2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

SystemVerilog Format of Portable Stimulus

Wayne Yun, David Chen, Theta Yang, Evean Qin
Advanced Micro Devices, Inc.[1]
1 Commerce Valley Dr. East
Markham, ON Canada L3T 7X6

***Abstract*-The Portable Test and Stimulus Standard defines two input formats, namely DSL and C++. A portable stimulus specification can be created accordingly. This specification can be regenerated and run on different platforms like simulation, emulation, post-silicon, etc. A simulation platform usually starts earlier than others. It is naturally where the specification is developed. The majority of simulation environments are in SystemVerilog. This mismatch of programming language increases overhead and reduces the value added by portable stimulus. A SystemVerilog input format is defined in this paper. It is semantically identical to other two input formats and eliminates the language mismatch. It is prototyped and demonstrated feasible.**

I. Introduction

The Portable Test and Stimulus Standard [1] defines a specification for creating a single representation of stimulus and test scenarios, usable 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 simulation, emulation, FPGA prototype, post-silicon, etc.

A set of behaviors captured following this standard is called a portable stimulus specification (PSS). Enabled by this standard, a PSS can be specified once, from which multiple implementations may be derived. A PSS can be reused at multiple levels like IP, subsystem, SoC, emulation, etc.

Moving along with the project execution flow, a PSS is likely to be developed at the IP level in a simulation environment, then reused at higher levels. An industry survey [2] indicates that most simulation environments use SystemVerilog language [3] and Universal Verification Methodology (UVM) [4].

The Portable Test and Stimulus Standard describes two input formats. One is a domain-specific language (DSL). The other is C++. When a PSS is deployed in a SystemVerilog environment, a new language has to be introduced and additional tools may be required. This complicates the simulation environment and generates more learning requirements. For many established IP UVM environments, some existing sequences have to be rewritten in an unfamiliar language. At a late stage of a project, when a hack is needed, it may have to cross a language boundary. These aspects impact effort, time, and cost negatively. Hence, they reduce the value added by adopting PSS.

This paper defines a SystemVerilog input format for creating PSS. It provides the same semantics as DSL and C++, but it is in the SystemVerilog language and is compatible with UVM. It largely eliminates the above concerns. Since its backend libraries are available in source code, the issues can be fixed quickly and new features can be added easily, especially for customized use cases.

DSL, C++, and SystemVerilog input formats can be converted between each other since they are semantically identical. Such, existing portable stimulus tools can still be supported.

II. Major Language Elements

A. Package

A PSS package is described as a SystemVerilog package. Items defined in a package can be referred to by items from another package. Packages can import selected or all items from another package, too. Figure 1 is an example.

[1] AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

© 2018 Advanced Micro Devices, Inc. All rights reserved.

```
// SystemVerilog
package my_pkg;
   …
endpackage : my_pkg
```

```
// DSL
package my_pkg {
   …
}
```

Figure 1 Example of Package

B. Component

A PSS component is described as a SystemVerilog package which contains a class derived from base class pss_component. Declarations including definitions of actions are in the package, instantiations are in the class. Figure 2 is an example.

```
// SystemVerilog
package my_comp_pkg;
   // declarations
   …
   class my_comp extends pss_component;
      // instantiations
      …
   endclass
endpackage : my_pkg
```

```
// DSL
component my_comp {
   // declarations
   // instantiations
   …
}
```

Figure 2 Example of Component

C. Action

A PSS action is described as a class derived from base class pss_action defined in a component package. Figure 3 is an example.

```
// SystemVerilog
package my_comp_pkg;
   …
   class my_action extends pss_action;
      …
   endclass
endpackage : my_pkg
```

```
// DSL
component my_comp {
   …
   action my_action {
      …
   }
}
```

Figure 3 Example of Action

D. Activity

A PSS action can have an activity. The activity is described as a task of the action class. Figure 4 is an example. Constraint and bind statements are moved to the action scope from inside the activity. Since an activity is unique in an action, these constraints and bind statements are unambiguous.

```
// SystemVerilog
package my_comp_pkg;
   …
   class my_action extends pss_action;
      …
      task activity();
        …
      endtask
   endclass
endpackage : my_pkg
```

```
// DSL
component my_comp {
   …
   action my_action {
      …
      activity {
        …
      }
   }
}
```

Figure 4 Example of Activity

Scheduling statements, namely sequence, parallel and schedule, are described with predefined functions and tasks. Figure 5 is an example of scheduling statements.

```
// SystemVerilog
class my_action extends pss_action;
   task activity();
     parallel_begin();
       sequence_begin();… scheduling_end();
       schedule_begin();… scheduling_end();
     scheduling_end();
   endtask
endclass
```

```
// DSL
action my_action {
   activity {
      parallel {
        sequence {…}
        schedule {…}
      }
   }
}
```

Figure 5 Example of Scheduling Statements

Symbol is implemented as a SystemVerilog task. Figure 6 is an example of a symbol.

```
// SystemVerilog
class my_action extends pss_action;
   task symbol_1();
     sequence_begin();… scheduling_end();
   endtask
   task activity();
     symbol_1();
   endtask
endclass
```

```
// DSL
action my_action {
   symbol symbol_1 {
     …
   }
   activity {
     symbol_1;
   }
}
```

Figure 6 Example of a Symbol

Traversal of an action-type field is described as a few function and task calls in SystemVerilog. A macro can be defined to generate these calls. Figure 7 is an example of traversing an action-type field.

```
// SystemVerilog
class A extends pss_action;
   …
endclass
class B extends pss_action;
   A a;
   task activity();
     a.pre_activity();
     a.randomize();
     a.activity();
     a.post_activity();
   endtask
endclass
```

```
// DSL
action A {
   …
}
action B {
   A a;
   activity {
     a;
   }
}
```

Figure 7 Example of Traversing an Action-Type Field

An action variable field is traversed in a similar way with a helping dummy action. A macro can be defined for the declaration of it. Another macro can create the code of traversal. Figure 8 is an example of action variable field.

``// SystemVerilog
class A extends pss_action;
   rand int var;
   pss_action _action_var = new();
   task activity();
     _action_var.pre_activity();
     var.randomize();
     _action_var.post_activity();
   endtask
endclass```
	``// DSL
	action A {
	   action int var;
	   activity {
	     var;
	   }
	}```

```
// SystemVerilog
class A extends pss_action;
   rand int var;
   pss_action _action_var = new();
   task activity();
     _action_var.pre_activity();
     var.randomize();
     _action_var.post_activity();
   endtask
endclass
```

```
// DSL
action A {
   action int var;
   activity {
     var;
   }
}
```

Figure 8 Example of an Action Variable Field

Anonymous action traversal is described using a temporary action-type variable. A macro can be defined to reduce the amount of code needed to be typed. Figure 9 is an example of anonymous action.

```
// SystemVerilog
class A extends pss_action;
endclass
class B extends pss_action;
   task activity();
     begin
       A tmp;
       tmp = new();
       tmp.pre_activity();
       tmp.randomize();
       tmp.activity();
       tmp.post_activity();
     end
   endtask
endclass
```

```
// DSL
action A {
}
action B {
   activity {
     do A;
   }
}
```

Figure 9 Example of Anonymous Action

PSS constraints use syntax of SystemVerilog. Hence, most constraints are the same. SystemVerilog inline constraints have a few restrictions because they are local and not considered while the containing class is randomized. Consequently, inline constraints are not included while resolving input, output, resource and/or component bindings. The comp variable of an action is a SystemVerilog class handle, and the handle itself is not randomized. To overcome this difficulty, comp variable constraints are described using predefined functions. These constraints are passed to the solver to consider.

E. Flow Objects

Buffer, stream and state objects are defined as SystemVerilog classes derived from pss_buffer, pss_stream, and pss_state base classes respectively. Figure 10 is an example of a buffer object.

```
// SystemVerilog
class buffer_s extends pss_buffer;
   …
endclass
```

```
// DSL
buffer buffer_s {
   …
}
```

Figure 10 Example of a Buffer Object

Actions specify flow object references as inputs and outputs. A random SystemVerilog class handle is used for this purpose. A helping class is defined to gain access to the handle. This helping class and its instantiation can be created by a macro. The base class of the helping class distinguishes input from output. Input flow object classes are derived from pss_input base class, and output ones from pss_output. Figure 11 is an example of output using a buffer object.

// SystemVerilog	// DSL

class prod_a extends pss_action;
   rand buffer_s out;
   class _access_out extends
                    pss_output#(prod_a);
     function void set(pss_common_if p);
       $cast(action.out, p);
     endfunction
   endclass
   _access_out _i_access_out=new(this);
   …
endclass
	action prod_a {
	   output buffer_s out;
	   …
	}

Figure 11 Example of Output

F. Resource Objects

A resource object is a child class of base class pss_resource. Figure 12 is an example of resource object.

// SystemVerilog	// DSL

class channel_s extends pss_resource;
   …
endclass
	resource channel_s {
	   …
	}

Figure 12 Example of Resource Object

An action can claim a resource in lock or share mode. A helping class selects the mode and provides access to the handle. This helping code could be generated by a macro. Figure 13 is an example of a lock mode resource.

// SystemVerilog	// DSL

class prod_a extends pss_action;
   rand channel_s chan;
   class _access_chan extends
                     pss_lock#(prod_a);
     function void set(pss_common_if p);
       $cast(action.chan, p);
     endfunction
   endclass
   _access_chan _i_access_chan=new(this);
   …
endclass
	action prod_a {
	   lock channel_s chan;
	   …
	}

Figure 13 Example of Locked Resource

G. Pools

A pool is described with the parameterized class pss_pool. Figure 14 is an example of pool.

// SystemVerilog	// DSL

class main_c extends pss_component;
   pss_pool #(channel_s, 4) chan_p;
endclass
	component main_c {
	   pool [4] channel_s chan_p;
	}

Figure 14 Example of Pool

Pool binding is described using a predefined function. Figure 15 is an example of pool binding.

```
// SystemVerilog
class main_c extends pss_component;
   _bind_flag=pss_bind("chan_p", "");
endclass
```

```
// DSL
component main_c {
   bind chan_p {*};
}
```

Figure 15 Example of Pool Binding

H. Type Extension

A type extension of flow control object types, action, or component is implemented as an add-on class object. Multiple extensions can co-exist together. To enable these extensions to access the original class object, a class handle "orig" is declared in each add-on. The "orig" handle points to the original class object. Figure 16 is an example of defining a type extension of an action.

```
// SystemVerilog
class my_action extends pss_action;
   int cntl;
end
class my_ext extends pss_extend#(my_action);
   rand int move;
   constraint t {move<=orig.cntl;}
endclass
```

```
// DSL
action my_action {
   int cntl;
}
extend action my_action {
   rand int move;
   constraint t {move <= cntl;}
}
```

Figure 16 Example of Action Type Extension

Enum type extension is not directly supported by SystemVerilog. As a workaround, a PSS enum variable is defined as an integer having a constraint of valid values. The queue of valid values can be extended to implement the extension of the enum type. Figure 17 is an example of enum type extension.

```
// SystemVerilog
typedef enum {
  GOOD, ERROR
} status_e;
int status_e_values[$] = {GOOD, ERROR};
rand int dev_status;
constraint dev_status_value {
   dev_status inside {status_e_values};
}
typedef enum {FATAL=ERROR+1} ext_status_e;
   …
status_e_values.push_back(FATAL);
```

```
// DSL
enum status_e {
   GOOD, ERROR
}
rand status_e dev_status;
extend enum status_e {FATAL};
```

Figure 17 Example of Enum Type Extension

Type override is implemented using factory. When SystemVerilog format of PSS is in a UVM environment, UVM factory is used. Otherwise, a standalone factory provides same functionality.

III. CLASS SYSTEM

The class system is designed to be able to mix into existing SystemVerilog environments like UVM ones. It is also capable to run standalone. Interface classes are defined for referencing all types of PSS objects. Figure 18 shows these interface classes. Interface class pss_common_if is implemented by every class. It can be used to point to any class object.

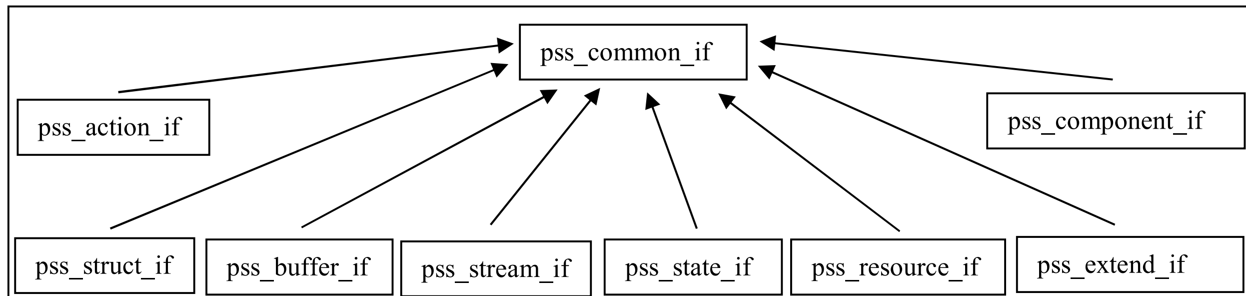

Figure 18 Inheritance of Interface Classes

Base classes for language elements are derived from a parameterized class type and implement an interface class. Figure 19 is an example of base class pss_action.

```
class pss_action #(type BASE=pss_object) extends pss_common#(BASE) implements pss_action_if;
   …
endclass : pss_action

class pss_common #(type BASE=pss_object) extends BASE implements pss_common_if;
   …
endclass : pss_common
```

Figure 19 Example of Base Class pss_action

Class pss_action has a parameter BASE whose default is pss_object. Class pss_object defines common functions for the standalone mode. It can be redefined or replaced by another class, for example uvm_seqeunce. Such a PSS action is also a UVM sequence and compatible to any UVM environment.

Class pss_common implements common functions of PSS for every mode including UVM and standalone. Prototypes of these functions are defined in interface class pss_common_if. Since it is parent class of other interface classes, these functions are directly available at every base class of language element.

These base classes collect information of the PSS and save to internal data structures. Internal data structures store information of components, actions, flow control objects, resource objects, pools, various binds, variable comp constraints, class extensions, etc. The solver relies on such information to interpret the PSS and generate scenarios. Helping classes also provides access of class handles to the solver.

The solver is part of the component. It finds all actions first from class registration database. Then, it builds a map of flow control objects. After flattening the target action, it iterates through all legal flow control connections at every connecting node. Resource allocation is considered, too. For each graph successfully built, it is also randomized to check if constraints are met while setting random variable values. Once all conditions are satisfied, every action in the graph is executed according to their scheduling relations.

IV. Results

An example with extensions was tested in addition to basic PSS functionality. The SystemVerilog format is demonstrated to be feasible. Certain DSL features like data range are not directly supported by SystemVerilog syntax. A constraint has to be added for such a case. All tests were done using Synopsys VCS 2018.09 though standard SystemVerilog simulators are expected to work.

V. Future Works

A. *Generation Flow*

Qualification of the class system and examples were run in integrated generation and execution mode. Pre-generation of tests is designed to be supported. Further work is needed to test out.

B. *DSL and C++ Generator*

SystemVerilog provides VPI to introspect the code. Using such introspective information of the code, it is possible to generate DSL and/or C++ format PSS from the SystemVerilog one.

C. *Cleaner Source Code*

Macros are used to collect information and facilitate execution. Such code can be cleaned up if the SystemVerilog format PSS is compiled twice. After the first, the code can be inspected via VPI and the helping code can be generated. Which is included in the second compile and provides information of the PSS. Consequently, the source code can be cleaner.

D. *Syntax not supported yet*

A few cases of DSL syntax are not native supported by SystemVerilog, like data range, C type constants, constraints in activity, inline constraints of comp, etc. Workarounds exit. Investigation is needed to decide if they should be supported and how.

E. *Coverage Based Tests*

Currently coverage is only collected using SystemVerilog syntax, but not used for stimulus generation. Enhancement can be added to generate tests based on coverage specification.

VI. Summary

The Portable Test and Stimulus Standard defines DSL and C++ input formats. However, most simulation environments are of SystemVerilog. This language mismatch reduces the value of portable stimulus. This reduction can be avoided by a SystemVerilog input format which is defined in the paper. The base classes can be used with UVM or standalone. Experimental SystemVerilog specifications worked as expected. The SystemVerilog format is semantically identical to DSL and C++. It enables UVM testbenches to implement portable stimulus in the same language and also provides more flexibility in defining new features.

References

[1] Accellara Systems Initiative, *Portable Test and Stimulus Standard,* Version 1.0, http://www.accellera.org/images/downloads/standards/pss/Portable_Test_Stimulus_Standard_v1.0.pdf, June 2018

[2] Verification Academy, *Industry Data and Survey*, https://verificationacademy.com/seminars/archived-industry-data-and-surveys, October 2018

[3] IEEE 1800-2017, *IEEE Standard for SystemVerilog,* https://standards.ieee.org/standard/1800-2017.html

[4] IEEE 1800.2-2017, *IEEE Standard for Universal Verification Methodology Language Reference Manual,* https://standards.ieee.org/standard/1800_2-2017.html