Making Legacy Portable with the Portable Stimulus Specification

Matthew Ballance
Mentor Graphics Corp.
8005 SW Boeckman Rd
Wilsonville, OR, 97071

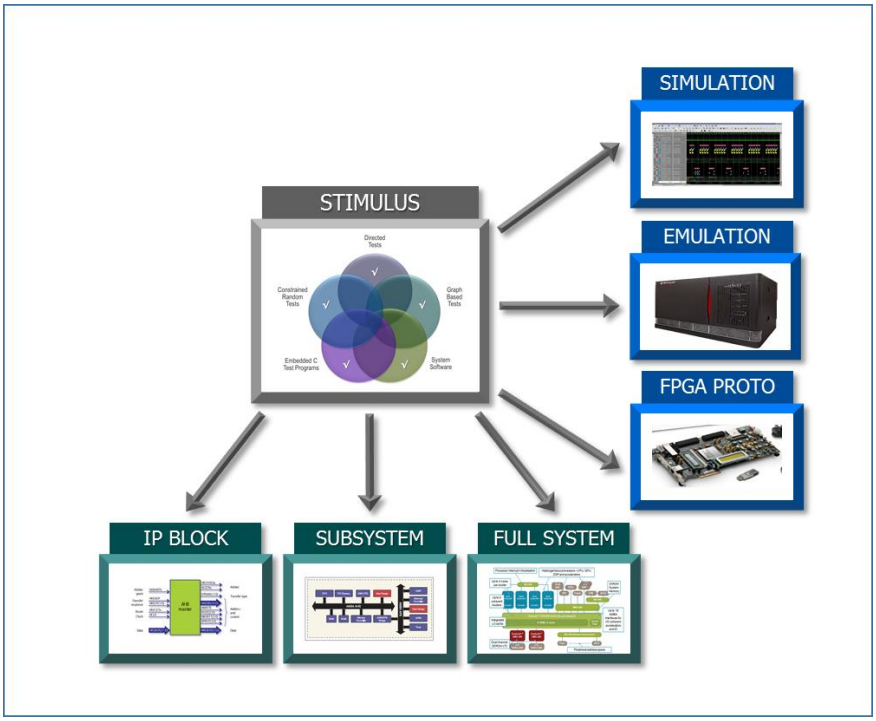
***Abstract*- In today's complex designs, more and more verification and validation is being done at the SoC level. The Accellera Portable Stimulus Working Group is refining an input specification standard that enables tools to automate creation of tests across block, subsystem, and SoC levels. Users of the emerging standard have legacy descriptions that they wish to reuse with standards-compliant tools. This paper provides best practices, via examples, for applying reuse mechanisms provided by the standard to accomplish reuse of various types of legacy descriptions with standards-compliant tools.**

I. INTRODUCTION

In today's complex designs, more and more verification and validation is being done at the SoC level – driven by the need to verify system-level effects that involve both hardware and software, and necessitated by the fast emulation and post-silicon execution platforms required to execute a meaningful amount of software in a reasonable amount of time. Block- and subsystem-level verification is still critical, of course, and reusing some portion of the test intent from block level to SoC level is critical to making testing productive at the SoC level. Automating block-level test creation, specifically by adopting constrained-random generation techniques, has been key to test-creation productivity at the block level. Automation is also critical for creating SoC-level tests, and many in-house tools have been created to automate this process.

These needs of automation and reuse have motivated the work of the Accellera Portable Stimulus Working Group (PSWG). A standardized input format allows users to achieve portability of test intent across execution platforms and verification environments, and reuse the test specifications they create across tools from multiple vendors. The PSWG has selected a baseline input-language specification that will shortly be refined into the first version of the Portable Stimulus Specification standard [1]. The emerging standard includes several mechanisms to enable existing descriptions, such as utility code, to be incorporated into tests specified in terms of the portable stimulus specification, with a goal of accelerating adoption of the standard.

II. What is Portable Stimulus?



A Portable Stimulus Specification description is a single representation of test intent that is reusable by a variety of users across different levels of abstraction, and on different execution platforms. As a consequence, it caters to the requirements and concerns of users doing verification at block level, system level, and SoC level. The Portable Stimulus Specification supports data structures, constraints, and data-centric randomization, in much the same way that existing verification languages (e.g. SystemVerilog) do. In addition, however, the Portable Stimulus Specification supports a declarative specification of control flow: sequential and parallel execution of operations, selections between operations, and loops. The fact that these portions of the specification have declarative semantics enables processing tools to apply automation, and enables the test intent to be retargeted to various platforms in ways far beyond what could be supported by simply translating existing behavioral code.

III. Anatomy of a Portable Stimulus Specification Description

A description using the Portable Stimulus Specification fits into a framework similar to what is shown in Figure 1. Just like any other language, such as SystemVerilog or C/C++, it important to understand what surrounds a description in this language.

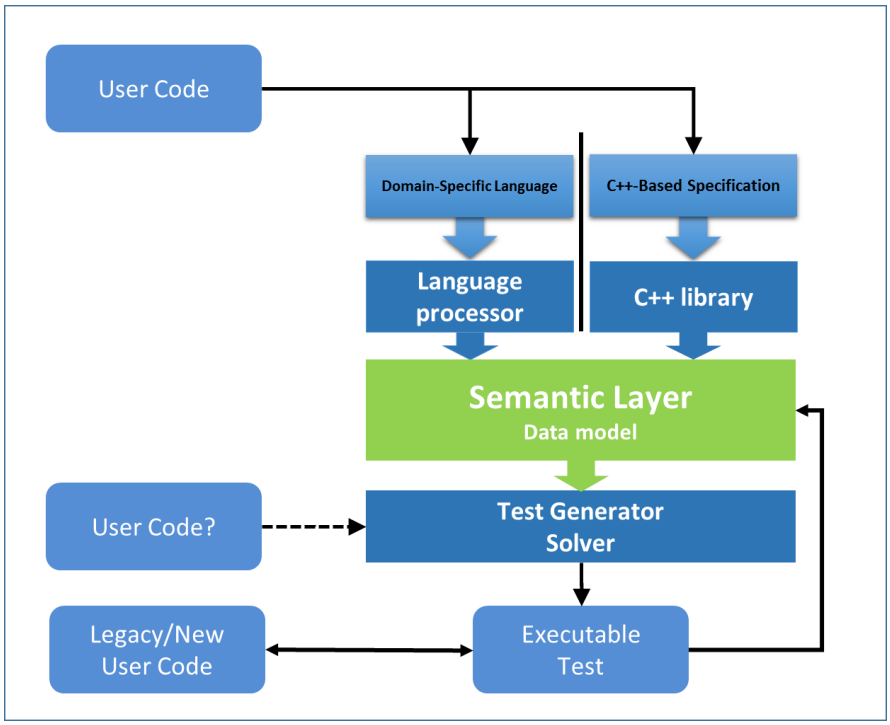


Figure 1 - Anatomy of a Portable Stimulus Description

In the case of a SystemVerilog description, it is generally assumed that a simulation engine surrounds the description. Furthermore, foreign-language code (typically C) may be linked to the SystemVerilog description via the Direct Procedural Interface (DPI).

In the case of a C/C++ description, it is important to understand that a specific set of standard library functions are expected to exist, as well as other pre-compiled libraries.

In the case of a Portable Stimulus Specification description, it is important to understand that the user-specified description of that model could be made using either of two equivalent input formats. The standard specifies a domain-specific language for capturing the key elements of test intent. The standard also specifies an equivalent C++ class library or API for capturing the key elements of test intent. Critically, both of these user-visible input formats are equivalent and will result in creation of the same semantic data model. What this means is that the same elements (actions, data structures, etc) and relationships (constraints, etc) can be specified using both input formats. This allows the choice of input format to be made based on user familiarity with a given type of language and potential integration benefits of integrating with existing tool flow via an API.

While the syntax of the standard is specified for both the domain-specific language and the C++ input format, the semantics of the standard are specified in terms of the semantic data model. Consequently, the behavior of tools in processing a portable stimulus description can be specified in terms of the semantic data model. When a tool processes the semantic data model, it must select specific values, from a universe of possible values, for random fields and select specific scenarios from the universe of legal scenarios. Doing so often requires assistance from external algorithms. The Portable Stimulus Specification standard provides a procedural interface that allows the user to specify that a processing tool must call external code at specific points in the solving process. This external code can return data that the processing tool incorporates into the solving process.

External code is also used when triggering test behavior. Design blocks often have utility code, such as the beginning of drivers, that can be used to perform basic programming. This code can be directly called by the portable-stimulus test instead of re-implementing this code inside the portable stimulus description.

IV. The Need for Reuse

Adoption of any new description is heavily supported by the ability users have to leverage existing descriptions, and 'evolve' to use more features of the new description. Adoption of the SystemVerilog language standard was facilitated by the fact that SystemVerilog naturally extended from the Verilog language. Adoption of C was facilitated by the fact that it was easy to call existing assembly-language routines from C, and adoption of C++ by the fact that the object-oriented features of C++ build on top of C, allowing existing C libraries and code to be used by object-oriented C++ code.

The situation with Portable Stimulus is slightly different. Because the Portable Stimulus Specification is a declarative description, it doesn't naturally extend from any existing procedural language. This is true, despite the fact that a portable stimulus description can be procedurally created using a C++ class library to programmatically build up the declarative model. So, moving to a portable stimulus description represents a greater discontinuity than the process or moving from C to C++ or Verilog to SystemVerilog.

Users adopting the new portable stimulus standard have a wealth of existing descriptions that could be leveraged. Identifying and efficiently leveraging these sources of reusable description enables the portability and automated scenario-creation benefits enabled by the portable stimulus standard to be rapidly realized.

V. Opportunities for Reuse

We can identify several opportunities for reuse by examining the interface points between user code and the portable stimulus processing flow shown in Figure 1. Some types of descriptions map nicely to the declarative style of description. Existing procedural code that can be used during the solve process to create expected results and manage procedurally-heavy processes such as memory management. Existing procedural code can also be used during test execution to carry out the test intent described by the Portable Stimulus Specification description.

Each of these opportunities for reuse has tradeoffs in terms of the cost/benefit to achieve reuse. These opportunities for reuse for will be explored in more detail below.

A. Translating Existing Declarative Descriptions

Declarative descriptions exist within verification environments today. For most verification engineers, the most readily-available source are the random variables and constraints in the classes of a SystemVerilog testbench environment. When properly structured, these random variables and constraints are easily reused within a Portable Stimulus description. The fact that the Portable Stimulus constraint constructs are consistent with SystemVerilog is instrumental in making this type of reuse possible.

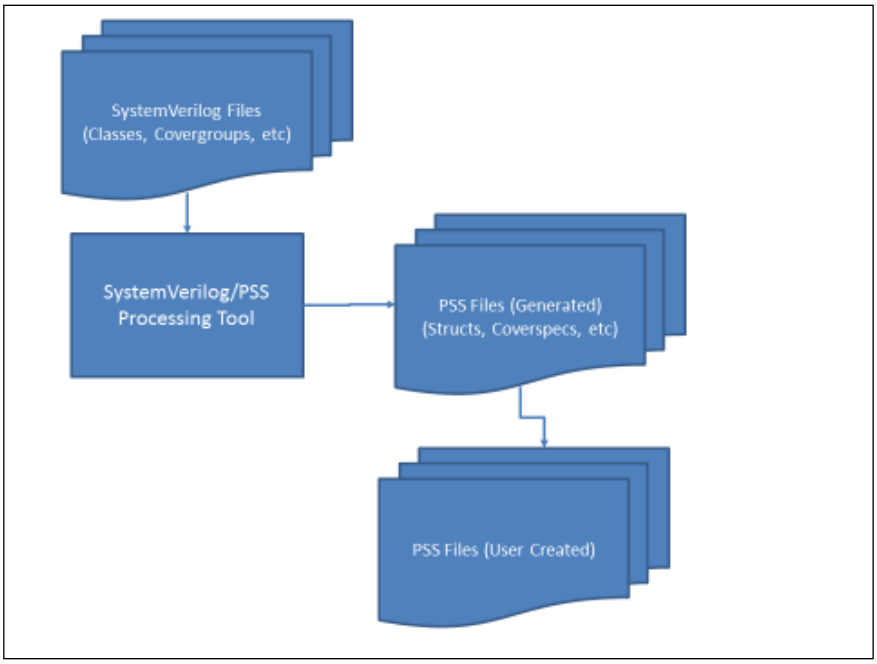


Figure 2 - Reuse flow for SystemVerilog Declarative Description

Figure 2 shows a typical flow for reusing SystemVerilog classes in a Portable Stimulus description via translation. A processing tool reads in the SystemVerilog source code, identifies classes and/or covergroups to translate, then generates corresponding language constructs in the Portable Stimulus Specification language.

```
typedef enum {
  MODE_1,
  MODE_2,
  MODE_3,
  MODE_4
} mode_t;

class device_cfg;
  rand mode_t     mode;
  rand bit[15:0]  coeff1;
  rand bit[15:0]  coeff2;

  constraint coeff_c {
    coeff1 <= coeff2;
  }

  constraint mode_c {
    if (mode == MODE_1 ||
        mode == MODE_2) {
      coeff1 == 0;
    }
  }
endclass
```

SystemVerilog

```
enum mode_t {
  MODE_1,
  MODE_2,
  MODE_3,
  MODE_4
}

struct device_cfg {
  rand mode_t     mode;
  rand bit[15:0]    coeff1;
  rand bit[15:0]    coeff2;

  constraint coeff_c {
    coeff1 <= coeff2;
  }

  constraint mode_c {
    if (mode == MODE_1 ||
        mode == MODE_2) {
      coeff1 == 0;
    }
  }
}
```

Portable Stimulus Specification

Figure 3 - SystemVerilog Translation to Portable Stimulus Specification

Figure 3 shows an example of translating a SystemVerilog class that captures device-configuration rules to the Portable Stimulus Specification language. Note that the syntax of the constraints is the same, and the major difference is the use of the ‘class’ type in SystemVerilog and the ‘struct’ type in the Portable Stimulus Specification language. This reflects the fact that data structures in portable stimulus are pure data structures, and do not include methods the way that SystemVerilog classes do.

Classes and random constraints must be structured such that they are modular and the constraints have no dependencies on class-external or procedural aspects of the SystemVerilog description. In the paper *Jump-Start Portable Stimulus creation with SystemVerilog Reuse* [2], the author proposes the following guidelines for structuring SystemVerilog for optimal reuse in a portable stimulus description:

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

B. Existing Proprietary File Formats

Existing in-house tools often use portable stimulus techniques for very targeted verification tasks, such as creating register-checking tests that can be implemented in both UVM and embedded C code. It often makes the most sense to tailor the input to these tools to the verification task. So, for example, a register-checking tool might accept a spreadsheet of the register addresses, access restrictions, and expected results.

In many cases, existing file formats can be translated to the portable stimulus specification language. Another option is to use the C++ class library input path, since there is often existing C/C++ code that can read the proprietary file format.

C. Existing Test Realization Code

Perhaps the biggest opportunity to reuse legacy descriptions with Portable Stimulus is existing test realization code. Test realization code is the code that carries out test intent. Examples of test realization code include:

- UVM sequences that carry out lower-level operations
- SystemVerilog utility tasks
- C/C++ driver-stub code for programming an IP

The procedural interface provided by the portable stimulus specification enables external functions to be referenced from a portable stimulus description in order to implement test behavior.

```
#include <stdint.h>

typedef enum {
  MODE_1,
  MODE_2,
  MODE_3,
  MODE_4
} mode_t;

void set_mode(mode_t m);

void set_coeff(uint8_t coeff_num, uint16_t coeff);

void init(void);
```

Figure 4 - Device Programming API

Figure 4 shows a C utility API we might use to configure a device. There's a method to set the device mode, a method to set one of the coefficients, and another API to finalize the device configuration.

```
import void set_mode(mode_t m);
import void set_coeff(bit[3:0] coeff_num, bit[15:0] coeff);
import void init();

component device_comp {

  action do_device_cfg {
    rand device_cfg cfg;

    exec body {
      set_mode(cfg.mode);
      set_coeff(1, cfg.coeff1);
      set_coeff(2, cfg.coeff2);
      init();
    }
  }
}
```

Figure 5 - Calling Test Realization Code

Figure 5 shows how this API might be used from within a portable stimulus description. An *action* named *do_device_cfg* encapsulates the behavior and data for configuring the device. Function prototypes are specified for the external methods. Portable Stimulus tools may provide utilities for automatically extracting the method prototypes from C/C++ header files.

The portable stimulus specification provides an exec block of type *body* to specify how the behavior of an action is realized. In this case, the *do_device_cfg* action is implemented by calling set_mode and passing the randomly-selected *mode* field, calling set_coeff twice with the values of *coeff1* and *coeff2*, then calling the init method to finalize the device configuration.

```
class device_reg_seq extends uvm_sequence;
  // ...
endclass

class device_api_uvm;
  uvm_sequencer #(reg_seq_item)seqr;

  task set_mode(mode_t mode);
    device_reg_seq seq = device_reg_seq::type_id::create("seq");
    seq.item.addr = MODE_REG;
    seq.item.data = mode;
    seq.start(seqr);
  endtask

  task set_coeff(byte unsigned id, shortint unsigned value);
    // ...
  endtask

  task init();
    // ...
  endtask

endclass
```

Figure 6 - Reusing UVM Sequences

Figure 6 shows how we might reuse UVM sequences for programming the device. Note that in this case the portable stimulus description is unchanged from Figure 5. In this case, the external functions are implemented as SystemVerilog tasks that use a UVM sequence to program registers within the device. While only the implementation for *set_mode* is shown, the other function implementations will be very similar.

Using external test realization code allows the portable stimulus description to be more abstract than if the portable stimulus description was refined down to a specific implementation – for example, writing registers or launching UVM sequences. This makes the description more-easily portable, encourages reuse of existing code, and helps to avoid bugs.

D. Existing Modeling Code

Test realization code is tightly coupled to triggering test behavior when the test runs. Another class of code is more concerned with modeling the data used when triggering test behavior. Distinguishing between test realization and modeling code is most important when using a test flow that separates test-generation activity from test execution activity. For example, when the portable stimulus processing tool generates a standalone C test on the host workstation that then is compiled and executes on the embedded process of the design.

```
import bit[31:0] alloc(bit[31:0] sz);

struct buf_addr {
  rand bit[15:0]    sz;
  bit[31:0]         addr;

  exec post_solve {
    addr = alloc(sz);
  }
}

component top_comp {

  action do_write {
    rand buf_addr   buffer;
    // ...
  }

  action entry {
    do_write    wr1, wr2;

    constraint c {
      wr1.buffer.sz != wr2.buffer.sz;
    }

    graph {
      wr1;
      wr2;
    }
  }

}
```

Figure 7 - Reusing Modeling Code

Memory allocation is a classic use of modeling code. In the example shown in Figure 7, we want to control and constrain the size of memory buffers that we write to. The address, however, is largely irrelevant as long as it is valid and doesn't overlap the address of another memory buffer. Reusing an existing memory-allocation implementation is far simpler than attempting to create our own directly in the portable stimulus language. And, doing so doesn't cost us anything in terms of the functionality of the overall test.

The Portable Stimulus Specification provides a mechanism specifically to enable interaction between existing behavioral code and the solver within the processing tool that evaluates a Portable Stimulus Specification description. Exec blocks in a portable stimulus specification description specify the interaction between the portable stimulus description and external code. Exec blocks of type pre_solve are evaluated prior to the solver selecting specific values for random variables. External code can called to initialize the value of non-random variables that the solver will later use. Exec blocks of type post_solve are evaluated after the solver selects specific values for random variables. External code can be called to compute values for non-random variables based on the specific value selected by the solver for random variables.

In the example in Figure 7, we declare an external function named *alloc*. We call this function from the *post_solve* exec block of the buf_addr data structure. According to the portable stimulus semantics, the *post_solve* exec block will be executed after values are selected for the random fields of the buf_addr struct. So, the value for *sz* will have already been selected, and we can simply call the *alloc* function to select a valid address.

VI. Conclusion

Reuse of legacy descriptions is key to rapid adoption of any new computer language, and the emerging portable stimulus specification language is no different. Recognizing the need for reuse, the standard built in mechanisms to make reuse of existing descriptions easy. The data structures and constraint syntax supported by the standard allow users to easily reuse existing random data structures and constraints from their SystemVerilog environments. The procedural interface allows easy reuse of utility code for triggering test behavior, as well as reusing data-computation algorithms. Together, these three reuse mechanisms maximize the use of legacy code when adopting the emerging portable stimulus description.

References

[1] Accellera Portable Stimulus Working Group - http://workspace.accellera.org/apps/org/workgroup/pswg/

[2] M. Ballance, *Jump-Start Portable Stimulus Test Creation with SystemVerilog Reuse*, DVCon, March 2016 - https://s3.amazonaws.com/verificationacademy-news/DVCon2016/Posters/dvcon-2016_jump-start-portable-stimulus-test-creation-with-systemverilog-reuse_poster_paper.pdf