_subsystem_pkg, package);
    class mem_block_tag_e_ext : public mem_defs_pkg::mem_block_tag_e {
      public:
        PSS_ENUM(mem_block_tag_e_ext, mem_defs_pkg::mem_block_tag_e, A_MEM, B_MEM);
        extend_enum<mem_defs_pkg::mem_block_tag_e, mem_block_tag_e_ext> mem_block_tag_e_ext;
    } type_decl<AB_subsystem_pkg> AB_subsystem_pkg_decl;
    class soc_config_pkg : public package {
      public:
        PSS_CTOR(soc_config_pkg, package);
    class mem_block_tag_e_ext : public mem_defs_pkg::mem_block_tag_e {
      public:
        PSS_ENUM(mem_block_tag_e_ext, mem_defs_pkg::mem_block_tag_e, SYS_MEM, DDR);
        extend_enum<mem_defs_pkg::mem_block_tag_e, mem_block_tag_e_ext> mem_block_tag_e_ext_decl;
    } type_decl<soc_config_pkg> soc_config_pkg_decl;
    class dma_c_ext : public dma_c { public:
      PSS_CTOR(dma_c_ext, dma_c);
    class dma_test : public action {
      public:
        PSS_CTOR(dma_test, 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_extDecl;
  } extend_component<dma_c, dma_c_ext> dma_c_extDecl;
} Example 139—C++: Enum type extensions
15.1.6 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.

15.2 Overriding types

The override block (see Syntax 93 or Syntax 94) 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.

15.2.1 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 93—DSL: override declaration

15.2.2 C++ syntax

The corresponding C++ syntax for Syntax 93 is shown in Syntax 94.
15.2.3 Examples

Example 140 and Example 141 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.
Example 140—DSL: Type overrides

```plaintext
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;
    }
}
```
class axi_write_action : public action
{
    PSS_CTOR(axi_write_action, action); }

type_decl<axi_write_action> axi_write_action_decl;
class xlator_action : public action {
public:
    PSS_CTOR(xlator_action, 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
    };
};
type_decl<xlator_action> xlator_action_decl;
class axi_write_action_x : public axi_write_action
{
    PSS_CTOR(axi_write_action_x, axi_write_action); /*...*/
} type_decl<axi_write_action_x> axi_write_action_x_decl;
class axi_write_action_x2 : public axi_write_action_x
{
    PSS_CTOR(axi_write_action_x2, axi_write_action_x); /*...*/
} type_decl<axi_write_action_x2> axi_write_action_x2_decl;
class axi_write_action_x3 : public axi_write_action_x
{
    PSS_CTOR(axi_write_action_x3, axi_write_action_x); /*...*/
} type_decl<axi_write_action_x3> axi_write_action_x3_decl;
class reg2axi_top : public action {
public:
    PSS_CTOR(reg2axi_top, 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
        }
    };
};
type_decl<reg2axi_top> reg2axi_top_decl;
class reg2axi_top_x : public reg2axi_top {
public:
    PSS_CTOR(reg2axi_top_x, reg2axi_top);
    override_instance<axi_write_action_x3>
        _override_inst_2{xlator->axi_action};
};
type_decl<reg2axi_top_x> reg2axi_top_x_decl;

Example 141—C++: Type overrides
16. 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 *packages* using the *import* statement (see Syntax 95 or Syntax 96). From a namespace point of view, *packages* and *components* have the same meaning and use (see also 11.4). Note that both *components* and *packages* are top-level scopes and cannot be further enclosed in other *components* and *packages*. However, in contrast to *components*, *packages* 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.

16.1 Package declaration

Type declarations and type extensions (of *actions*, *structs*, and *enumerated* types) are associated with exactly one package. This association is explicitly expressed by enclosing these definitions in a *package* statement (see Syntax 95 or Syntax 96), either directly or indirectly when they occur in the lexical scope of a *component* definition.
16.1.1 DSL syntax

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

Syntax 95—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.

16.1.2 C++ syntax

The corresponding C++ syntax for Syntax 95 is shown in Syntax 96.

```
/// Declare a PSS package
class package : public detail::PackageBase {
protected:
    /// constructor
    package (const scope& s);
    ~package();
};
```

Syntax 96—C++: package declaration

16.1.3 Examples

For examples of package usage, see 17.2.7.
16.2 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 imported by the context package.

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

16.3 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 16.1.

16.4 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 15.1.
17. 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 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*.

17.1 exec blocks

*exec blocks* provide a mechanism for declaring specific functionality associated with a *component* or *action* (see *Syntax 97* or *Syntax 98*). As discussed in 11.5, *init exec blocks* 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 blocks* specify the actual runtime implementation of atomic actions.
- *pre_solve* and *post_solve* *exec blocks* of *actions* and *structs* 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.

17.1.1 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 97—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.

### 17.1.2 C++ syntax

The corresponding C++ syntax for Syntax 97 is shown in Syntax 98.
/// 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,
      const std::initializer_list<detail::AttrCommon>& write_vars
    );
    /// Declare target template exec
    exec(
      ExecKind kind,
      const std::string& language_or_file,
      const 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
    );
    /// Declare generative target-template exec
    exec(
      ExecKind kind,
      const std::string& language_or_file,
      std::function<void(std::ostream& code_stream)> genfunc
    );
};
17.1.3 Examples

For examples of exec block usage, see 17.5.

17.2 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 17.2.1). The PI can also be used to reference external foreign-language classes via import classes (see 17.7).

The PI consists of two layers: the PSS layer and a foreign language 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.

17.2.1 Import 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 99 or Syntax 100.

17.2.2 DSL syntax

```
import_method_decl ::= import 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 99—DSL: PI method declaration

17.2.3 C++ syntax

The corresponding C++ syntax for Syntax 99 is shown in Syntax 100.
class import_func {
public:
    /// Declare import function input
    template <class T> class in : public detail::ImportFuncParam {
        public:
    };  
    /// Declare import function output
    template <class T> class out : public detail::ImportFuncParam {
        public:
    };  
    /// Declare import function inout
    template <class T> class inout : public detail::ImportFuncParam {
        public:
    };  
    /// Declare import function result
    template <class T> class result : public detail::ImportFuncResult {
        public:
    };  
    /// Declare import function with no result
    import_func(
        const scope &name,
        const std::initializer_list<detail::ImportFuncParam> &params
    );
    /// Declare import function with result
    import_func(
        const scope &name,
        const detail::ImportFuncResult &result,
        const std::initializer_list<detail::ImportFuncParam> &params
    );
    /// Call an import function
    template <class... T> detail::AlgebExpr operator() (  
        const T&... /* detail::AlgebExpr */ params);
};

Syntax 100—C++: PI method declaration

17.2.4 Examples

For examples of using import functions, see 17.2.7.
17.2.5 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 $\leq 64$ bits.

17.2.6 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 $\leq 64$ bits.
- `struct`
- `array`

17.2.7 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 142 and Example 143 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.

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

Example 142—DSL: PI method
17.3 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.

17.4 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 101 or Syntax 102). In typical use, qualifiers are specified in an environment-specific package (e.g., a UVM environment-specific package or C-Test-specific package).

17.4.1 DSL syntax

```
import_method_phase_qualifiers ::= import import_function_qualifiers type_identifier ;
import_function_qualifiers ::= 
    method_qualifiers [ language_identifier ]
| language_identifier
method_qualifiers ::= 
    target 
| solve
```

_Syntax 101—DSL: PI function qualifiers_

17.4.2 C++ syntax

The corresponding C++ syntax for Syntax 101 is shown in Syntax 102.

```
class generic_methods : public package { 
    PSS_CTOR(generic_methods,package);

    import_func compute_value { "compute_value",
        import_func::result<int>() ,
        {
            import_func::in<int>("val"),
            import_func::out<int>("out_val")
        }
    };
};
type_decl<generic_methods> generic_methods_decl;
```
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 144 and Example 145 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.
17.4.4 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 useensible 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 146 and Example 147 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.
17.5 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 17.1).

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. Components’ init blocks are called before the scenarios 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 148 and Example 149 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.

```verilog
package external_functions_pkg {
  import bit[31:0] alloc_addr(bit[31:0] size);

  import 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 inside [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 148—DSL: Calling PI functions**
17.6 Target-template implementation for import functions

By default, import 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 import functions can be provided by target-template code strings.

The target-template form of PI import functions (see Syntax 103 or Syntax 104) allow non-functional languages, such as assembly, to be targeted in an efficient manner. The target-template form of PI import functions are always target implementations. Variable references may only be used in expression positions. Function return values shall not be provided, i.e., only import functions that return void are supported.
17.6.1 DSL syntax

import_method_qualifiers ::=  
  import_method_phase_qualifiers  
  | import_method_target_template
import_method_target_template ::= import language_identifier method_prototype = string ;

Syntax 103—DSL: Target-template import implementation

17.6.2 C++ syntax

The corresponding C++ syntax for Syntax 103 is shown in Syntax 104.

    /// Declare an import function
    class import_func {
    public:
        /// Declare target-template import function with no result
        import_func(
          const scope &name,
          const std::string &language,
          const std::initializer_list<detail::ImportFuncParam> &params,
          const std::string &target_template
        );
        /// Declare target-template import function with result
        import_func(
          const scope &name,
          const std::string &language,
          const detail::ImportFuncResult &result,
          const std::initializer_list<detail::ImportFuncParam> &params,
          const std::string &target_template
        );
    };

Syntax 104—C++: Target-template import implementation

17.6.3 Examples

Example 150 and Example 151 provide an assembly-language target-template code block implementation for the do_stw import function. Function parameters are referenced using mustache notation ({{variable}}).
17.7 Import classes

In addition to interfacing with external foreign-language functions, the PSS description can interface with foreign-language classes. See also Syntax 105 or Syntax 106.

17.7.1 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 105—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.

### 17.7.2 C++ syntax

The corresponding C++ syntax for **Syntax 105** is shown in **Syntax 106**.

```cpp
// Declare an import class
class import_class : public detail::ImportClassBase {
  public:
    // Constructor
    import_class(const scope &name);
    // Destructor
    ~import_class();
};

Syntax 106—C++: Import class declaration
```

### 17.7.3 Examples

**Example 152** and **Example 153** 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**.

```cpp
import class base {
  void base_method();
}

import class ext : base {
  void ext_method();
}

Example 152—DSL: Import class
```
### 17.8 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.

#### 17.8.1 Target-template code exec block kinds

There are several kinds of target template code *exec blocks*.

1. **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 blocks* of actions with the same set of scheduling dependencies logically takes place at the same time. Implementation of the standard should guarantee that *exec blocks* 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 *packages*, given that these packages are not used for the very same test.

2. **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.

3. **declaration** - specifies declarative statements used to define entities that are used by subsequent code blocks. Examples are the definition of global variables or functions.

4. **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.

5. **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.

```cpp
class base : public import_class {
    public:
        PSS_CTOR(base, import_class);
        import_func base_method { "base_method", {} };
    }
    type_decl<base> base_decl;
}
class ext : public base {
    public:
        PSS_CTOR(ext, base);
        import_func ext_method { "ext_method", {} };
    }
    type_decl<ext> ext_decl;
```
17.8.2 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.

17.8.3 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 17.1) 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.

17.9 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 107) 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.

17.9.1 C++ syntax

```cpp
/// Declare an exec block
class exec : public detail::ExecBase {
  public:
    /// Declare in-line exec
    exec(
      ExecKind kind,
      const std::initializer_list<detail::AttrCommon>& write_vars
    )
  );
};
```

Syntax 107—C++: in-line exec block declaration
/// Declare an action
class action : public detail::ActionBase {
  protected:
  /// Constructor
  action ( const scope& s );
  /// Destructor
  ~action();
  public:
  /// In-line exec block
  virtual void pre_solve();
  /// In-line exec block
  virtual void post_solve();
}; // class action

// Syntax 108—C++: in-line action declaration

/// Declare a structure
class structure : public detail::StructureBase {
  public:
  /// In-line exec block
  virtual void pre_solve();
  /// In-line exec block
  virtual void post_solve();
};

// Syntax 109—C++: in-line structure declaration

/// Declare a buffer object
class buffer : public detail::BufferBase {
  public:
  /// In-line exec block
  virtual void pre_solve();
  /// In-line exec block
  virtual void post_solve();
};

// Syntax 110—C++: in-line buffer object declaration
/// Declare a stream object
class stream : public detail::StreamBase {
public:
    /// In-line exec block
    virtual void pre_solve();
    /// In-line exec block
    virtual void post_solve();
};

Syntax 111—C++: in-line stream object declaration

/// Declare a state object
class stream : public detail::StateBase {
public:
    /// In-line exec block
    virtual void pre_solve();
    /// In-line exec block
    virtual void post_solve();
};

Syntax 112—C++: in-line state object declaration

/// Declare a resource object
class resource : public detail::ResourceBase {
public:
    /// In-line exec block
    virtual void pre_solve();
    /// In-line exec block
    virtual void post_solve();
};

Syntax 113—C++: in-line resource object declaration

/// Declare a component
class component : public detail::ComponentBase {
public:
    /// In-line exec block
    virtual void init();
};

Syntax 114—C++: in-line component declaration
17.9.2 Examples

Example 154 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 {
    PSS_CTOR(mem_buf,buffer);
    attr<int> mode ("mode");
    attr<int> size ("size");
};
type_decl<mem_buf> mem_buf_decl;
class decode_mem : public action {
    PSS_CTOR(decode_mem,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());
    }
};
type_decl<decode_mem> decode_mem_decl;
```

Example 154—C++: in-line exec

17.10 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 online-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.
17.10.1 Generative PI execs

Target PI execs (body, run_start, and run_end) can be specified in generative mode (see Syntax 115). 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.

17.10.1.1 C++ syntax

```cpp
// Declare an exec block
class exec : public detail::ExecBase {
public:
    /// Declare generative procedural-interface exec
exec(
        ExecKind kind,
        std::function<void()> genfunc // shadowed by variadic template c'tor
            // handle at construction time
    );
};
```

Syntax 115—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.

17.10.1.2 Examples

Example 155 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, an import function call occurs in the context of a native C++ loop, thereby generating a sequence of the respective calls as the loop unrolls.
Example 155—C++: generative PI exec

class mem_ops_pkg : public package {
  PSS_CTOR(mem_ops_pkg, package);
  import_func alloc_mem { "alloc_mem",
    import_func::result<bit>(width(63,0)),
    { import_func::in<int>("size") });
  import_func write_word { "write_word",
    { import_func::in<bit>("addr", width(63,0)) },
    import_func::in<bit>("data", width(31,0)) });
};
type_decl<mem_ops_pkg> mem_ops_pkg_decl;
class my_comp : public component {
  PSS_CTOR(my_comp, component);
  class write_multi_words : public action {
    PSS_CTOR(write_multi_words, action);
    rand_attr<int> num_of_words { "num_of_words", range<>(2,8) };
    attr<bit> base_addr { "base_addr", width(63,0) };
    // exec specification in generative mode
    exec body { exec::body, [&](){ // capturing action variables
      base_addr = mem_ops_pkg_decl->alloc_mem(num_of_words*4);
      // in pre-gen unroll the loop,
      // evaluating num_of_words on solve platform
      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;
};
type_decl<my_comp> my_comp_decl;

Example 156—C++: Possible code generated for write_multi_words()
17.10.2.1 C++ syntax

```cpp
class exec : public detail::ExecBase {
public:
    /// Declare generative target-template exec
    exec(
        ExecKind kind,
        const std::string& language_or_file,
        std::function<void(std::ostream& code_stream)> genfunc
            // shadowed by variadic template c'tor
            // handle at construction time
    );
};
```

Syntax 116—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 17.6). 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.

17.10.2.2 Examples

Example 157 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.
Example 157—C++: generative target-template exec

The possible code generated for write_multi_words() is shown in Example 156.

17.11 Comparison between mapping mechanisms

Previous sections describe three mechanisms for mapping PSS entities to external (on PSS) definitions: functions that directly map to foreign API (see 17.2), functions that map to foreign language procedural code using target code templates (see 17.6), and exec blocks where arbitrary target code templates are in-lined (see ). 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

<table>
<thead>
<tr>
<th></th>
<th>No target code generation</th>
<th>Per-model target code generation</th>
<th>Per-test target code generation</th>
<th>Non-procedural binding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct-mapped</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>functions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Target-template</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>functions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Target-template</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>exec-blocks</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

Table 5—Exec block kinds supported for mapping mechanisms

<table>
<thead>
<tr>
<th></th>
<th>Action runtime behavior exec blocks body</th>
<th>Non-procedural exec blocks header, declaration, file</th>
<th>Global test exec blocks run_start, run_end</th>
<th>Solve exec blocks pre_solve, post_solve</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct-mapped</td>
<td>X</td>
<td>X (only in pre-generation)</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>functions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Target-template</td>
<td>X</td>
<td>X (only in pre-generation)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>functions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Target-template</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>exec-blocks</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The possible use of action and struct attributes differs between mapping constructs. Explicitly declared signatures of import functions 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.
17.12 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 117 or Syntax 118.

17.12.1 DSL syntax

<table>
<thead>
<tr>
<th>Back assignment to PSS attributes</th>
<th>Passing user-defined and compound data-types</th>
<th>Using PSS attributes in non-expression positions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct-mapped functions</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Target-template functions</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Target-template exec-blocks</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 17.2.1), 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.

17.12.2 C++ syntax

The corresponding C++ syntax for Syntax 117 is shown in Syntax 118.
17.12.3 Examples

Example 158 and Example 159 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.
17.12.4 Export action foreign language binding

An exported action is exposed as a method in the target foreign language (see Example 160). 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);
}
```

NOTE—Foreign language binding is same for DSL and C++.
18. 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

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 
  | import_method_decl 
  | import_class_decl 
  | import_method_qualifiers 
  | export_action 
  | typedef_declaration 
  | bins_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

actionDeclaration ::= 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
   | bins_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

eexec_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
| bins_declaration
| coverspec_declaration
| exec_block_stmt

struct_field_declaration ::= [ struct_field_modifier ] data_declaration

struct_field_modifier ::= rand

B.4 Procedural interface (PI)

import_method_decl ::= import 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

import_method_qualifiers ::= import_method_phase_qualifiers
| import_method_target_template
import_method_phase_qualifiers ::= import import_function_qualifiers
type_identifier;

import_function_qualifiers ::= method_qualifiers [ language_identifier ]
| language_identifier

method_qualifiers ::=
target
| solve

import_method_target_template ::= import language_identifier method_prototype
= string;

method_parameter_list ::= ( [ expression { , expression } ] )

B.4.1 Import class declaration

import_classDecl ::= import class import_class_identifier
[ import_class_extends ] { { import_class_methodDecl } } [ ; ]

import_class_extends ::= : type_identifier [ , type_identifier ]

import_class_methodDecl ::= 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

component_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
| 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 } }
| 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 | , open_range_value { , open_range_value } |
   .. expression { , open_range_value } ] ] ]

integer_atom_type ::= 
   int | bit

open_range_value ::= expression [ .. expression ]

open_range_list ::= open_range_value { , open_range_value }

string_type ::= string

bool_type ::= bool

user_defined_datatype ::= type_identifier

action_type ::= type_identifier

struct_type ::= type_identifier

enum_type ::= type_identifier

enum_declaration ::= enum enum_identifier { [ enum_item { , enum_item } ] } [ ; ]

enum_item ::= identifier [ = constant_expression ]

typedef_type ::= type_identifier

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 
  ;

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 } }[ ; ]

coverspec_coverpoint_body_item ::= coverspec_option
| coverspec_coverpoint_binspec
| ignore_constraint
| illegal_constraint

coverspec_coverpoint_binspec ::= bins identifier
  bin_specification
| hierarchical_id ;

ignore_constraint ::= ignore expression ;

illegal_constraint ::= illegal expression ;

coverspec_cross ::= ID : cross coverpoint_identifier , coverpoint_identifier 
  { { coverspec_cross_body_item } }[ ; ]

coverspec_cross_body_item ::= coverspec_option
| ignore_constraint
| illegal_constraint

Bins

bins_declaration ::= bins identifier [ variable_identifier ] bin_specification
  ;

bin_specification ::= bin_specifier { bin_specifier } [ bin_wildcard ]

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 { < | <= | > | >= binary_shift_expr
    | inside | 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
    | method_function_call

paren_expr ::= ( expression )

variable_ref ::= hierarchical_id [ [ 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

customant ::=  
    number  
    | identifier

identifier ::=  
    ID  
    | ESCAPED_ID

hierarchical_id ::= identifier { . identifier }

action_type_identifier ::= type_identifier

type_identifier ::= ID { :: ID }

hierarchical_type_identifier ::= ID :: 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

pool_identifier ::= identifier

variable_identifier ::= identifier

bin_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_oct_number 
   | based_bin_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}

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.

C.1 File pss.h

```cpp
#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/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/inside.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/import_func.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_attr.h

```cpp
#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);
    };
}

#include "pss/timpl/action_attr.t"

C.3 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(const 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:
                        // Constructor
                        //template <class... R>
                        // activity(R&&... /* detail::ActivityStmt */ r);
                        // Constructor
                        // activity(const std::vector<detail::ActivityStmt*>& stmts);
                        // Destructor
                        ~activity();
                };
            }; // select() must be inside action declaration to disambiguate from
        };
    };
} // namespace pss
template < class... R >
select(R&&... /* detail::ActivityStmt */ r);
select(const 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(const 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(const 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(const 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 repeat while statement
    do_while(const detail::ActivityStmt& activity,
    const detail::AlgebExpr& cond
    );
}; // class action
}; // namespace pss
#include "pss/timpl/action.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* ()
/// enum 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);
};
    /// 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);

   /// 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);

   /// 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;
}
// 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 resource to multiple targets
            template <class R /*resource*/, 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();
```
C.9 File pss/chandle.h

```cpp
#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 );
    };
}; // namespace pss
```

C.10 File pss/comp_inst.h

```cpp
#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-> () const;
            /// Access content
            T& operator* () const;
    };
    /// 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
```

C.11 File pss/component.h

```cpp
#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
    }
    class constraint_block : public detail::AlgebExpr {
        public:
            template <class... R> constraint_block(
                const R&... /*detail::AlgebExpr*/ constraints);
    };
    /// 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
C.13 File pss/enumeration.h

```cpp
#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);
                };
            };
    };
#define PSS_ENUM(class_name, base_class, ...) 
    public: 
    class_name (const scope& p) : base_class (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; 
    }
#include "pss/timpl/enumeration.t"
```

C.14 File pss/exec.h

```cpp
#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,
    const std::initializer_list<detail::AttrCommon>& write_vars
);

/// Declare target template exec
exec(
    ExecKind kind,
    const std::string& language_or_file,
    const 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,
    const std::string& language_or_file,
    std::function<void(std::ostream& code_stream)> 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();
    };
}

Copyright © 2017 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.
\}; // namespace pss
#include "pss/timpl/extend.t"

C.17 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.18 File pss/import_func.h

#pragma once
#include "pss/scope.h"
#include "pss/bit.h"
#include "pss/width.h"
#include "pss/range.h"
#include "pss/detail/execStmt.h"
#include "pss/detail/importFuncParam.h"
#include "pss/detail/importFuncResult.h"
namespace pss {
    /// Declare an import function
    class import_func {
    public:
        /// Declare import function input
        template <class T> class in : public detail::ImportFuncParam {
        public:
            
        };
        /// Declare import function output
        template <class T> class out : public detail::ImportFuncParam {
        public:
            
        };
        /// Declare import function inout
        template <class T> class inout : public detail::ImportFuncParam {
        public:
            
        };
        /// Declare import function result
        template <class T> class result : public detail::ImportFuncResult {
        public:
            
        };
        /// Declare import function with no result
        import_func(
            const scope &name,
            const std::initializer_list <detail::ImportFuncParam> &params
        );
        /// Declare import function with result
import_func(
    const scope &name,
    const detail::ImportFuncResult &result,
    const std::initializer_list<detail::ImportFuncParam> &params
);

/// Call an import function
template <class... T> detail::AlgebExpr operator() (const T&... /* detail::AlgebExpr */ params);

/// Import function availability
enum kind { solve, target };

/// 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 target-template import function with no result
import_func(
    const scope &name,
    const std::string &language,
    const std::initializer_list<detail::ImportFuncParam> &params,
    const std::string &target_template
);

/// Declare target-template import function with result
import_func(
    const scope &name,
    const std::string &language,
    const detail::ImportFuncResult &result,
    const std::initializer_list<detail::ImportFuncParam> &params,
    const std::string &target_template
);

/// Template specialization for inputs
template <> class import_func::in<bit> : public detail::ImportFuncParam {
    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 import_func::in<int> : public detail::ImportFuncParam {
    public:
        in(const scope &name);
        in(const scope &name, const width &w);
        in(const scope &name, const width &w, const range<int> &rng);
    }

/// Template specialization for outputs
template <> class import_func::out<bit> : public detail::ImportFuncParam {
    public:
        out(const scope &name);
        out(const scope &name, const width &w);
        out(const scope &name, const width &w, const range<bit> &rng);
    }

template <> class import_func::out<int> : public detail::ImportFuncParam {
    public:
        out(const scope &name);
out(const scope &name, const width &w);
out(const scope &name, const width &w, const range<int> &rng);
};
// Template specialization for inouts
template <> class import_func::inout<bit> : public detail::ImportFuncParam {
public:
inout(const scope &name);
inout(const scope &name, const width &w);
inout(const scope &name, const width &w, const range<bit> &rng);
};
template <> class import_func::inout<int> : public detail::ImportFuncParam {
public:
inout(const scope &name);
inout(const scope &name, const width &w);
inout(const scope &name, const width &w, const range<int> &rng);
};
// Template specialization for results
template <> class import_func::result<bit> : public detail::ImportFuncResult {
public:
result();
result(const width &w);
result(const width &w, const range<bit> &rng);
};
template <> class import_func::result<int> : public detail::ImportFuncResult {
public:
result();
result(const width &w);
result(const width &w, const range<int> &rng);
};
}; // namespace pss

C.19 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.20 File pss/inside.h

```cpp
#pragma once
#include "pss/range.h"
#include "pss/attr.h"
#include "pss/rand_attr.h"
namespace pss {
  /// Declare a set membership
class inside : public detail::AlgebExpr {
    public:
    inside ( const attr<int>& a_var,
            const range<int>& a_range );
    inside ( const attr<bit>& a_var,
            const range<bit>& a_range );
    inside ( const rand_attr<int>& a_var,
            const range<int>& a_range );
    inside ( const rand_attr<bit>& a_var,
            const range<bit>& a_range );
    template < class T>
    inside ( const rand_attr<T>& a_var,
            const range<T>& a_range );
    template < class T>
    inside ( const attr<T>& a_var,
            const range<T>& a_range );
  }
}; // namespace pss
#include "pss/timpl/inside.t"
```

C.21 File pss/lock.h

```cpp
#pragma once
#include "pss/detail/lockBase.h"
namespace pss {
  /// Claim a locked resource
template<class T>
class lock : public detail::LockBase {
    public:
    lock(const scope& name);
    ~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-> () const;
/// Access content
T& operator* () const;
}; // 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:
            pool (const scope& name, std::size_t count = 1);
        };
}; // 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);
            /// Constructor and initial value
            rand_attr (const scope& name, const T& init_val);
            /// Copy constructor
            rand_attr(const rand_attr<T>& other);
            /// Struct access
            T* operator-> ();
            /// Struct access
            T& operator* ();
            /// enum 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 and initial value
            rand_attr (const scope& name, const int& init_val);
            /// Constructor defining width
            rand_attr (const scope& name, const width& a_width);
            /// Constructor defining width and initial value
            rand_attr (const scope& name, const width& a_width, const int& init_val);
            /// Constructor defining range
            rand_attr (const scope& name, const range<int>& a_range);
            /// Constructor defining range and initial value
            rand_attr (const scope& name, const range<int>& a_range, const int& init_val);
            /// Constructor defining width and range
            rand_attr (const scope& name, const width& a_width, const range<int>& a_range);
            /// Constructor defining width and range and initial value
            rand_attr (const scope& name, const width& a_width, const range<int>& a_range, const int& init_val);
            /// 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 <>
class rand_attr<bit> : public detail::RandAttrBitBase {
public:
    /// Constructor
    rand_attr (const scope& name);
    /// Constructor and initial value
    rand_attr (const scope& name, const bit& init_val);
    /// Constructor defining width
    rand_attr (const scope& name, const width& a_width);
    /// Constructor defining width and initial value
    rand_attr (const scope& name, const width& a_width, const bit& init_val);
    /// Constructor defining range
    rand_attr (const scope& name, const range<bit>& a_range);
    /// Constructor defining range and initial value
    rand_attr (const scope& name, const range<bit>& a_range, const bit& init_val);
    /// Constructor defining width and range
    rand_attr (const scope& name, const width& a_width, const range<bit>& a_range);
    /// Constructor defining width and range and initial value
    rand_attr (const scope& name, const width& a_width, const range<bit>& a_range, const bit& init_val);
    /// 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);
    /// Constructor and initial value
    rand_attr (const scope& name, const std::string& init_val);
    /// 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);
    /// Constructor and initial value
    rand_attr (const scope& name, const bool init_val);
    /// 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);
/// 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);
/// Access to specific element
```cpp
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;
};

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& 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);
};
};

#include "pss/timpl/rand_attr.t"

C.27 File pss/range.h

```
C.29 File pss/scope.h

```cpp
#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) {}

C.30 File pss/share.h

```cpp
#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
```
```cpp
#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 {
  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 {
  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();
    }; // 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 an unique constraint
    class unique : public detail::AlgebExpr {
        public:
            /// Declare unique constraint
            template < class ... R >
            unique ( const 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> ;
}; // namespace pss

C.38 File pss/width.h

#pragma once
#include "pss/detail/widthBase.h"
namespace pss {
    /// 
    /// brief Declare width of a numeric scalar attribute
    /// class width : public detail::WidthBase { 
    /// public:
    /// // brief Declare width as a range of bits
    /// width (const std::size_t& lhs, const std::size_t& rhs);
    /// // brief Declare width in bits
    /// width (const std::size_t& size);
    /// // brief copy constructor
    /// width (const width& a_width);
    ///
    /// namespace pss

C.39 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
    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 inside()

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);

/// 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

#include "algebExpr.t"
#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"
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

<table>
<thead>
<tr>
<th>PSS type</th>
<th>C type Input</th>
<th>C type Output / Inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>const char *</td>
<td>char **</td>
</tr>
<tr>
<td>bool</td>
<td>unsigned int</td>
<td>unsigned int *</td>
</tr>
<tr>
<td>chandle</td>
<td>void *</td>
<td>void **</td>
</tr>
<tr>
<td>bit (1-8-bit domain)</td>
<td>unsigned char</td>
<td>unsigned char *</td>
</tr>
<tr>
<td>bit (9-16-bit domain)</td>
<td>unsigned short</td>
<td>unsigned short *</td>
</tr>
<tr>
<td>bit (17-32-bit domain)</td>
<td>unsigned int</td>
<td>unsigned int *</td>
</tr>
<tr>
<td>bit (33-64-bit domain)</td>
<td>unsigned long long</td>
<td>unsigned long long *</td>
</tr>
<tr>
<td>int (1-8-bit domain)</td>
<td>char</td>
<td>char *</td>
</tr>
<tr>
<td>int (9-16-bit domain)</td>
<td>short</td>
<td>short *</td>
</tr>
<tr>
<td>int (17-32-bit domain)</td>
<td>int</td>
<td>int *</td>
</tr>
<tr>
<td>int (33-64-bit domain)</td>
<td>long long</td>
<td>long long *</td>
</tr>
</tbody>
</table>

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: enumerated 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:

```plaintext
import void foo(derived_s d);
import solve CPP foo;
```

with the corresponding PSS definitions:

```plaintext
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:

```plaintext
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>
<thead>
<tr>
<th>PSS type</th>
<th>SystemVerilog type</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>string</td>
</tr>
<tr>
<td>bool</td>
<td>boolean</td>
</tr>
<tr>
<td>chandle</td>
<td>chandle</td>
</tr>
<tr>
<td>bit (1-8-bit domain)</td>
<td>byte unsigned</td>
</tr>
<tr>
<td>bit (9-16-bit domain)</td>
<td>shortint unsigned</td>
</tr>
<tr>
<td>bit (17-32-bit domain)</td>
<td>int unsigned</td>
</tr>
<tr>
<td>bit (33-64-bit domain)</td>
<td>longint unsigned</td>
</tr>
<tr>
<td>int (1-8-bit domain)</td>
<td>byte</td>
</tr>
<tr>
<td>int (9-16-bit domain)</td>
<td>shortint</td>
</tr>
<tr>
<td>int (17-32-bit domain)</td>
<td>int</td>
</tr>
<tr>
<td>int (33-64-bit domain)</td>
<td>longint</td>
</tr>
</tbody>
</table>

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 initial/root action(s):
   1) Specified by the user.
   2) Implicitly in component pss_top, unless otherwise specified.
      [Not unlike specifying a top-level module in SystemVerilog.]
   3) If the specified root action is a compound action:
      i) Identify the initial action(s) in the action’s activity statement.
      ii) Identify scheduling dependencies among all other actions in the activity.

b) Beginning with the initial action(s), for each action:
   1) For each output object declared in the action:
      i) Identify the object pool of the appropriate type to which the action is bound.
      ii) Identify all other action(s) bound to the same pool that declare a matching input type.
      iii) The constraints for evaluating field(s) of the flow object are the intersection of the constraints in all actions sharing that object and the constraints specified in the object itself.
      iv) Identify scheduling dependencies enforced by the shared objects and add these to the set of dependencies identified in a.3.ii.
         If there is a scheduling conflict, go to c.
   2) For each input object declared in the action:
      i) If the initial action has an input object, go to c.
      ii) If the action is not an initial action, identify the object pool of the appropriate type to which the action is bound.
      iii) Identify all other action(s) bound to the same pool that declare a matching output type.
      iv) The constraints for evaluating field(s) of the flow object are the intersection of the constraints in all actions sharing that object and the constraints specified in the object itself.
      v) Identify scheduling dependencies enforced by the shared objects and add these to the set of dependencies identified in a.3.ii.
         If there is a scheduling conflict, go to c.
   3) Once all field constraints for each object have been determined (including chaining across actions, e.g., src.foo == dest.bar or src.foo < dest.bar):
      i) If the constraint set is null, an error shall be generated.
      ii) Choose a random value for each field of each object.
   4) For each resource locked or shared (i.e., claimed) by the action:
      i) Identify the resource pool of the appropriate type to which the action is bound.
      ii) Identify all other action(s) bound to the same pool that claim a resource of the same type.
      iii) Each instance in the resource pool has an implicit instance_id field that is unique for that pool.
iv) The constraints for evaluating field(s) of the resource are the intersection of the constraints in all actions claiming that resource and the constraints specified in the resource object itself.

1. If the resulting constraint set is `null`, an error shall be generated.
2. Otherwise, choose a random value for each field that satisfies the constraint set.

NOTE—If multiple actions require the same value for `instance_id`, then those actions shall claim the same instance of the resource.

v) Identify scheduling dependencies enforced by the claimed resource and add these to the set of dependencies identified in a.3.ii.

1. If an action locks a resource, no other action claiming that resource may be scheduled in parallel.
2. If actions scheduled in parallel attempt to lock more resources than are available in the pool, an error shall be generated.
3. If the resource is not locked, there are no scheduling implications of sharing a resource.

c) Inferencing

If the flow object allocation scheduling implications create a conflict with the activity scheduling semantics:

[there are no actions declared in the activity that can legally provide/consume a given flow object that is required/provided by a given action in the activity.]

1) To supply a required input object, the tool needs to infer a new action that outputs an object of the desired type.
   i) The flow object needs to be of the type defined in the pool to which the consuming action is bound.
   ii) The inferred action needs to be bound to the same pool to which the consuming action is bound.
   iii) The inferred action is treated as if it were instantiated in the same component as the consuming action.
   iv) The action may be inferred from the set of actions defined in the same component scope as the consuming action or in any parent component scope.
   v) If the inferred action requires an input, it may be provided by an action already instantiated in the activity that may legally provide it or a new action may be inferred as in c.1.
   vi) If the inferred action produces an output, it may be consumed by an action already instantiated in the activity that may legally consume it or a new action may be inferred as in c.2.

2) If the action outputs a stream object (which requires a consuming action), the tool needs to infer a new action that inputs an object of the desired type.
   i) The flow object needs to be of the type defined in the pool to which the producing action is bound.
   ii) The inferred action needs to be bound to the same pool to which the producing action is bound.
   iii) The inferred action is treated as if it were instantiated in the same component as the producing action.
   iv) The action may be inferred from the set of actions defined in the same component scope as the producing action or in any parent component scope.
   v) If the inferred action requires an input, it may be provided by an action already instantiated in the activity that may legally provide it or a new action may be inferred as in c.1.
   vi) If the inferred action produces an output, it may be consumed by an action already instantiated in the activity that may legally consume it or a new action may be inferred as in c.2.
3) If the inferred action claims a resource object, go to b.4.

4) Inferencing shall continue until a terminating action is inferred:
   i) an action that produces an object of the desired type that does not have any input declarations;
   ii) an action that consumes an object of the desired type that does not have any output declarations of stream type;
   iii) if the tool reaches the maximum inferencing depth, it shall infer a terminating action if one is available.

See also 9.5.
Annex F
(informative)

HSI UART example

This is a sample HSI specification for a UART.

Pc16550_intr.h:

```cpp
// 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:

```cpp
// 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);
    }
};
```

```cpp
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,
    TXREGEIGHT = 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 == TXREGEQEMPT));

    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(&pcl6550::configure_fifo, this,  
"configure_fifo", "API to configure FIFO",  
pss::target_var<int>("enable_fifo");
register_target_function(&pcl6550::start_receive, this, "start_receive",  
"enables the reception of data");
enable_tx_handle = register_target_function(&pcl6550::enable_transmit,  
this, "enable_transmit", "enables the transmission of data");
register_target_function(&pcl6550::configure, this, "configure", "API to  
configure different features of Uart",  
UartConfig("config"));
}

int main(int argc, char *argv[])  
{  
  pcl6550 device("pcl6550");
  device.register_functions();

  return pss::main(argc, argv);
};