Release Notes
For
Portable Test and Stimulus Standard

This document contains release notes for the 2026 revisions of the standard collected and managed by the Accellera Portable Test and Stimulus Standard (PSS) Working Group. The language reference manual can be downloaded here. In addition, it contains a history of deprecations from previous versions of the standard.

Version 3.1

Added and extended features

- **String literals**
  Both quoted string literals and triple-quoted string literals may be used anywhere a string literal is desired or required. See the LRM sections “Lexical conventions” and “Target-template implementation for functions.”

- **New keywords**
  Added the keywords “annotation”, “overlap”, “mutable”, and “soft”. See the LRM “PSS keywords” table for a list of all PSS reserved keywords.

- **Action attribute initialization**
  Added support for inline specification of non-rand action attribute values. See the LRM section “Action attribute initialization.”

- **Target-time communication and synchronization between executors**
  Added support for target-time communication and synchronization between executors to dynamically synchronize or communicate at runtime. See the LRM section “Synchronization and communication” and the “Package sync_pkg” section of the “Core library package”

- **Support for multi-dimensional arrays of action handles**
  Multi-dimensional arrays are now supported for the following:
  - Component instance arrays
  - Action handle arrays
  - Monitor handle arrays
  - Flow object input/output arrays
  - Resource object (lock/share) array

  Refer to the LRM section “Action handle array traversal”

- **Support for "message" on both solve and target platforms**
  PSS now supports the function `message()` to log certain information during the solving and execution of a test in a portable way. See the "Message logging" section of the LRM.

- **Support for implementation- and model-level annotation elements**
  Implementation- and model-level annotations allow metadata to be attached to elements of a PSS model. See the LRM section "Standard annotations."

- **Sharing an address claim between overlapping actions**
  The allocation mode struct captures address sharing relationships with other claims as well as optional directives to memory allocation mechanisms. See the LRM section "Allocation mode."

- **Support for register base types to work with register collections**
  This release supports more flexibility in forming and working with collections of same-size registers by introducing register base types. Refer to the "Registers" section in the "Core Library" clause.

- **Support for exported functions**
  This release supports exported functions. Exported functions allow the environment to call back into the implementation of a PSS model. See the LRM section "Definition using exported functions" in the "Test realization" clause.

- **Support for target execution units**
  Target execution units are components used by PSS tools to group executor realizations (emitted target code) into common files that share the same target language, enabling those realizations to be compiled together using the target platform's compilation tools. This improves usability by providing a common standard for setting up the target environment across different vendor tools. See the LRM section "Target execution units" in the "PSS core library" clause.

- **Soft constraints**
  The constraints described in previous versions of the LRM can be thought of as "hard" constraints. These constraints always apply. If they conflict, the result is a solve contradiction. There are many cases where some default condition must be specified and overridden in specific situations. Default constraints provide a simple approach to doing this when the overriding condition is static in nature. Soft constraints are used when an overriding condition is dynamic (e.g., within constraint conditions). See the LRM section "Soft constraints" in the "Randomization" clause.

- **Symbolic representation of registers in target code**
  Register instances (of type `reg_c<>`) within a register group hierarchy are currently accessed using address handles computed from the `getoffset*` functions and the handle set via `set_handle()`. The resulting generated target code refers to registers by their resolved numeric address values.

  It is sometimes beneficial to refer to a register by a symbolic name rather than a numeric address. This can facilitate compile-time type checking, improve readability of generated code for software developers, and allow a PSS model to align with symbolic register definitions already present in an existing software stack.

  This release of the LRM supports symbolic representation of registers in target code and defines extensions to `reg_group_c` and to `addr_reg_pkg` that enable symbolic register access as an alternative to address-handle-based access. Refer to the "Symbolic register names" section of the LRM.

- **Target-template and triple-quotes clarifications and enhancements**
  Target-template and triple-quotes clarifications and enhancements are included in this version. Refer to the examples in the "String literals" section of the LRM and the "Control-flow directives in triple-quoted string literals" table. See the "Lexical conventions" clause of the LRM.

- **Added support for designating "instance" semantics for component ref fields**
  This release of the LRM supports treating a component ref field as semantically equivalent to a non-ref component instance field, allowing component ref fields to be used as action traversal context. Refer to the "Component references" section of the LRM.

- **Added support negative offsets with "make_handle_from_claim" and "make_handle_from_handle"**
  The make_handle_from_claim and make_handle_from_handle functions now support an optional offset subtraction choice (sub) of type bool. Using a negative offset can, for example, assist in creating different virtual addresses for different CPU's using the same physical address space. Refer to the "Make_handle_from_claim function" and "Make_handle_from_handle function" sections in the LRM.

- **Added support for mutable attributes in components**
  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. However, with the new "mutable" qualifier, fields declared as mutable are an exception and may be modified after the construction of the component tree. Refer to the "Mutable component attributes" section of the "Modeling element types" clause and the "Component declarations" section in the "Formal syntax" annex in the LRM.

- **Exec block tags**
  Added support for exec block tags, enabling users to associate a structured tag value with target template exec blocks to control equivalence and deduplication during code generation. Matching exec blocks of the same kind with identical tags may be suppressed. This feature applies to selected target template exec kinds and exec file blocks, without affecting traversal, solving, or runtime behavior. Refer to the “Exec block tag” section of the LRM.

- **Support for polymorphic action implementation**
  When the referenced component context type declares an action that overrides one declared in the base type, traversing an action of the base type results in an action of the override type being traversed. This pattern supports specifying an activity in terms of action interfaces that may have multiple implementations. Refer to the “Components” section of the LRM.

- **Support for array-slice syntax with “in” and “unique” constraints**
  Array-slice syntax is now supported for *in* and *unique* constraints. This enables more-concise constraints when an expression must be limited to the values in the elements of an array subset, or when the values of a subset of array elements must be unique. See the “Set membership operator” and the “Unique constraints” sections of the LRM for additional details.

- **Support for generic constraints**
  Generic constraints support definition of reusable, parameterizable constraints. Both Boolean and value-yielding forms are supported. See the “Generic constraints” section of the LRM for more details.

Backward compatibility

- **Access operations**
  Read/write operations of PSS data from/to byte-addressable address space are defined as a set of target functions. Target exec blocks (**exec body**, **run_start**, **run_end**), and functions called from them, may call these core library functions to access allocated addresses.

  Access functions use an address handle to designate the required location within an address space.

  PSS provides a way for a tool or user to customize the implementation of access functions based on user input. The access functions have an optional parameter of type `mem_access_desc_s` declared in the `addr_reg_pkg`.

  The tool or user can extend the core library struct `mem_access_desc_c {}` to provide additional controls to the access functions.

  This additional memory access descriptor argument is not backward compatible. PSS code that overrides access primitive functions in a component derived from the `executor_c` component will need to change. For example, the following code:

```
component cpu_executor_c : executor_c<extrait_s> {
  int offset;
  target function void write32(addr_handle_t hndl, bit [32] data)
  {
    addr_reg_pkg::write32(hndl, data + offset);
  }
}
```

  Will need to be converted to:

```
component cpu_executor_c : executor_c<extrait_s> {
  int offset;
  target function void write32(addr_handle_t hndl, bit [32] data,
    mem_access_desc_s
mem_access_desc = {}) {
    addr_reg_pkg::write32(hndl, data + offset);
  }
}
```

  See the “Access operations” section in the “PSS core library” in the LRM.

- **Deprecated dynamic constraints**
  dynamic constraints are deprecated. Their functionality is replaced by a generic constraint with no parameters. See the “Generic constraints” section of the LRM





Changes & fixes

- **Clarified numeric expressions treatment**
  Clarified descriptions of numeric expressions treatment, regarding type conversions and mixing different types. See the LRM section “Evaluation rules for numeric expressions.”

Version 2.1 backward compatibility

- **Deprecated compile if with a single item**
  The **compile if** branch syntax consisting of a single item without braces, such as a single *package_body_item*, is deprecated. It should always be enclosed in curly braces.

- **Deprecated declarations of endianness_e, packed_s, sizeof_s in addr_reg_pkg**
  Prior to PSS 2.1, the declarations of `endianness_e`, `packed_s`, and `sizeof_s` core library types were in the **addr_reg_pkg** package. They are now declared in the **std_pkg** package, and referring to them via **addr_reg_pkg** is deprecated.

Version 2.0 backward compatibility

- **Deprecated exec init**
  `exec init_up` is equivalent and should be used instead.

- **Deprecated array properties .size and .sum**
  Array properties `.size` and `.sum` are deprecated. The matching array methods, `size()` and `sum()`, respectively, should be used instead.

- **Deprecated dual bounds syntax for numeric types**
  Specifying a numeric type’s width using dual bounds, such as `bit[3:0]`, is deprecated. A specification of `[N]` is equivalent to `[N-1:0]`. For example, `bit[4]` should be used instead of `bit[3:0]`.