Jump-Start Portable Stimulus Test Creation with SystemVerilog Reuse

Matthew Ballance
Mentor Graphics Corp.
8005 SW Boeckman Rd
Wilsonville, OR 97070

Abstract- **Design and verification have historically been driven by domain-specific languages. Reuse guidelines have long existed for reuse of a description within the same language or methodology, but do not exist for reusing a description across languages. An Accellera standardization effort around a Portable Stimulus Specification standard, and the existence in the industry of portable stimulus tools that can retarget an abstract test specification to multiple environments, provide a driver for the creation of such guidelines. This paper provides guidelines for structuring SystemVerilog stimulus and coverage specifications to maximize reuse with a portable stimulus specification language.**

I. Introduction

Reuse in verification has been a goal as long as verification has existed as a unique discipline. Languages such as SystemVerilog and methodologies such as the Universal Verification Methodology (UVM) have greatly helped automation and reuse in transaction-oriented simulation and accelerated-simulation verification environments. Bringing test-creation automation and reuse to an even broader set of environments and executions platforms is the goal of the Accellera Portable Stimulus Specification (PSS) standard.

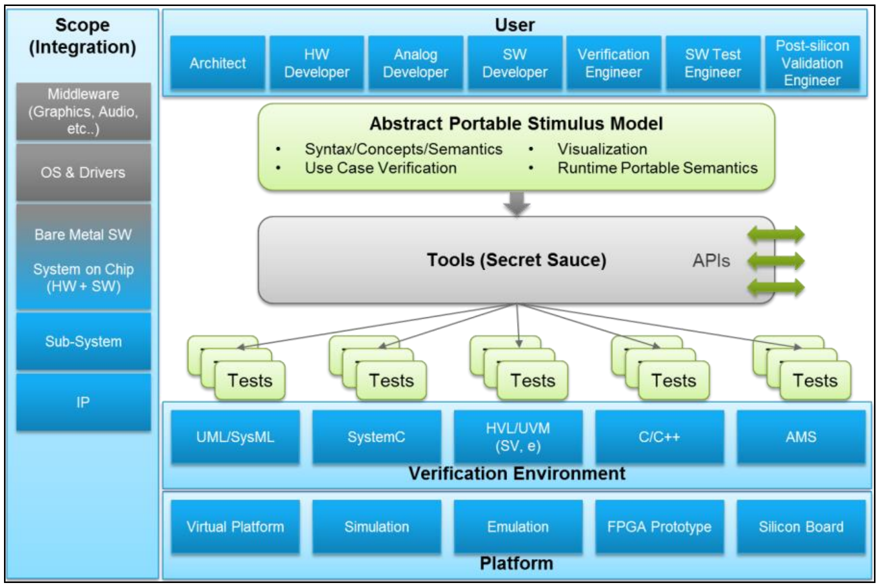

Figure 1 - Portable Stimulus Specification Goals

The goal of this standard, as summarized by Figure 1[1], is to enable multiple user constituencies to use and reuse the same specification of test stimulus, expected results, and coverage goals across a variety of platforms, including simulation, emulation, and prototype.

The developing Portable Stimulus Specification is fundamentally a declarative language that includes data structures, constraints, coverage-specification features, and graphs – a formally-analyzable specification of test procedure. Like other standards in the verification space, the developing Portable Stimulus Specification benefits from the experience of multiple vendors and users with tools that already exist in this space.

Propagation and adoption of new language standards benefit heavily from the ability to reuse existing descriptions, and the Portable Stimulus Specification is no different. Given that the bulk of verification today is done using SystemVerilog, it makes sense to see what elements of a SystemVerilog description can be easily and reliably reused in a Portable Stimulus Specification description. This paper explores the SystemVerilog constructs most easily reused in a Portable Stimulus Specification, and provides coding guidelines to make reuse simple and reliable.

II. Reuse Goals

Reuse is a pretty broad term that covers a wide variety of workflows. Reuse of a description can be as simple (and low-tech) as hand re-implementation of an algorithm, originally implemented in one language, in a different language. Clearly, there is value in leveraging the original author’s thought process, but this is hardly an automatable process.

When it comes to reuse of SystemVerilog content in a Portable Stimulus Specification description, a high degree of automation is key. The source SystemVerilog description is fluid, and will change over the lifetime of the project. Requiring a human to identify modifications to the relevant SystemVerilog source and update the Portable Stimulus Specification representation is labor-intensive, as well as being error prone.

A typical SystemVerilog to Portable Stimulus Specification reuse diagram looks something like Figure 2. The primary source for shared data structures, and constraints on those data structures, is maintained in SystemVerilog. Portable Stimulus Specification representations of those data structures and constraints are derived directly and automatically from the SystemVerilog representation. This generated Portable Stimulus Specification description is then leveraged in user-created Portable Stimulus Specification descriptions.

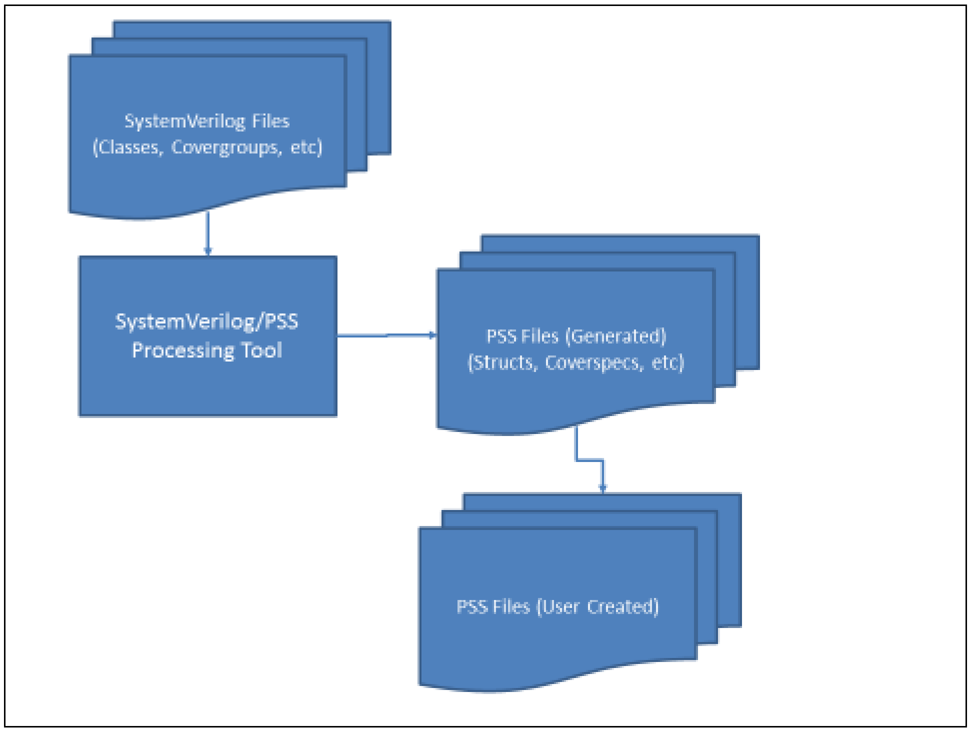

Figure 2 - Typical SystemVerilog to Portable Stimulus Reuse

Of course, an alternative reuse flow would be to declare all shared data structures and constraints on those data structures in the Portable Stimulus Specification language and automatically derive SystemVerilog representations, as shown in Figure 3. As the emerging Portable Stimulus Specification standard achieves greater industry acceptance and adoption, this flow should become more typical. However, at the moment and for the foreseeable future, there is much more content written in SystemVerilog that can usefully be leveraged as part of a Portable Stimulus Specification description.

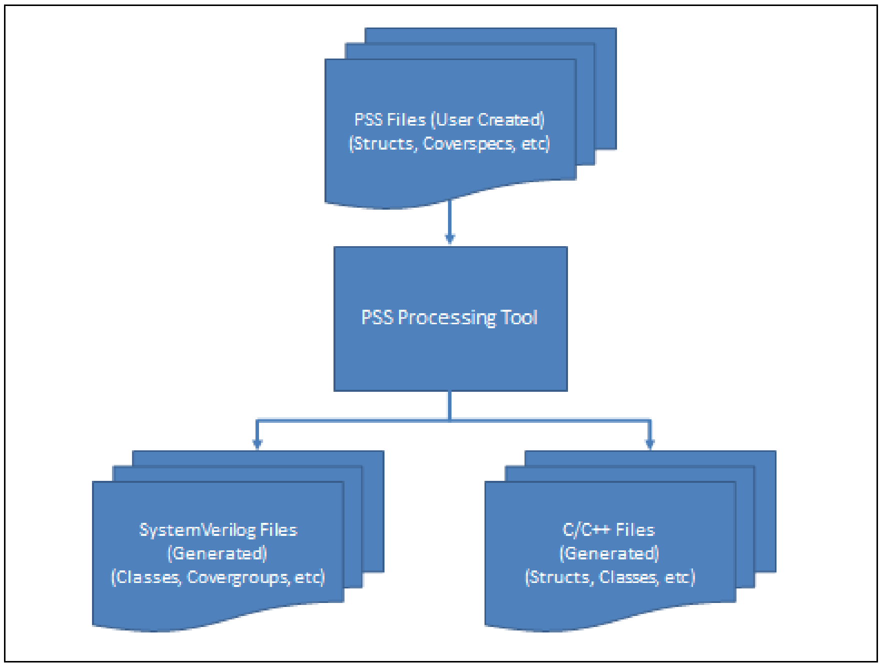

Figure 3 - PSS Primary-Source Reuse

The guidelines for structuring SystemVerilog for reuse presented in this paper reflect the preference for a fully-automated flow to reuse information captured, and reflect best practices that evolved from the experience of users of the Questa inFact Portable Test tool from Mentor Graphics.

II. Structuring SystemVerilog for Reuse

SystemVerilog is a fully-featured language for register-transfer level (RTL) design and verification [2]. As such, it includes constructs inherited from the Verilog language for describing hardware constructs. It also includes object-oriented programming constructs, and constructs for automated stimulus generation and functional coverage collection, that are used for verification of a hardware description. The emerging Portable Stimulus Specification language is predominantly a declarative language, with a few procedural constructs. In contrast, SystemVerilog is predominantly a procedural language with a few declarative constructs.

The constructs in a SystemVerilog description that are easiest to reuse as part of a Portable Stimulus Specification description are the declarative constructs: specifically data structures and constraints on the fields of those data structures, and covergroups used to model functional coverage goals.

Best practices for reuse have been around for as long as language interoperability has. Best practices exist, for example, for structuring a C library for optimal reuse from Java via the Java Native Interface (JNI). The article *Best Practices for using the Java Native Interface* provides an excellent summary of these best practices [3]. The goal in structuring SystemVerilog for reuse is to create self-contained elements of description that can be extracted from the SystemVerilog description and added to the Portable Stimulus Specification description. In general, design patterns that negatively impact reuse of SystemVerilog descriptions are ones that blend procedural and declarative description, or impact encapsulation of a declarative description. In most cases, the best practices that enable reuse are also simply good SystemVerilog coding practices that result in well-encapsulated code. Of course, there are cases where it is pragmatic or necessary to use constructs and patterns that limit reuse. Even with these cases, a little foresight can maintain a sensible balance between reusable constructs and pragmatic design patterns.

Inline Randomization Constraints

SystemVerilog allows inline constraints to be specified when an object is randomized. This can be a handy way to customize random transactions to create directed-random test sequences. Figure 4 shows a directed-random test that leverages an existing transaction class and inline constraints to perform a series of reads and writes.

```
class transaction extends uvm_sequence_item;
	`uvm_object_utils(transaction)

	rand bit[31:0]		addr;
	rand bit[31:0]		data;
	rand bit[1:0]			size;
	rand bit			rnw;
endclass


class rw_test_sequence extends uvm_sequence;
	`uvm_object_utils(rw_test_sequence)

	task body();
		transaction t1 = transaction::type_id::create("t1");
		transaction t2 = transaction::type_id::create("t2");

		for (int unsigned offset=0; offset<'h1000; offset+=4) begin
			// Do a write
			start_item(t1);
			assert(t1.randomize() with { // Size and data left open
					t1.addr == 'h8000+offset;
					t1.rnw == 0;
				});
			finish_item(t1);

			// Do a read
			start_item(t2);
			t2.addr = t1.addr;
			t2.rnw = 1;
			t2.size = t1.size; // Ensure same-sized read is performed
			finish_item(t2);
		end
	endtask
endclass
```

Figure 4 – Example of Inline Randomization

While the use of this design pattern in a SystemVerilog testbench can be very useful, it does limit reuse. The additional in-line constraints are introduced in a procedural block and often will include references to variables within the procedural block. Consequently, reusing these inline constraints is difficult if not impossible.

Inline constraints can also be abused, of course. The author has worked with testbench environments where up to a hundred lines of inline constraints were specified simply to ensure valid relationships between class fields. Worse, these large blocks of inline constraints were copied across multiple tests resulting in a maintenance headache when these base constraints needed to be adjusted.

Increase code maintainability and reuse by factoring common constraints out of inline constraint blocks. Figure 5 shows how the read/write inline constraint is factored out by creating a read transaction and a write transaction class. This increases test readability and maintainability, and provides more specific elements that can be reused in a Portable Stimulus Specification description.

```
class read_transaction extends transaction;
  `uvm_object_utils(read_transaction)

  constraint read_c { rnw == 1; }
endclass

class write_transaction extends transaction;
  `uvm_object_utils(write_transaction)

  constraint write_c { rnw == 0; }
endclass
```

Figure 5 - Factoring out Inline Constraints

Dynamic Enabling/Disabling of Constraint Blocks

Another interaction between SystemVerilog procedural code and declarative constraints is support for dynamically enabling and disabling constraint blocks. Under ideal circumstances, dynamically disabling constraints is used in very limited cases to do things such as enabling generation of illegal transactions. A highly-problematic use of dynamic constraint-block control is where at least one block must be disabled in order for randomization to complete successfully. Figure 6 shows an operation sequence-item class that specifies two conflicting constraints – one that sets "normal" operating modes, and another that sets "extended" operating modes. The user of this class must know to disable at least one of the normal_mode_c or ext_mode_c constraints prior to randomizing the object.

```
typedef enum { MODE_NORMAL_1, MODE_NORMAL_2, MODE_EXT_1, MODE_EXT_2} mode_t;

class op_item extends uvm_sequence_item;
  `uvm_object_utils(op_item)

  rand mode_t         mode;
  // ... Other fields

  constraint normal_mode_c {
    mode inside {MODE_NORMAL_1, MODE_NORMAL_2};
  }

  constraint ext_mode_c {
    mode inside {MODE_EXT_1, MODE_EXT_2};
  }

endclass
```

Figure 6 - Dynamically-Controlled Conflicting Constraints

From a reuse perspective the use of dynamically-controlled constraints presents two challenges. The current Portable Stimulus Specification language doesn't provide facilities for dynamically controlling constraints. So, at minimum, additional changes will be required in the PSS description to adapt the description imported from SystemVerilog. In addition, the use of conflicting constraint[1]s presents a usability challenge in both SystemVerilog and Portable Stimulus Specification descriptions. Valid combinations of enabled/disabled constraint blocks must be documented somewhere. At best, the valid enabled/disabled constraint blocks will be captured as a comment which will likely not propagate to the Portable Stimulus Specification description. At worst, the valid combinations of enabled/disabled constraint blocks are "tribal knowledge" captured in working test sequences.

Using class hierarchy and constraint overloading instead of dynamic constraint enable/disable results in SystemVerilog sequence items that are easier to use, and a SystemVerilog description that is easy to reuse as part of a Portable Stimulus Specification. There are two equivalent approaches to structuring a class hierarchy to handle cases often implemented using dynamic constraint enable/disable. The first approach, additive constraint refactoring, is shown in Figure 7. Using this approach, derived classes add in just the constraints needed to achieve their intended purpose. So, the normal_op_item class adds constraints to ensure that only "normal" modes are produced, while the ext_op_item class adds constraints to ensure that only "extended" modes are produced. This approach is a good general approach.

```
typedef enum { MODE_NORMAL_1, MODE_NORMAL_2, MODE_EXT_1, MODE_EXT_2} mode_t;

class op_item extends uvm_sequence_item;
  `uvm_object_utils(op_item)

  rand mode_t         mode;
  // ... Other fields
endclass

class normal_op_item extends op_item;
  `uvm_object_utils(normal_op_item)

  constraint normal_mode_c {
    mode inside {MODE_NORMAL_1, MODE_NORMAL_2};
  }
endclass

class ext_op_item extends op_item;
  `uvm_object_utils(ext_op_item)

  constraint ext_mode_c {
    mode inside {MODE_EXT_1, MODE_EXT_2};
  }
endclass
```

Figure 7 - Additive Constraint Refactoring

The second approach, subtractive constraint refactoring, is more often used to create exceptional use cases, such as error injection. Taking the example used thus far, let's assume that "normal" modes are the ones used the majority of the time.

[1] a

```
typedef enum { MODE_NORMAL_1, MODE_NORMAL_2, MODE_EXT_1, MODE_EXT_2} mode_t;

class normal_op_item extends uvm_sequence_item;
  `uvm_object_utils(op_item)

  rand mode_t         mode;
  // ... Other fields

  constraint normal_mode_c {
    mode inside {MODE_NORMAL_1, MODE_NORMAL_2};
  }

endclass

class ext_op_item extends normal_op_item;
  `uvm_object_utils(ext_op_item)

  constraint normal_mode_c {
    // empty constraint - follows reuse guideline #2
  }

  constraint ext_mode_c {
    mode inside {MODE_EXT_1, MODE_EXT_2};
  }

endclass
```

Figure 8 - Subtractive Constraint Refactoring

Figure 8 shows the definition of the normal_op_item class with constraints that enforce 'normal' operation. In order to create the 'exceptional' condition that could be created by disabling the 'normal_mode_c' constraint, the ext_op_item class overrides the normal_mode_c constraint with an empty constraint – effectively disabling this constraint in a declarative way.

Using class hierarchy and constraint refactoring results in more-reusable classes in SystemVerilog environments. It also results in SystemVerilog descriptions that are reusable in a Portable Stimulus Specification description by moving controls from procedural description into the declarative portion of the description.

References to Global Data

Referring to global data items outside the class presents a challenge for reuse – both in SystemVerilog and in a Portable Stimulus Specification description. Figure 9 shows such a case, where the coeff field in the transform_item class is bounded by a global min and max field in the global_data_pkg package.

```
package global_data_pkg;
  bit[31:0]           coeff_max;
  bit[31:0]           coeff_min;

endpackage

class transform_item extends uvm_sequence_item;
  `uvm_object_utils(transform_item)

  rand bit[31:0]        coeff;

  constraint coeff_c {
    coeff >= global_data_pkg::coeff_min;
    coeff <= global_data_pkg::coeff_max;
  }
endclass
```

Figure 9 - Constraint References to Global Data

This has its convenient aspects, of course, in allowing the coefficient bounds to be set once per testbench environment. The downside is that this makes reuse more challenging. Perhaps the coefficient bounds were global in the original testbench environment, but this may not always be the case. Perhaps in the next testbench environment there will be two UVM agents that use the transform_item class. Each of these agents may need to use different bounds for the coefficient.

```
class transform_item extends uvm_sequence_item;
  `uvm_object_utils(transform_item)

  bit[31:0]             coeff_min;
  bit[31:0]             coeff_max;

  rand bit[31:0]        coeff;

  constraint coeff_c {
    coeff >= coeff_min;
    coeff <= coeff_max;
  }
endclass
```

Figure 10 - Removing Global Data References

Figure 10 shows an approach to removing global data references by creating fields inside the class. The procedural SystemVerilog code that randomizes the refactored transform_item class will need to set the coeff_min and coeff_max prior to randomizing the class. However, the refactored code results in a class that is self-contained and easier to reuse. A self-contained class is also much more reusable within a Portable Stimulus Specification, since there are no references to external entities.

Reusable Functional Coverage

SystemVerilog covergroups provide an exceptionally flexible construct for monitoring and aggregating testbench data. However, the exceptional flexibility of the construct impacts its reusability.

```
class config_item extends uvm_object;
  `uvm_object_utils(config_item)

  rand bit[3:0]             mode1;
  rand bit[15:0]            mode1_coeff;
  rand bit[3:0]             mode2;
  rand bit[15:0]            mode2_coeff;

endclass

covergroup config_cg(ref config_item item);

  mode1_cp : coverpoint (item.mode1) {
    bins mode1[] = {[0:15]};
  }

  mode1_coeff_cp : coverpoint (item.mode1_coeff) {
    bins mode1_coeff_min = {0};
    bins mode1_coeff_mid[14] = {[1:'hfffe]};
    bins mode1_coeff_max = {'hffff};
  }

  mode1_coeff_cross : cross mode1_cp, mode1_coeff_cp;

  mode2_cp : coverpoint (item.mode2) {
    bins mode2[] = {[0:15]};
  }

  mode2_coeff_cp : coverpoint (item.mode2_coeff) {
    bins mode2_coeff_min = {0};
    bins mode2_coeff_mid[14] = {[1:'hfffe]};
    bins mode2_coeff_max = {'hffff};
  }

  mode2_coeff_cross : cross mode2_cp, mode2_coeff_cp;

endgroup
```

Figure 11 - Configuration Class and Functional Coverage

Figure 11 shows a SystemVerilog covergroup structured in a way that is ideal for reuse in a Portable Stimulus Specification description. All the coverpoints and cross-coverpoints reference the config_item class field *item* that specified as a parameter to the covergroup. This ensures that the covergroup is self-contained. In addition, the class monitored by the covergroup is the same class used for generating stimulus. This allows configurations to be generated that efficiently satisfy the coverage goals without the user providing additional directives to relate the stimulus-generation class to the class used to capture coverage data.

Summary of Best Practices

The best practices for modularity and reuse described in this section help to enable reuse across SystemVerilog environments and Portable Stimulus Specification descriptions:

1. Avoid use of inline constraints.
2. Avoid enabling and disabling constraints from procedural code.
3. Avoid referencing class-external variables from constraints

4. Structure functional coverage descriptions to explicitly reference a single class, ideally a stimulus-generation class.

The sections above illustrate alternative modeling approaches to implement the same types of situations often handled by these constructs that impede reuse.

III. Conclusion

The emerging Portable Stimulus Specification standard allows a single representation of test stimulus, expected results, and coverage goals to be captured in a description that can be reused across test environments from UVM to software-driven test environments. Leveraging portions of existing SystemVerilog-based descriptions can accelerate creation of Portable Stimulus Specification descriptions, since so much content has already been created in SystemVerilog. Following some simple coding guidelines, outlined in this paper, makes SystemVerilog classes, constraints, and covergroups more amenable to automated reuse in a Portable Stimulus Specification description. Reuse of the wealth of existing SystemVerilog descriptions, such as transactions and device configuration classes, allows users to more-quickly take advantage of the benefits of Portable Stimulus tools such as efficient closure on test goals and automated creation of C tests.

References

[1] "Proposed Portable Stimulus Diagram", Accellera Portable Stimulus Working Group, December 2014

[2] "IEEE 1800-2012: SystemVerilog – Unified Hardware Design, Specification, and Verification Language," IEEE Standards Association, February 2013.

[3] M. Dawson, G. Johnson, A. Low, "Best Practices for Using the Java Native Interface", IBM developerWorks, July 2009.