Bridge the Portable Test and Stimulus to UVM Simulation Environment

Theta Yang, Evean Qin
Advanced Micro Devices
({theta.yang,even.qin}@amd.com)

Abstract: The adoption of Portable test and Stimulus Standard (PSS) [1] enables function verification can create test intent at abstract level and generate different test scenarios for a specific simulation engine. Given that Universal Verification Methodology (UVM) [2] is by far the most popular verification framework used in the industry, by bridging PSS to UVM, the verification engineer can specify test intents in PSS Domain Specific Language (PSS DSL), and leverage the UVM verification sequence library underneath to create concreted tests. The paper describes a prototype of pss_uvm, the framework to compile and synthesize PSS script into to a simulation test based on UVM testbench. With the tool set in pss_uvm, a test case can be constructed in PSS DSL script, and converted into ruby, a dynamic program language, with which it can be run and interacted with a SystemVerilog HDL simulator through binding it to UVM testbench with SystemVerilog DPI. The verification framework helps adopting PSS seamlessly without major change in the simulation flow and testbench, as well as, augments the distinct capabilities to UVM such as dynamic programming and ease of debug. By using PSS script on top of UVM, the verification testcases can be specified in PSS and synthesized to UVM test, which improves the efficiency of verification.

I. INTRODUCTION

With the portable test and stimulus standard, an abstract representation of test stimulus could be created and reused by a variety of users, across different levels of integration, or under different configurations. It enables different implementations of the same scenario that runs on various testing platforms such as simulation or emulation [1]. The adoption of PSS standards requires simulation tools to interpret inputs in PSS specification language, construct the test scenarios from abstract representation and execute the simulation in a user-friendly way. In this paper, a *pss_uvm* verification framework will be introduced and depicted to allow PSS script to be integrated and run in a UVM simulation environment. In this framework, PSS script is compiled and converted into its equivalence in ruby script, which can be evaluated in UVM simulation environment with the help of a runtime library called *ruby_uvm*. It enables the users to implement the PSS intuitively and leverage with the distinct capabilities augmented to UVM such as dynamic programming and ease of debug. By using PSS script on top of UVM, the verification scenarios can be specified in PSS and synthesized to UVM test, which improve the efficiency of verification.

UVM is by far the most popular verification framework, however UVM based simulation environment has a few limitations,

- Hard to debug: The interactive debugger of SystemVerilog/UVM depends on simulator from EDA vendors, and it is not easy to change the content of testcase when the simulation is undergoing.
- Long learning curve: UVM is harder to learn than most modern script languages. It will be an interested attempt if we can use script to do verification and gain the same verification quality.

The goal for implementing PSS on top of UVM is to specify test intents in PSS and implement test scenario in UVM. Leveraging the support of constrained randomization and coverage analysis provided from PSS, the test scenarios can potentially be generated automatically. Through simulating the automatically created test scenarios, the test intent can be confirmed by back-annotating the simulation result with the corresponding coverage, where the user can close the verification cycle. Bridging PSS framework and UVM testbench with a modern script language like ruby, we can obtain the following benefits,

- Speed-up development: test scenarios are specified in abstract level (PSS) and converted in concrete representation in ruby. PSS allows us to develop the test indent once and used consistently by different verification teams working on different verification engines such as simulator, accelerator or emulator. Meanwhile, ruby is ubiquitous and easy to be picked up and deployed by different teams for adopting PSS. The setup enables work sharing among horizontal verification teams, and makes the test reuse possible among vertical teams such as IP and SOC teams. By eliminating the unnecessary repeated tasks, the development of semiconductor project potentially get speeded up as the design verification cycle is shortened.
- Easy to learn and use: The functionality of UVM is exported to high level of abstraction (by C through DPI) and executed from ruby extension library. The verification team uses PSS empowered by ruby script to drive UVM verification environment, and develops test cases in ruby instead of raw UVM, which reduces the learning time for new engineers because ruby is comparatively easier to learn and debug.
- Nonintrusive with existing UVM library: PSS DSL and underneath *ruby_uvm* library are developed in a nonintrusive way as an add-on functionality of existing UVM testbench. The deployment of *ruby_uvm* library and PSS abstract test intent does not have any impact to the current UVM based test environment. If there are testcases already developed in native UVM, they will work as well without any change.
- Compile-free method: The PSS script is loaded after simulation image being built, so the user can modify the test scenario in the script without recompiling the simulation image. In other words, if DUT and the UVM common

code are not changed in the iteration, then with only PSS DSL code changes, the user does not need to recompile it when trying out fixes or running new tests.

The proposed pss_uvm verification framework consists of a PSS translator which compiles PSS script to ruby test, a Ruby interpreter, a PSS object model library, a *ruby_uvm* library and a base UVM testbench. After evaluated by the PSS translator, the Ruby interpreter builds PSS object models according to the ruby script translated from PSS, which internally represent the PSS abstract test intents. The models are based on a set of predefined ruby classes implementing PSS functionality such as PSS components or PSS actions. They can be run on and interact with a UVM testbench with the help of *ruby_uvm* simulation runtime library. To interact with the UVM environment in the script, the UVM functions such as running a sequence or programing a register with RAL are exported to C side through DPI, which are eventually wrapped as ruby classes. For primitive actions of PSS execution, the *ruby_uvm* will serve any exec calls when the action is triggered. At the beginning of the simulation, the *ruby_uvm* runtime creates ruby interpreter and loads the ruby script converted from PSS with the test intents. By running the ruby script, the test scenario will be simulated according to the test intents specified in PSS.

Figure 1 demonstrates the flow of how ruby bridges PSS and UVM. When simulation starts, the ruby interpreter is created inside a *pthread* run on the simulator. In *ruby_uvm*, the basic functions from commonly-used UVM classes are exported to C through DPI, and these DPI functions are wrapped into ruby proxy class library. With this UVM-ruby library, the functionalities of UVM such as *factory*, *config-db*, *sequence* execution and *RAL* operations all become available to the ruby script. The PSS translator converts the PSS script into ruby, which becomes a test case ready to run on the UVM testbench enhanced by *ruby_uvm*. In this paper, the PSS translator and object model are introduced in details in section I, and the implemented of *ruby_uvm* is depicted in section II.

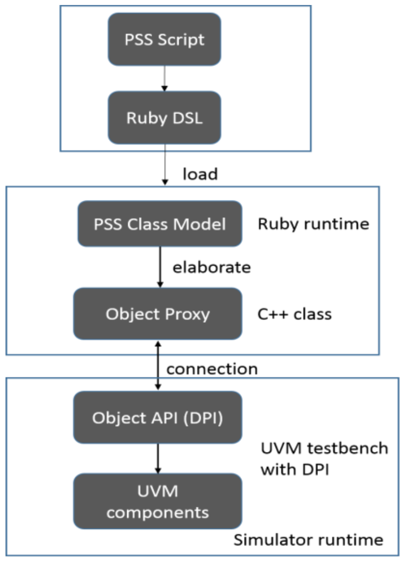

Figure 1. Diagram of pss_uvm work flow

I. DESIGN OF PSS TO RUBY SCRIPT TRANSLATOR

A. PSS parser and generator

The PSS describes a declarative Domain-Specific Language (DSL), which embraces the core concepts from Object-Oriented Programming (OOP) languages, Hardware-Verification Languages (HVL), and behavioral modeling languages. PSS features native constructs for system notions, such as data/control flow, concurrency and synchronization, resource requirements, states and transitions. Its lexical and syntactic conventions come from the C/C++ family, which are included in PSS specification. On the other hand, for its constraint and coverage capability, it uses SystemVerilog as a reference.

The PSS language parser (used in the translator/interpreter) is based on ruby treetop [3]. Treetop is a Ruby library that allows users to create parsers easily by describing them using a Parsing Expression Grammar (PEG). PEG is introduced by Bryan Ford [4], which is a type of analytic formal grammar, and it describes a formal language in terms of a set of rules for recognizing patterns in the language. In general, PEGs are a simple but powerful generalization of regular expressions that are easier to work with than the *LALR* or *LR-1* grammars from traditional parser generators. There's no need for a tokenization phase, and look ahead assertions can be used for a limited degree of context-sensitivity. With these natures of PEG, it is well-suited for parsing computer languages as it is simple to write and easy to maintain.

The current PSS treetop grammar is derived from the original *BNF* grammar included in the PSS specification, and the entire grammar is included in a signal file with 1000 lines supporting the full PSS grammar parsing. The codes below are examples of some PSS *activity* statement in PSS grammar. Each rule in treetop grammar comes with two parts: one is the regular expression that defines what this entity looks like, and the other is the type of syntax node it should be turned into. Treetop matches up the name encapsulated in <> with the custom syntax nodes defined as ruby class. The rules in Treetop are written using a superset of regular expressions.

```
     rule activity_labeled_stmt
       (identifier ':')? activity_stmt <ActivityLabeledStmt>
     end
     rule activity_repeat_stmt
       ('repeat' 'while' '(' expression ')' activity_sequence_block_stmt) /
       ('repeat' '(' (identifier ':')? expression ')' activity_sequence_block_stmt) /
       ('repeat' activity_sequence_block_stmt ('while' '(' expression ')')?) ';' s <ActivityRepeatStmt>
     end
```

Table 1. Treetop format of PSS grammar

When parsing a PSS script, an Abstract Syntax Trees (AST) is created and traversed, along with intermediate optimization and code generation steps are performed on it. The current implementation doesn't include much of optimization except skipping extraneous nodes that we don’t need or care. The custom syntax node classes can return a symbolic expression(*S-expr)* [5], as a simpler representation of themselves using simple nested sets of arrays and native Ruby data-types. S-expr is selected as intermediate AST format for further handling, taking advantage of that Ruby provides decent library to handle S-expr such as filters and convers. The AST node class is carefully tuned to create the S-expr with high compatibility to those available ruby language tools. Each complex node delegates the job for producing S-expr down to its children, and each primitive child node (e.g. StringLiteral, Identifier) is responsibly to how to return a simple version of itself. The current implementation of PSS parser supports all examples included in PSS specification. An example is shown as the following,

```
   resource struct cpu_core_s {};
   component dma_c {
     resource struct channel_s {};
     pool[2] channel_s channels;
     bind channels *;
     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};
     action par_dma_xfers {
       activity {
         parallel {
           dma_c::transfer xfer_a;
           dma_c::transfer xfer_b;
           constraint xfer_a.comp != xfer_b.comp;
           constraint xfer_a.chan.instance_id == xfer_b.chan.instance_id;
           constraint xfer_a.core.instance_id == xfer_b.core.instance_id;
         };
       };
     };
   };
```

Table 2. PSS script example

The generated S-Exp table of the example PSS script is shown below,

```
   s(:block,
    s(:call, nil, :resource, s(:lit, :cpu_core_s)),
    s(:iter,
     s(:call, nil, :component, s(:lit, :dma_c)),  0,
     s(:block,
      s(:call, nil, :resource, s(:lit, :channel_s)),
      s(:call, nil, :pool, s(:lit, :channels), s(:lit, :channel_s), s(:lit, 2)),
      s(:call, nil, :bind, s(:lit, :channels), s(:str, "*")),
      …
```

Table 3. Parser intermediate format of PSS in S-expr

After the S-expr is successfully returned with the root node, the *ruby2ruby* [6] tool can be used to convert the S-expr to valid ruby script. *ruby2ruby* provides a means of generating pure ruby code easily from *RubyParser* compatible with S-expr. In the design of PSS to ruby translator, the S-expr code generator from the translator is carefully designed to keep the result acceptable by ruby2ruby, in order that it can leverage this tool to translate the parser output to valid ruby code. The example of the final generated ruby code is shown as below,

```
   resource(:cpu_core_s)
   component(:dma_c) do
     resource(:channel_s)
```

```
    pool(:channels, :channel_s, 2)
    bind(:channels, "*")
    action(:transfer) do
      lock(:channel_s, :chan)
      lock(:cpu_core_s, :core)
    end
  end
  component(:pss_top) do
    instance(:dma_c, [:dma0, :dma1])
    pool(:cpu, :cpu_core_s, 4)
    bind(:cpu[:dma0, :dma1])
    action(:par_dma_xfers) do
      activity(:a1) do
        parallel(:p1) do
          need_act(:dma_c, :transfer, :xfer_a)
          need_act(:dma_c, :transfer, :xfer_b)
          constraint("xfer_a.comp != xfer_b.comp;")
          constraint("xfer_a.chan.instance_id == xfer_b.chan.instance_id;")
          constraint("xfer_a.core.instance_id == xfer_b.core.instance_id;")
        end
      end
    end
  end
```

Table 4. Result from PSS to ruby translator

B. *PSS object model*

The generated ruby script from PSS script translation is based on PSS's ruby object model. Most of basic types defined in PSS specification are implemented as ruby classes such as components, actions and resources. The PSS object model in ruby provides API for global usage, which were referenced by the PSS translated ruby scripts.

The component class provides a mechanism to encapsulate and reuse elements of functionality. They are structural entities and instantiated under other components, through which the instances constitute a hierarchy (tree structure) from the root component called *pss_top*. The actions are implemented as classes instead functions of their parent component. Since PSS *actions* need to encapsulate the rules for their interaction with other actions, by means of an action depended by the completion of other actions, the dependency relationship among actions are maintained by an action manger. This action manager will schedule and call the associated ruby block maintained in action class once the dependences of the action are resolved. It also manages flow and resource objects in order to coordinate each components to work together, where the action can easily reference to flow object and resource object. In this setup, the flow object specifies the action's inputs and outputs, and the resource object specifies the action's resource claims. Furthermore, if an action is atomic, its implementation is supplied via an exec block. In the *pss_uvm* framework, the exec block is a ruby block invoking the underlining *ruby_uvm* simulation runtime library, which will be discussed in details in the next section.

II. DESIGN OF RUBY_UVM RUNTIME LIBRARY

Concrete testcases in *pss_uvm* framework can be either converted from PSS script or written manually in Ruby. They need to be run on a ruby interpreter with the support from a *ruby_uvm* library, the runtime library of *pss_uvm*. In details, the ruby interpreter is created on a Linux pthread spawned from the HDL simulator process for load and execute the Ruby scripts. On the other hand, the *ruby_uvm* library provides the functionality imported from UVM testbench as Ruby's extensions, so that UVM functionality could be used by the Ruby tests. This section will introduce how *ruby_uvm* library works.

A. *Create ruby interpreter running from simulation process*

In UVM framework, a hierarchy of components are simulated under the coordination from simulation phases. An *uvm_test* is the top level simulation component which creates the whole uvm simulation environment and connects to DUT in the simulation build and connect phases. By the time when simulation reaches the *run_phase*, the DUT and UVM simulation objects have been elaborated and internal runtime models have been built inside the simulator, and the simulator starts to proceed the simulation actions with scheduling the runtime model and advancing the simulation time. *Uvm_ruby* is a Systemverilog DPI library, which communicates with the simulation run on the HDL simulator, as well as, support the Ruby script with the available UVM functionality. When DPI *uvmc_init_ruby* is called, a ruby interpreter will be created and runs as a part embedded in the simulator. The control of the simulation can be transferred to the ruby interpreter, so that the interpreter can perform evaluation of the ruby program loaded. *uvmc.rb*, the very first Ruby program loaded, does the bootstrap and loads in *ruby_uvm's* extension library. Then it move on to evaluate the user-defined Ruby test script. Through the ruby test, users can apply stimulus to the DUT, inspect the DUT object or control verification environment being simulated.

```
//create ruby interperter
static void *init_ruby(void *)
{
     ruby_init();
     ruby_init_loadpath();
     char script_file[1024];
     //specify the default ruby file to load after embedded ruby interpreter
      sprintf(script_file, "%s%s", getenv("ANCHOR"), "rb/uvmc.rb");
     std::cout << "load script file:" << script_file << std::endl;
     rb_require(script_file);
     return NULL;
}
void uvmc_run_ruby()
{
     pthread_t init_ruby_pthread;
     pthread_create(&init_ruby_pthread, NULL, init_ruby, NULL);
     pthread_join(init_ruby_pthread, NULL);
     return;
}
```

Table 5. Create ruby interpreter from Linux pthread

Figure 2 shows the internal structure of the *ruby_uvm* DPI library. Once the simulation control is handed over from UVM environment to ruby test, the ruby test can use any valid ruby syntax supported by the ruby interpreter for pre-defined simulation actions, for example, loading in extra ruby library. Through this, it can use the functionality provided by ruby extension from *ruby_uvm* to composite the content of the simulation testcase. In the *pss_uvm* framework, the ruby test is an exerciser layer of the UVM environment and it also supports interactively control the UVM simulation environment, allowing users to modify and inject actions in the middle of the simulation process.

The base testbench is still implemented in native UVM code, where the common sequence should be already developed and validated in native UVM testbench. Meanwhile, DPI and C++ ruby wrappers are compiled as part of *ruby_uvm* library. The integration comes into effect when the ruby interpreter is created first and the *ruby_uvm* library is get loaded as a ruby extension at the beginning of the simulation. While a ruby function is invoked which calls UVM extension function, the corresponding C++ function in the *ruby_uvm* library is triggered, and SystemVerilog DPI will be called afterworth. With this *ruby_uvm* library, the functionalities of UVM such as *factory*, *uvm_config_db*, *sequence* execution and *RAL* operations are all ready to use in the ruby scripts, and hence the test case can be developed in ruby instead of Systemverilog. The block diagram of *ruby_uvm* based simulation is shown in below figure.

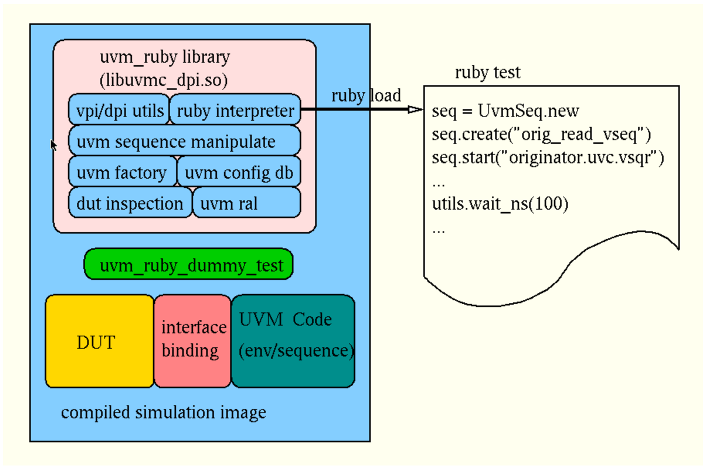

Figure 2. Block diagram of *ruby_uvm* class library

B. *Expose the UVM functionality into ruby class library*

A C++/ruby cross language library called “*ruby rice*” [7] is used for wrapping the DPI function in C into ruby classes and functions. Ruby rice makes writing ruby extensions with C++ easier, and fits well in our *pss_uvm* framework when creating ruby extensions from DPIs, as well as, wrapping the UVM functionality.

1. *Use UVM factory from ruby*

Below codes show how the UVM factory mechanism is supported in ruby, so that the ruby class can create various UVM classes such as an *uvm_sequence*. The DPI calls are using prefix “UVMC_”, for example, `UVMC_debug_factory_create`. They are placed into separate SystemVerilog package when exposing the uvm factory functionality to the C side. Then with the available DPIs, ruby rice class can be created to generate the ruby classes.

```
   extern "C" {
        ...
        void UVMC_set_factory_type_override(const char* original_type_name,
                                             const char* override_type_name,
                                             bool replace=1);
        void UVMC_debug_factory_create(const char* requested_type,
                                        const char* context="");
   }
   class UvmFactory {
        ...
        void set_factory_type_override(const char* original_type_name,
                                       const char* override_type_name,
                                       bool replace=1) {
                UVMC_set_factory_type_override(original_type_name, override_type_name, replace);
        }
        void debug_factory_create(const char* requested_type, const char* context="") {
                UVMC_debug_factory_create(requested_type, context);
        }
   };
   //Define ruby class UvmFactory
   define_class<UvmFactory>("UvmFactory")
                .define_constructor(Constructor<UvmFactory>())
                .define_method("print", &UvmFactory::print_factory);
```

Table 6. ruby_uvm implementation of factory API

2. Access Register through RAL

To support register access API, a group of UVM RAL extension classes has been developed to provide register access with "*access register by name*" style. The arguments to those functions are heretical reference to “register block”, “register name” and “field name”. By leveraging this style in the API, there is no need to model a register on ruby side, which saves the repetitive modeling work and simplifies the design of supporting library on ruby side. Ruby rice is used to create an UvmReg class wrapping around the exported uvm reg model functions using string type only, for example, a register names as function argument. From the ruby test script, user can create Uvm Ral transaction by specifying the register or field name without referencing RAL directly. The code example below is the ruby rice binding definition and the C++ class of UvmReg. The UvmReg references the UVM exported DPI functions, and provides member functional call on the ruby side. After ruby interpreter loads the compiled extension library, the UvmReg class will becomes available in the ruby program. The write or read to the ruby object UvmReg will result in the RAL access call to the register specified by the register hierarchical name.

```
class UvmReg {
public:
     UvmReg() {}
     void setup(string context, string blkname) {
         UVMC_reg_setup(context.c_str(), blkname.c_str());
     }
  void write(string regname, unsigned int value) {
         UVMC_reg_write(regname.c_str(), value);
     }
  unsigned int read(string regname) {
         unsigned int value;
         UVMC_reg_read(regname.c_str(), &value);
         return value;
     }
};
define_class<UvmReg>("UvmReg")
      .define_constructor(Constructor<UvmReg>())
      .define_method("setup", &UvmReg::setup)
      .define_method("write", &UvmReg::write)
      .define_method("read", &UvmReg::read);
```

Table 7. *ruby_uvm* implementation of register API

3. Execute UVM Sequence

Transaction class in a UVM testbench is defined as a derived class from *uvm_object*. Unlike modern script languages, SystemVerilog do not support language inspection. It is not easy to setup manipulation the object members without directly reference to the member variable. However there are still various methods to modify the class instances of member variables in run time. For example, the *ruby_uvm* supports randomization and *set_int/string_local* to change the instances of member variables in run time. Class UvmSeq is defined in ruby side which is corresponding to *uvm_sequence* from UVM. In the example below, *UVMC_sequence_create* is a task to create a UVM sequence using UVM factory mechanism with the specified type name. And *UVMC_sequence_start* is a task to start a sequence on a sequencer specified by its hierarchical name. These functions/tasks are exposed to C side through SystemVerilog DPI, and then wrapped into several C++ UVMSeq classes with member functions to create and start UVM sequence. Now with the help of the ruby rice, the C++ classes are further wrapped into ruby classes which can be used in the ruby programming.

```
seq = UvmSeq.new
seq.create "seq_type", "seq_name"
seq.set_int_local("field_name", num)
seq.randomize
seq.set_int_local("field_name", num2)
seq.send "sequencer"
```

Table 8. Short code segment of a ruby_script

In the *pss_uvm* simulation framework, ruby test can run interactively under the control of a ruby debugger or a user program. When the user defines an UvmSeq instance, the member variable can be setup using randomization or *set_int/string_local* multiple times until the class instance's values are proper. In SystemVerilog by using the *randomize() with* construct, users can declare in-line constraints at the point where the *randomize()* method is called. And the following the constraint blocks will define all of the required constraint types and forms as otherwise be declared in a class. However, in current implementation of *ruby_uvm* library, there is still no corresponding facility on ruby side yet. In the *pss_uvm* framework, all the constraints needs to be declared first inside the System Verilog class, then the ruby side can call the class member function to dynamically control a present constraint or setting the mode through *constraint_mode*. After the *UvmSeq* instance is defined, the member variables value can be printed to confirm the proper setup for debug purpose.

4. Concurrent Process Execution

On Systemverilog side, the UVM Sequences are created and run in the *run_phase* where simulation time is consumed. Correspondingly, in *ruby_uvm* library, the common sequence can be created using *uvm_factory* mechanism which is available to ruby script. They can be started on specific uvm sequencers. From ruby side, the start function call has a parameter which specifies the hierarchical name of the uvm sequencer. After a sequence has been created in ruby script, the sequence and corresponding sequencer pair is pushed into the sequence group, as a queue structure in System Verilog. There is a helper task called *sequence_execute* in Systemverilog and associated data tables to record all sequences and sequencers need to be used in ruby side. When a sequence has been triggered in ruby, the task *sequence_execute* will first obtain the sequence handler in the sequence table, and then launch the sequence on the corresponding sequencer. Once the registered sequence is finished, the ruby function (seq.run) will return. If all content of ruby tests has been completed, the ruby interpreter will end the job and release the simulation control. Furthermore, it is possible to execute multiple UVM sequences in parallel. In a ruby test, a number of uvm sequence can be created and launched altogether. The SystemVerilog helper task called *sequence_group_execute* is designed to serve this purpose. In addition, *Uvm_ruby* can use fork-join construct for controlling the sequences running in parallel. Once the ruby starts running the sequence group, the helper task in SystemVerilog will be kicked off. The below example is the pseudo code of this Verilog task,

```
fork
  begin : isolating_thread
    for(int index=0;index<num_of_seq;index++)begin : for_loop
      fork
      automatic int idx=index;
        begin
            `uvm_do_on(p_sequence_queue[idx],p_sequencer_queue[idx]);
        end
      join_none;
    end : for_loop
  wait fork;
  end : isolating_thread
join
```

Table 9. Concurrent process creation in Systemverilog

fork/join blocks the execution control until all direct child threads within it are completed, and the *wait fork* is blocking and waiting for all grandchild threads complete. In the code example above, there is only one child thread in the outer fork/join, the *isolating_thread begin/end* block, which will wait for the grandchildren threads spawned by the fork/join_none inside the '*for'* loop. The *isolating_thread* creates thread layer that guarantees only blocking the wait for the child threads of itself. It is also possible to create other variant to mimic the '*join all*' and '*join any*' constructs.

The majority interaction between the native UVM testbench and the ruby interpreter are through the ruby function bound to the Systemverilog DPI. There are several helper data structures on UVM side by which the ruby script can identify the components in Systemverilog easily. For example, all the *uvm_object* created by ruby are stored in Systemverilog as a hash table indexed by object names, in this way, a variable in ruby and a class instantiated in Systemverilog can be associated. The ruby script can leverage the facility supported by native UVM to inspect and manipulate the Systemverilog object in run time. For example, the user can print the UVM class content in a transaction, make modification in the fields or randomize them, and then send the modified transaction to uvm sequencer to execute. By binding ruby VPI to print out the DUT signal value, the user can inspect the DUT behavior while debugging the ruby test case. Besides the stimulus side, the UVM environment needs the analysis components such as uvm checkers and scoreboards to check the design behaviors, and these analysis components will work as they are in a ruby enabled UVM testbench. After ruby interpreter has been created in *uvm_test run_phase*, the simulation is controlled by this ruby interpreter, when the ruby test evaluated in the ruby interpreter will take over the simulation. All the DPI functions which have been exposed to the ruby side as ruby function calls can be used in the ruby tests. The ruby test can initialize and spawn transactions to DUT or to monitor the state of it. If the DPI called consumes simulation time, the ruby interpreter will block the simulation until the task finishes the execution. After all ruby expressions have been evaluated, the ruby interpreter will shut down and exit. And the simulation will be handed over back to the UVM side, where *uvm_test.run_phase* will take charge of the remaining simulation.

III. CONLUSION

The adoption of PSS needs tools to interpret PSS DSL, create scenarios or automate the testcase generation. In *pss_uvm* framework, tools has been created including PSS to ruby language translator, and the corresponding runtime library like *ruby_uvm*. With the framework and leveraging the tool set, the generated ruby test script could be run directly on a UVM based simulation environment. The projects is still in its early phase, without full PSS grammar supported (coverage is not support and only limited support of randomization), but it provides a prototype to validate the idea and prove the feasibility.

REFERENCES

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

[2] Accellera System Initiative. "Standard Universal Verification Methodology (UVM)."

[3] Treetop, a Ruby-based parsing DSL based on parsing expression grammars. http://treetop.rubyforge.org.

[4] Ronald L. Rivest, "S-Expressions, Network Working Group, Internet Draft, Expires November 4, 1997", May 1997

[5] Bryan Ford. "Packrat parsing: Simple, powerful, lazy, linear time". In Proceedings of the 2002 International Conference on Functional Programming, Oct 2002.

[6] Giles Bowkett. "Code Generation: The Safety Scissors of Meta programming". In Proceedings of Event. Emerging Tech 2008, Mar 26, 2008.

[7] Rice, Ruby Interface for C++ Extensions. rice.rubyforge.org