UVM Random Stability

Don’t leave it to chance

Avidan Efody
Mentor Graphics, Corp.
10 Aba Eban Blvd.
Herzilya 46120, Israel
avidan_efody@mentor.com

Abstract— **“Random stability” is a powerful tool that Hardware Verification Languages (HVLs) put at our disposal, allowing us to repeat specific parts of random simulations, even after significant changes to the Device Under Test (DUT) and testbench. Unfortunately, due to lack of awareness and proper planning users are often painfully reminded of this concept only when it breaks down unexpectedly, in which case it can make simulation results appear non-consistent or non-intuitive, and considerably slow down debugging.**

In this paper, we aim to demystify random stability in SystemVerilog and the Universal Verification Methodology (UVM) and show how it can easily be tamed to our benefit. First we will define the concept of random stability/instability and take a quick look at the benefits of the first and at the problems associated with the latter. We will then give an overview of the default random stability behavior of SystemVerilog and the API that can be used to customize it. Simple examples will be used to clarify why the default SystemVerilog behavior is usually not sufficient when working with UVM or any other advanced verification methodology, and to explain why UVM customizes this behavior the way it does.

Although UVM can be used to create a well planned random stable testbench, there are quite a few areas where clear coding guidelines must be followed to achieve that goal. Diving deeper into the technical details of UVM’s random stability layer, we list the loopholes that exist and how they could be avoided. In the case of sequences and sequence items we also provide some UVM code that wraps around the API and helps users keep out of the problematic areas.

Finally we show how the isolation of the random parts of a random stable testbench can be used for individual seeding. Individual seeding allows users to run many simulations with some random parts kept constant, and some randomly changing. It makes is possible to quickly test some complex modes that would be hard to describe from a test, or to generate similar but not identical scenarios to a scenario that exposed a bug. Here again, we provide some simple UVM code to help users get started.

Keywords- random stability, SystemVerilog, UVM

I. INTRODUCTION

A. *What is Random Stability?*

In SystemVerilog random stability can be defined as the resistance of random results to code changes. It is not, as some mistakenly think, the ability to repeat the same random sequence twice given identical code (on identical software version, identical OS, etc). The first is dependent on the structure of the user code and the usage it makes of SystemVerilog random stability features, while the second depends on the vendor’s simulator code, and is out of the scope of this paper.

Any SystemVerilog code that randomizes something is random stable and instable to some degree. It is always possible to change it in a way that won’t affect random results (say by adding a variable declaration), or in a way that will (say by adding an additional *$urandom()* or *randomize()* at the right place). However, in the case of “random stable testbench” (i.e. well, planned testbench from the “random stability” point of view), the changes that could affect random results are easily traceable back to a limited area in the code, whereas in the case “random instable testbench” (i.e. badly planned testbench from the “random stability” point of view), random results could be affected by code changes practically anywhere.

B. *Why is Random Stability Important?*

1) *Intuitive results*

With a random instable testbench small modifications in testbench code might result in big simulation differences that do not match what the user expects. To take a simple example, adding one random transaction to the stimuli, might trigger a change of all random transactions from that point on.

With random stable testbenches, what you see is what you get. Adding a transaction in the middle of stimuli would make the simulator generate identical transactions up to the point where the transaction is inserted, and after the point where the transaction is inserted. The only difference the user would observe is, as expected, the additional transaction in between.

2) *Consistent results*

Storing a test/seed pair as means of rerunning a specific case at some later stage is never guaranteed to work with constrained-random testbenches if modifications to DUT or testbench code take place. Unfortunately the alternative – writing coverage for the specific case and running the entire regression to find a new seed/pair each time – is so time consuming that users do often work with test/seed pairs. With random stable testbenches test/seed pairs are more likely to give the same results, and, when not, it should be easy to understand why. With non-random stable testbenches the generated scenario might change so radically that users will doubt their earlier observations. Often this causes a bug to "disappear", only to be found later on at another regression run.

3) Replicating bugs

In today's complex verification environments debugging is rarely a single person's task. A typical case can involve a verification engineer, an integration engineer, and a few owners of specific IPs. In some cases, they could all work in the single workspace where a specific bug was found. In many others, they would need to replicate it in a private workspace with their own private settings.

As defined above, in a random instable testbench the results of randomization can be dependent on code differences in a very wide perimeter from where actual randomization occurs. The chances that two different code bases would show a similar behavior are small. Therefore users are often forced to create fully identical copies, including private modifications in order to reproduce a bug. As almost every user who worked in a large environment knows, the effort spent on this task often outweighs the debugging effort that follows it.

4) Testing bug fixes

Once a bug has been fixed it often makes sense to test it under similar conditions to those in which the bug has been exposed. As we will see below, a random stable can be easily written in a way that would allow users to freeze the random state in some areas, and allow other areas to change, with or without additional constraints. This might help in creating a pin-pointed test, designed to check the robustness of a specific bug fix.

II. Understanding SystemVerilog Random Stability

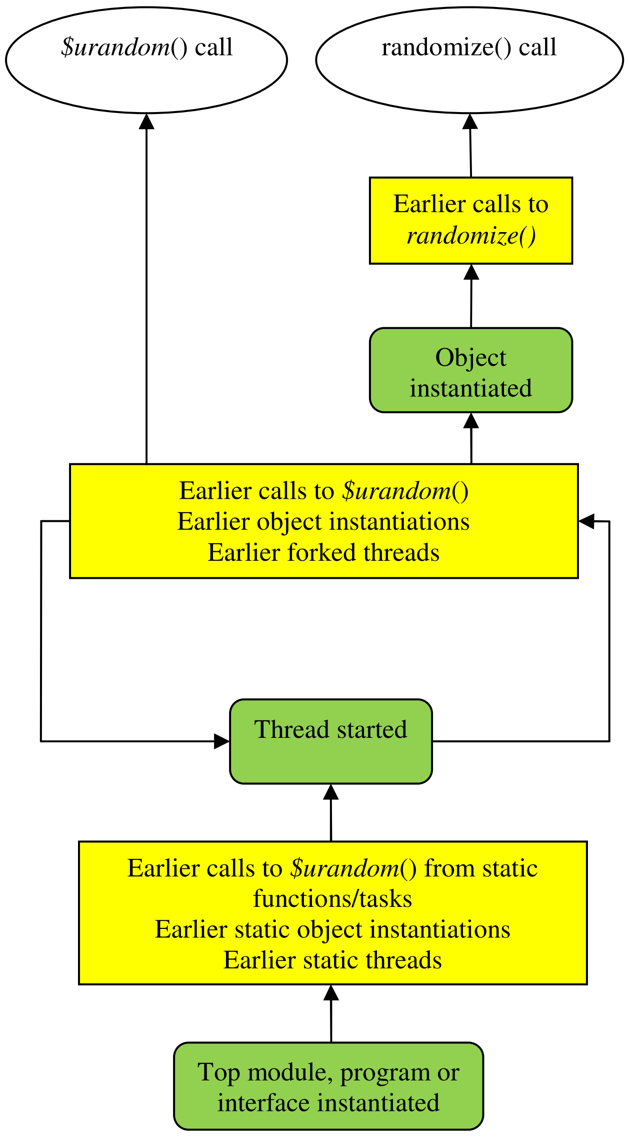
There are several methods of creating random values in SystemVerilog, but either way the values generated depend, at least to some extent, on the location of the randomization instruction in the execution flow. By default, the random values will depend on the absolute location of the randomization instruction. Relative location dependency with regards to a specific known point in the code can be achieved through the **manual seeding** feature of the SystemVerilog API (also referred to as **reseeding** throughout this text). The save and restore capability of the API allows for execution path modifications without affecting the results of any subsequent random instructions. In this section we will first understand the default behavior, then look at the ways to manipulate it via the SystemVerilog API [1]. Since UVM makes extensive use of this API, this is a first required step in order to be able to make the most of UVM random stability support.

A. Absolute path dependency

The element responsible for generating random values in SystemVerilog is called Random Number Generator, abbreviated RNG. Each thread, package, module instance, program instance, interface instance, or class instance has a built-in RNG. Thread, module, program, interface and package RNGs are used to select random values for *$urandom()*, *$urandom_range()*, *std::randomize()*, *randsequence*, *randcase*, and *shuffle()* and to initialize the RNGs of child threads or child class instances. A class instance RNG is used exclusively to select the values returned by the class's predefined *randomize()* method. All examples and text below refer to either *$urandom()* or *randomize()*. The remaining randomization instructions (i.e. *$urandom_range()*, *std::randomize()*, *randsequence*, *randcase*, *shuffle()*) behave in the same way as *$urandom()* so every occurrence of *$urandom()* in the text that follows should be read as referring to either of those.

Whenever an RNG is used either for selecting a random value or for initializing another RNG, it will "change state" so that the next number or set of numbers it generates is different. Therefore, the value a specific randomization call returns, depends on the number of times the RNG has been used and on its initialization. The RNG's initialization, in turn, depends on the number of times its parent RNG has been used and on the parent RNG initialization. The topmost RNG is always a module, program, interface or package RNG, and all of these RNGs are initialized to the same value, which is chosen by the simulator according to the simulation seed.

Fig. 1 illustrates the paragraph above and shows how the values returned by randomization calls are affected by the execution of earlier code. The value a given *$urandom()*, returns, is determined by the RNG of the thread executing it. Since the point where this thread was initialized by its parent thread, its RNG changed state for every earlier call it made to *$urandom()*, for every earlier object it instantiated, and for every earlier child thread it forked. Its initialization value was determined by the RNG state of its parent thread at the moment when it was forked, and this RNG state depended once again on any earlier use of the same three types of instructions : calls to *$urandom()*, object instantiations, and forks. This goes all the way back to the static thread that started the whole tree. In the vast majority of cases this static thread is an *always* or *initial* block whose RNG was initialized by the RNG of the module, program or interface that contains it. That RNG is initialized by the simulator to some seed-dependent value, then changes state for every static call to *$urandom()*, every static instance created and every static thread that is forked.



$urandom() call
randomize() call
Earlier calls to randomize()
Object instantiated
Earlier calls to $urandom()
Earlier object instantiations
Earlier forked threads
Thread started
Earlier calls to $urandom() from static functions/tasks
Earlier static object instantiations
Earlier static threads
Top module, program or interface instantiated

Figure 1. Execution path influence on SystemVerilog randomization methods. Green rounded squares reprenet new RNG initialization. Yellow squares show RNG state modifying instructions.

For a given *randomize()* call the process is essentially the same up to the point where the object is allocated. Once the object is allocated it gets its own RNG which, unlike package, module, program, interface or thread RNGs, changes state only when *randomize()* is called. Therefore, from instantiation point onwards the only instructions that affect the results of a given *randomize()* call, are earlier *randomize()* calls[1].

The code below shows an example of the instructions that will affect the results of a given *$urandom()* and *randomize()* calls. Lines colored in red (with the comment "both") affect both the *$urandom()* and *randomize()* calls in bold blue (with the commend "endpoint"). Lines colored in purple (with the comment "*$urandom()* only") affect only the *$urandom()* call. Lines colored in green (with the comment "*randomize()* only"), only the *randomize()* call. Everything that's in black doesn't affect either.

```
package my_pkg;
  class some_class;
  endclass

  class my_class;
    rand int x;
  endclass
endpackage

module my_top();
  import my_pkg::*;

  my_class cls = new();[2] //->both

  int j = $urandom();[3] //->both

  initial;

  initial begin
    automatic int i;
    some_class sc;

    fork
      some_thread(); //->both
    join_none

    sc = new();[4] //->both
    i = $urandom(); //->both

    fork
      my_thread(); //->both
    join_none
  end

  task some_thread();
    #1;
  endtask


  task my_thread();
    automatic int i;
    some_class sc;
    my_class mc;

    fork
      some_thread(); //->both
    join_none

    mc = new(); //->both
    sc = new(); //->$urandom() only
    i = $urandom(); //->$urandom() only
```

[1] Note that an object *randomize()* might be called from several threads. However, this is rarely done, and in any case, a bad coding practice.

[2] Although this line should affect both the *$urandom*() and randomize() at the bottom, in some simulators it doesn't.

3 Same for this line

[4] With some simulators this line doesn't affect results unless some_class has some rand variables in it

```
    mc.randomize(); //->randomize() only

    fork
      some_thread();//->$urandom only
    join_none

    mc.randomize();//->randomize() endpoint
    $display("randomize result is %d", mc.x);
    i = $urandom();//->$urandom() endpoint
    $display("urandom result is %d", i);
  endtask
endmodule
```

B. *Relative Path Dependency*

Dependency on absolute execution path will make random results extremely sensitive to code changes even in a small size project. By manually setting an RNG to a specific known state, the execution path up to a certain point becomes a don't care. This is referred to as "manual seeding" and makes any subsequent random results depend only on the relative execution path from the manual seeding point onwards. The code below shows how this is done for a thread or an object.

```
class my_class;

  rand int y;

  task not_random();
    process p = process::self();
    p.srandom(1); // thread reseeding
    $display("constant result %d",
$urandom());

    this.srandom(2); // object reseeding
    randomize();
    $display("and another one %d", y);
  endtask

endclass
```

In a constrained random testbench this code doesn't make a lot of sense because it is too stable: it will make randomization results constant in every simulation because they are no longer dependent on the simulation seed. To prevent this, the argument to *srandom()* is usually a function of the simulation seed, preferably one that is evenly distributed to prevent constant repetition of same value.

C. *Saving and Restoring State*

In some cases it is required to add code that won't affect any subsequent random results, i.e. RNG state. To achieve this, the SystemVerilog API provides means for saving and restoring and RNG state. The code below shows how this is used to protect an additional object instantiation from changing a *$urandom*() result later on.

```
class class_a;
endclass

class class_b;

  task main();
    class_a a;
    string rand_state;
    process p = process::self();

    rand_state = p.get_randstate();

    $display("first random value %d",
$urandom());

    a = new();

    p.set_randstate(rand_state);

    $display("and a second identical one %d",
$urandom());
  endtask
endclass
```

III. Understanding UVM Random Stability

In a typical UVM testbench, the following areas are random to some extent:

1. Random Bus Functional Model (BFM) parameters such as delays (when not part of the transaction)
2. Random configuration parameters (for example register values)
3. Sequences and transactions

For 1) and 2) randomization is usually done inside a static *uvm_component*. For 3) it is usually done inside a dynamic *uvm_sequence* or *uvm_sequence_item*. We will now look at the random stability mechanisms available in each, understand their limitations, and suggest the best ways to cope with those.

A. *uvm_component random stability*

1) The requirement

During a project life cycle some components are expected to be added or removed from a UVM hierarchy. Also, UVM testbenches are often highly configurable and components or clusters of components might be added or removed based on test configuration parameters. Users expect all of these changes not to affect the random values generated by specific components in the hierarchy for a given seed. This simplifies orientation in a simulation with a new configuration, since all the parts that were there before continue to behave in much the same way. It will also allow for quicker isolation of bugs.

In Fig. 2 For example, adding the component in red with additional instantiations, forks and randomizations, should not affect any randomization that takes place in the components in blue.

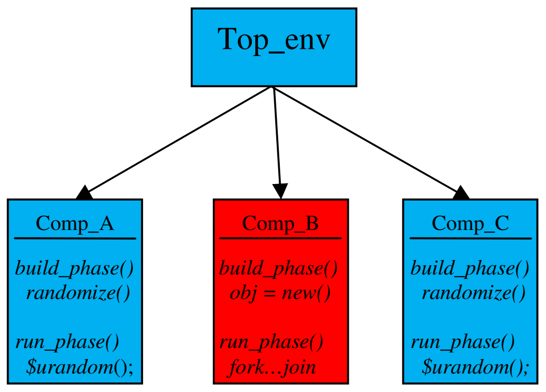


Figure 2. Adding the red component with instantiations, forks and randomizatios (not shown), should not change any random results in the blue components

2) *The solution*

We have seen earlier that by default, the values returned by various SystemVerilog randomization API commands are dependent on the absolute execution path. Looking at SystemVerilog verification methodologies such as UVM, OVM and VMM, it becomes apparent that relying on absolute execution path would make it impossible to comply with the requirement above. In these methodologies (and others) the phase that does the actual running, and therefore most of the randomization, is preceded by a testbench construction phase, in which most testbench elements are instantiated. Relying on the absolute execution path would mean that any additional instance created in the testbench construction phase would end up changing all random results in the run phase.

UVM addresses the problem by cutting the execution path into a multitude of slices, each of which protected by its own *srandom()* call. Any observed change in the random results generated, can then be easily traced back to a fairly limited area in the code. For *uvm_component*s the executing thread is manually seeded before each function or task phase, and the component itself is manually seeded after its creation [2]. This means that any change in the results returned by a *$urandom*() within a specific component must be due to an additional instantiation, fork, or *$urandom()* call within the same function or task phase and within that same component. Any change in the results returned by a *randomize()* call of a specific component must be due to some additional calls to *randomize()* of the same component. Inside these well defined borders, it should be easy to understand where changes are coming from.

As mentioned above, the seeds to different *srandom()* calls should be unique and evenly distributed, or else they might make a random testbench less random than users actually expect it to be. For example, if two instances of the same BFM are always manually seeded with the same value, they would always operate in sync, leaving other situations untested. In the case of *uvm_component*s this problem is relatively easily solved by using an integer value extracted from their unique full name as a parameter to *srandom()* (Calls to *srandom()* before a function or task phase simply append the phase name to the full name of the component). Note that this ties all random values generated by a component to its location in the UVM hierarchy, which makes sense, since users don't expect two instances in different testbench parts to produce the same results. It also means that if a testbench becomes a part of a bigger testbench (i.e. vertically reused), it can no longer be expected to produce the same random results.

3) *Limitations*

a) *Thread Stability in UVM-1.1 and Earlier*

Manual seeding prior to execution of function and task phases has been added to UVM only since UVM-1.1a, and isn't a part of earlier UVM (or OVM) versions. In its absence, some randomization instructions will fall back to the default SystemVerilog random stability. This is most probably an unwanted behavior. Fortunately, it can be easily prevented.

One case where this would happen is shown by the code example below. The value returned by *$urandom()* is dependent on the thread RNG and since with UVM-1.1 and earlier this RNG is not manually seeded prior to the execution of the thread, its state depends on the absolute execution path up to this point. Therefore additional components instantiated during the *build_phase()* construction phase by this component or elsewhere would influence it, making it unstable.

```
class my_bfm extends uvm_component;

  //…

  task run_phase(uvm_phase phase);
    int unsigned response_delay;

    //…

    response_delay = $urandom_range(0, 20);

    //…
  endtask
endclasss
```

To make this code stable with UVM-1.1 and earlier, just replace *$urandom*() calls with a call to *randomize()* (Note that this requires making the randomized variable a rand class member):

```
class my_bfm extends uvm_component;

  //…
  rand int unsigned response_delay;

  task run_phase(uvm_phase phase);

    //…

    randomize(response_delay) with
{(response_delay >= 0) && (response_delay <=
20);};

    //…
  endtask
endclasss
```

This would make the results depend on the component RNG, and since this one is manually seeded in UVM-1.1 (and OVM), it means that the random value returned would be affected only by other calls to *randomize()* of the same object.

The following code shows a similar case:

```
class rand_config extends uvm_object;
  `uvm_obejct_utils(rand_config)
  rand bit config_field1;
  rand int unsigned config_field2;
endclass

class some_env extends uvm_env;

  //…

  rand rand_config rand_config_i;

  function void build_phase(uvm_phase phase);
    rand_config_i =
rand_config::get_type::create("rand_config_i")
;
    assert(randomize(rand_config_i)); //1
    assert(rand_config_i.randomize()); //2
  endfunciton

  //…
endclass
```

Although the green line (with the comment "1") and the red line (with the comment "2") might appear equivalent, from a random stability point of view they are not: while the first *randomize()* call would select the object values based on the parent component RNG, the second *randomize()* call would select them based on the object's own RNG. Since objects don't have unique names and are not manually seeded during their creation, their RNG initialization depends on the RNG state of the thread that instantiated them. In a UVM-1.1a testbench, however, both methods would be stable enough, because the instantiating function phase is manually seeded prior to its execution. Unfortunately, with UVM-1.1 and earlier this is not the case, and when the second option is used, the values of the object will once again end up being chosen based on the absolute execution path.

B. uvm_sequence/uvm_sequence_item random stability

1) The requirement

Assume a user runs the blue sequence shown in the Fig. 3 below, then for some reason modifies it by adding the red sequence item.

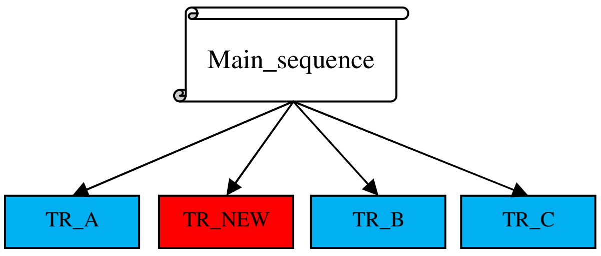


Figure 3. An additional sequence item is not expected to change items before or after it

What the user would expect after running is that TR_A and TR_B remain identical, and a new random sequence item is added between them. This would allow for quicker orientation in the results of the new test, and enable accurate fine-tuning of sequences in order to simulate corner cases. For example, a transaction that fills up a FIFO to a certain level could be added in the middle of an existing sequence to target a specific problematic area.

Looking at a slightly more general example in Fig. 4 users would expect that any of the red extensions to the blue sequence hierarchy would not modify the results. Original sequence stimuli should stay the same when new items and sequences are added anywhere in the hierarchy, or when other sequences are started on the sequencer in parallel.

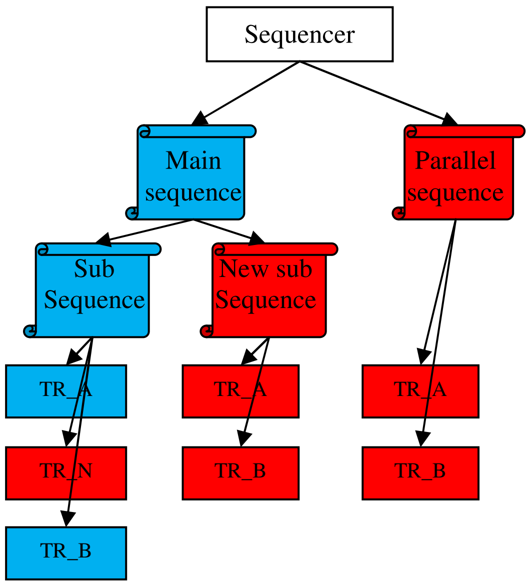


Figure 4. Sequence hierarchy modifications that are not expected to alter original results

2) The Solution

Although a component hierarchy stays the same throughout a simulation, while an entire sequence hierarchy could be created and destroyed multiple times, the random stability requirements in both cases are very similar: A change in hierarchy should not affect existing elements. Therefore, UVM tries to address this problem with a similar approach to the one taken with components. Sequences and sequence items are isolated from each other and from the rest of the testbench by manual seeding which is based on their full name. Since users do not expect identical sequences running on different sequencers to produce identical results (that would be too stable), UVM prefixes a sequence or sequence item full name with the full component path of the sequencer it is executed on. For example, TR_A in Fig. 3 about would be manually seeded based on the following name:

[Name of *uvm_sequencer* on which Main_sequence is running].Main_sequence.TR_A

3) *Limitations*

a) *Unique Naming not Enforced*

Although the whole concept of sequence random stability rests on the assumption that every sequence and sequence item are uniquely identifiable by their full name, UVM doesn't force sequences and sequence items to have a unique name, or even have a name at all. In the absence of unique names, manual seeding becomes meaningless, since al items sharing a name would be reseeded based on some sort of a counter to differentiate them from each other. Random stability as described in Fig. 3 or Fig 4. simply can't be achieved in this case, all the more so because users themselves can't identify transactions by any other means except for counting. It is very likely that in such situation, all random results within a specific sequence hierarchy would change due to an additional instantiation or an additional call to *$urandom()* in the thread that started the top most sequence.

We strongly advice users to assign sequences and sequence items with unique names. This is not only a pre-condition for random stability as shown in Fig. 3 and Fig. 4 above, but also makes debugging of long sequences of transactions much easier. As mentioned above, if transactions and sequences are not uniquely named, the only way to match them with, for example, whatever shows in the waveform viewer, is by counting. This is time consuming and error prone.

To enforce unique naming it is possible to extend the *uvm_sequence/uvm_sequence_item* and *uvm_sequencer* classes, so that they check if a name of a top level sequence, child sequence, or item has already been used prior to execution. A simple implementation of these extensions for UVM 1.1a is shown in appendix A.

b) *Sequence thread is not manually seeded*

In the section dedicated to components we have mentioned that UVM manually seeds every function and task phase before they're executed. This makes calls to randomization methods that are dependent on the thread RNG, insensitive to anything that occurs outside the specific function or task in which they are used. It also makes any objects created within the function or task more random stable, since their RNG initialization depends only on code inside that same function or task.

Unfortunately with UVM-1.1a neither the sequence tasks, nor its main thread which calls all of them are reseeded. Users wishing to avoid random stability issues can either make sure their code complies with the guidelines given in the uvm_components section for UVM-1.1 users. Or, they can derive their own base class from uvm_sequence, and implement manual seeding during the *pre_start()* phase. The code below, which could be placed in a sequence base class, shows how to do this.

```
  class reseed_seq#(type REQ =
uvm_sequence_item,type RSP = REQ)  extends
uvm_sequence#(REQ, RSP);

    function new(string name = "");
      super.new(name);
    endfunction: new

    virtual task pre_start();
      process proc = process::self();

       proc.srandom(uvm_create_random_seed("bo
       dy", get_full_name()));
    endtask
  endclass
```

c) *Reseeding timing doesn't match use model*

UVM lets users choose between two ways for creating, randomizing and running sequences – using `*uvm_do* macros, or using create, randomize and start. These ways are shown below, in red and blue corresponding:

```
class my_sub_sequence extends
  //…
  rand int unsigned num_of_items;
  //…
uvm_sequence#(my_item);
  //…
endclass

class my_sequence extends
uvm_sequence#(my_item);
  my_sub_sequence sub_sequence;

  task body();

    // using create/randomize/start
    sub_sequence =
my_sub_sequence::type_id::create("sub_sequence
");
    sub_sequence.randomize();
    sub_sequence.start(get_sequencer(),this);

    //using `uvm_do
    `uvm_do(sub_sequence)
  endtask
```

```
endclass
```

Both options are equivalent to a large extent, but not from a random stability point of view. While with the `*uvm_do* macro the sequence will be manually seeded before it is randomized, with the create/randomize/start method it will be manually seeded only after randomization. This means, for example, that when using create/randomize/start as shown above, an additional instantiation in the *body()* task of the parent sequence will modify the rand fields of the sub-sequence (i.e. *num_of_items* in the example above). Note that this is true only for *uvm_sequence*s that are executed using *start()*, and not for *uvm_sequence_item*s that are executed using *start_item()/finish_item()*.

Since using `*uvm_do* macros should in general be avoided [3], we recommend that users simply reseed sequences by themselves before they are randomized. Also, it should be noted that `*uvm_do* derivatives can't create the a top level sequences, which must always be executed using *start()*. Therefore using them would solve the problem only partially. The code below shows how to use the preferred create/randomize/start method so that manual seeding takes place before randomization. The solution is to insert a call to *set_item_context()* before randomization. This function initializes the sequencer and parent sequence fields for the sub sequence and allows full name calculation and reseeding.

```
class my_sequence extends
uvm_sequence#(my_item);
  my_sub_sequence sub_sequence;

  task body();

    // using create/randomize/start
    sub_sequence =
my_sub_sequence::type_id::create("sub_sequence
");
    sub_sequence.set_item_context(this,
get_sequencer());
    sub_sequence.randomize();
    sub_sequence.start(get_sequencer(),this);


  endtask
endclass
```

IV. Individual Seeding of Testbench Parts

With a well planned UVM random stable testbench, all components, sequences and sequence items are isolated from one another. This means that modifying the code in each of those would not affect the random results in others (unless these random results are directly dependent on some random field of the part modified, for example through constraints). It also means that modifying the random results in each of these would not affect the random results in others (with the same restrictions as above).

Decoupling random parts from each other allows them to be individually seeded. If a specific part is individually seeded, only the results within it would change, but everything else would be kept constant. As mentioned above, this can be useful, for example, for testing a bug fix. If the bug is related to several independent parameters and events that happen together by chance, keeping some of those constant while modifying others might reveal similar bugs, or find the weaknesses of the bug fix.

As part of the work on this paper we have created a UVM package that can be used to individually seed specific testbench elements from the command line. The code of this package is fully available in appendix B of this paper. It allows users to configure a few selected components as random domain roots (sequences as random domain roots are not supported, although this could be implemented easily). A random domain root can be seeded from the command line through a SystemVerilog plusarg, or generate its own random seed, which is based on the main simulation seed, but different from it. It then makes its seed available to all components lower in the hierarchy as a configuration parameter. The components lower in the hierarchy pick up the specific domain seed from the configuration table and reseed themselves.

Note that users of the package **are required to always call the super class phase functions and tasks, on the first line of their own implementation**. This is needed in order to reseed execution thread before the task or function are executed. UVM users normally do this anyhow for *build_phase()* and sometimes for other phases, so that should not add too much overhead.

The abridged example below shows how the package is used. An environment (*env*) made out of two component hierarchies that the user defines to be seeded individually: *root_comp1* and *root_comp2*. Each of these contains random fields and some sub components that call *$urandom()*. The user could keep each of these hierarchies constant, while changing the seed and random values of the other. For example (assuming the user runs the simulation with Mentor's Questa), running the following two commands:

```
vsim top –sv_seed random +SEED0=1 +SEED1=1
vsim top –sv_seed random +SEED0=1 +SEED1=2
```

Would keep all random values generated by the *root_comp1* hierarchy constant, while changing all values of generated by *root_comp2*.

```
  class sub_component extends
multi_seed_component;
```

```
    //…

    task run_phase(uvm_phase phase);
      super.run_phase(phase); // --> required
by multi-seed package
      $display("sub component %s random value
is %d", get_full_name(), $urandom());
    endtask
  endclass



  class root_component extends
multi_seed_component;

    rand int unsigned num_of_sub_components;
    constraint max_sub_comps_c {
num_of_sub_components < 5; };

    sub_component sub_comps[];

    //…

    function void build_phase(uvm_phase
phase);
      super.build_phase(phase); // -->
required by multi-seed package
      randomize();
      $display("root component %s generating
%d components", get_full_name(),
num_of_sub_components);

      sub_comps = new[num_of_sub_components];
      //…create sub components
      end
    endfunction
  endclass


  // instantiates two root components, each of
which is configured to be a random domain
  class env extends multi_seed_env;

    root_component root_comp1, root_comp2;

    function void build_phase(uvm_phase
phase);
      domain_root_config domain1, domain2;

      super.build_phase(phase);

      //…create components

      // define random domains
      domain1 = new();
      domain1.set_id(0);
      domain1.name = "root_comp1";

      domain2 = new();
      domain2.set_id(1);
      domain2.name = "root_comp2";

      // map them to components
      uvm_config_db#(domain_root_config)::set
(this, "root_comp1", "domain_root_config",
domain1);

      uvm_config_db#(domain_root_config)::set
      (this, "root_comp2",
      "domain_root_config", domain2);
    endfunction

  endclass
```

V. Acknowledgements

VI. References

[1] IEEE Standard for System Verilog- Unified Hardware, Design, Specification and Verificaction Language", IEEE std 1800-2009, 2009.

[2] UVM 1.1a Reference, www.uvmworld.org

[3] "Are OVM & UVM Macros Evil? A Cost-Benefit Analysis", Erickson, Adam, 2011

VII. APPENDIX A – UVM EXTENSION TO ENFORCE UNIQUE SEQUENCE/SEQUENCE ITEM NAMES

```
// *** This code is provided as an example only and without guarantee or any commitment to
enhancements/support ***

// This package extends UVM sequences to check for unique names
// Should be used with UVM-1.1a only!

package unique_seq_pkg;
  import uvm_pkg::*;
  `include "uvm_macros.svh"

  class unique_sequencer #(type REQ=uvm_sequence_item, RSP=REQ) extends uvm_sequencer #(REQ,
RSP);

    typedef unique_sequencer #( REQ , RSP) this_type;

    `uvm_component_param_utils(this_type)

    function new (string name, uvm_component parent=null);
      super.new(name, parent);
    endfunction


    // check for top sequences executed at sequencer
    string used_names[$];
  endclass

  class unique_sequence#(type REQ = uvm_sequence_item,type RSP = REQ) extends uvm_sequence#(REQ,
RSP);

    typedef unique_sequence #(REQ, RSP) this_type;

    string used_names[$];

    function new (string name = "uvm_sequence");
      super.new(name);
    endfunction

    // This task is copied almost as is from uvm_sequence_base
    // modifications:
    // 1. check for unique name before sequence starts

    virtual task start (uvm_sequencer_base sequencer,
                        uvm_sequence_base parent_sequence = null,
                        int this_priority = -1,
                        bit call_pre_post = 1);

      set_item_context(parent_sequence, sequencer);

      if (!(m_sequence_state inside {CREATED,STOPPED,FINISHED})) begin
        uvm_report_fatal("SEQ_NOT_DONE",
           {"Sequence ", get_full_name(), " already started"},UVM_NONE);
      end

      if (this_priority < -1) begin
        uvm_report_fatal("SEQPRI", $psprintf("Sequence %s start has illegal priority: %0d",
                                          get_full_name(),
                                          this_priority), UVM_NONE);
      end
      if (this_priority < 0) begin
```

```
          if (parent_sequence == null) this_priority = 100;
          else this_priority = parent_sequence.get_priority();
       end

       // Check that the response queue is empty from earlier runs
       clear_response_queue();

       set_priority(this_priority); // --> changed by unique_seq_pkg to avoid use of local
variable

       if (m_sequencer != null) begin
           if (m_parent_sequence == null) begin
             m_tr_handle = m_sequencer.begin_tr(this, get_name());
           end else begin
             m_tr_handle = m_sequencer.begin_child_tr(this, m_parent_sequence.m_tr_handle,
                                                          get_root_sequence_name());
           end
       end

       // Ensure that the sequence_id is intialized in case this sequence has been stopped
previously
       set_sequence_id(-1);
       // Remove all sqr_seq_ids
       m_sqr_seq_ids.delete();

       // Register the sequence with the sequencer if defined.
       if (m_sequencer != null) begin
         void'(m_sequencer.m_register_sequence(this));
       end

       check_unique_name(this);   // --- added by unique_package

       fork
         begin
           m_sequence_process = process::self();

           m_sequence_state = PRE_START;
           #0;
           pre_start();

           if (call_pre_post == 1) begin
             m_sequence_state = PRE_BODY;
             #0;
             pre_body();
           end

           if (parent_sequence != null) begin
             parent_sequence.pre_do(0);    // task
             parent_sequence.mid_do(this); // function
           end

           m_sequence_state = BODY;
           #0;
           body();

           m_sequence_state = ENDED;
           #0;

           if (parent_sequence != null) begin
             parent_sequence.post_do(this);
           end

           if (call_pre_post == 1) begin
             m_sequence_state = POST_BODY;
             #0;
             post_body();
           end

           m_sequence_state = POST_START;
           #0;
           post_start();

           m_sequence_state = FINISHED;
           #0;

         end
       join

       if (m_sequencer != null) begin
         m_sequencer.end_tr(this);
       end

       // Clean up any sequencer queues after exiting; if we
       // were forcibly stoped, this step has already taken place
       if (m_sequence_state != STOPPED) begin
         if (m_sequencer != null)
           m_sequencer.m_sequence_exiting(this);
       end

       #0; // allow stopped and finish waiters to resume

     endtask


     // this task is an exact copy of start_item from uvm_sequence_base
     // the only difference is that it checks that each item has a unique name on this sequence
     virtual task start_item (uvm_sequence_item item,
                              int set_priority = -1,
                              uvm_sequencer_base sequencer=null);
       uvm_sequence_base seq;

       if(item == null) begin
         uvm_report_fatal("NULLITM",
             {"attempting to start a null item from sequence '",
             get_full_name(), "'"}, UVM_NONE);
         return;
       end

       if($cast(seq, item)) begin
         uvm_report_fatal("SEQNOTITM",
             {"attempting to start a sequence using start_item() from sequence '",
             get_full_name(), "'. Use seq.start() instead."}, UVM_NONE);
         return;
       end

       if (sequencer == null)
           sequencer = item.get_sequencer();

       if(sequencer == null)
           sequencer = get_sequencer();

       if(sequencer == null) begin
           uvm_report_fatal("SEQ",{"neither the item's sequencer nor dedicated sequencer has been
supplied to start item in ",get_full_name()},UVM_NONE);
       return;
       end

       if (sequencer == null)
         sequencer = item.get_sequencer();
       if (sequencer == null) begin
           uvm_report_fatal("STRITM", "sequence_item has null sequencer", UVM_NONE);
       end

       item.set_item_context(this, sequencer);

       if (set_priority < 0)
         set_priority = get_priority();

       check_unique_name(item); // --> added by unique_package

       sequencer.wait_for_grant(this, set_priority);

       `ifndef UVM_DISABLE_AUTO_ITEM_RECORDING
         void'(sequencer.begin_child_tr(item, m_tr_handle, item.get_root_sequence_name()));
       `endif

       pre_do(1);

     endtask

     virtual function void check_unique_name(uvm_sequence_item seq_item);
       // check if name is already defined, if not add it to table, if yes give an error

       string name = seq_item.get_name();

       if (seq_item.m_parent_sequence == null)
         begin
           unique_sequencer#(REQ, RSP) unique_sqr;
           string result[$];

           if (!$cast(unique_sqr, m_sequencer))
             uvm_report_error("UNIQUE_SEQUENCR_NOT_USED", {"Trying to start the sequence ", name,
 " , which is derived from unique_sequence, on sequencer ", m_sequencer.get_full_name() ," which
 is not derived from unique_sequencer"}, UVM_LOW);

           result = unique_sqr.used_names.find_first with (item == name);

           if (result.size() > 0)
             uvm_report_error("TOP_SEQ_NOT_UNIQUE", {"Top level sequence by name of ", name, "
 already started on the sequencer ", unique_sqr.get_full_name()}, UVM_LOW);
           else
             unique_sqr.used_names[$+1] = name;
         end
       else
         begin
           unique_sequence#(REQ, RSP) parent_unique_sequence;
           string result[$];

           if (!$cast(parent_unique_sequence, seq_item.m_parent_sequence))
             uvm_report_error("UNIQUE_SEQUENCS_NOT_USED", {"Trying to start a sequence derived
 from unique_sequence from the sequence ", m_parent_sequence.get_full_name() , " which is not
 derived from unique_sequence"}, UVM_LOW);

           result = parent_unique_sequence.used_names.find_first with (item == name);

           if (result.size() > 0)
             uvm_report_error("SEQ_NOT_UNIQUE", {"Sequence by name of ", name, " already started
 by the sequence ", parent_unique_sequence.get_full_name()}, UVM_LOW);
           else
             parent_unique_sequence.used_names[$+1] = name;
         end
     endfunction
   endclass
 endpackage
```

VIII. Appendix B – UVM Extensions to allow individual seeding of various testbench parts

```
// *** This code is provided as an example only and without guarantee or any commitment to
enhancements/support ***


package multi_seed_pkg;
  import uvm_pkg::*;
  `include "uvm_macros.svh"

  `define SEED_NUM 10

  int seeds[`SEED_NUM] = get_seeds();

  typedef int seed_array[`SEED_NUM];

  function seed_array get_seeds();
    foreach(seeds[i])
      seeds[i] = $urandom();
  endfunction

  class domain_root_config extends uvm_object;
    `uvm_object_utils(domain_root_config)

     string name;
     local int unsigned id;
     static int unsigned used_ids[$];

     function new(string name = "");
       super.new(name);
     endfunction

     function bit set_id(int unsigned id);
       int unsigned results[$];

       if (id > `SEED_NUM)
         set_id = 0;
       else
         begin
           results = used_ids.find with (item == id);

           if (results.size() > 0)
             set_id = 0;
           else
             begin
               this.id = id;
               used_ids[$+1] = id;
               set_id = 1;
             end
         end
     endfunction

     function int unsigned get_id();
       return id;
     endfunction
  endclass

  // This component:
  // 1. Checks to see if it is set to be a domain root by looking for a domain_root_config
object placed for it in config table

  // A multi_seed_component configured as domain root component:
```

```
  // 1. Tries to parse its own seed from command line
  // 3. If it finds nothing, or the value it parses is "random", it takes the seed corresponding
to its id from the package seeds list
  // 4. If it finds something it takes whatever it finds
  // 5. Seeds all components under it by setting a seed configuration parameter

  // A multi_seed_component not configured as domain_root_component
  // 1. Tries to get a "seed" configuration parameter from the table
  // 2. If it finds one, it will reseed itsefl with it
  // 3. If it finds nothing, it will do nothing
  // This means that components outside any domain just depend on the seed command line
parameter for generating values.


  `define function_phase_rereseed(PHASE_NAME) \
  function void PHASE_NAME(uvm_phase phase); \
    super.PHASE_NAME(phase); \
    rereseed(phase); \
  endfunction

  `define task_phase_rereseed(PHASE_NAME) \
  task PHASE_NAME(uvm_phase phase); \
    super.PHASE_NAME(phase); \
    rereseed(phase); \
  endtask

  `define component_extension \
  \
    domain_root_config m_domain_root_config;\
    bit root_found;\
    bit seed_found;\
    int unsigned m_seed;\
    \
    function new(string name, uvm_component parent = null);\
      super.new(name, parent);\
    endfunction\
    \
    function void build_phase(uvm_phase phase);\
      super.build_phase(phase);\
      \
      root_found = uvm_config_db#(domain_root_config)::get(this, "", "domain_root_config",
m_domain_root_config);\
      \
      if (m_domain_root_config != null)\
        // component is root\
        begin\
          string plusarg_name, s;\
          int i;\
          $sformat(plusarg_name, "SEED%0d", m_domain_root_config.get_id());\
          \
          void'($value$plusargs({plusarg_name, "=%s"}, s));\
          \
          if (s == "random")\
            m_seed = seeds[m_domain_root_config.get_id()];\
          else\
            if ($value$plusargs({plusarg_name, "=%d"}, i))\
              m_seed = i;\
            else\
              uvm_report_error("CMD_LINE_ARG_WRONG", {"Got ", s, " as one of the seed values
specified on the command line. This value is wrong. Allowed values are 'random' or an integer"},
UVM_LOW);\
          \
          \
          uvm_config_db#(int unsigned)::set(this, "*", "seed", m_seed);\
        end\
      \
      seed_found = uvm_config_db#(int unsigned)::get(this, "", "seed", m_seed);\
      \
      if ((root_found) || (seed_found))\
        begin\
          int unsigned global_seed = uvm_global_random_seed;\
          uvm_global_random_seed = m_seed;\
          reseed(); // reseed component RNG\
          uvm_global_random_seed = global_seed;\
        end\
        \
        rereseed(phase); // reseed thread RNG for $urandom at build\
    endfunction\
    \
    // phase function and tasks implementations preform reseeding\
    // users are required to call these via super, prior to their own implementation\
    function void rereseed(uvm_phase phase);\
      int unsigned global_seed;\
      process p;\
      \
      if (root_found || seed_found) begin\
        global_seed = uvm_global_random_seed;\
        uvm_global_random_seed = m_seed;\
        p = process::self();\
        p.srandom(uvm_create_random_seed(phase.get_name(), get_full_name()));\
        uvm_global_random_seed = global_seed;\
      end\
    endfunction\
    \
    `function_phase_rereseed(connect_phase)\
    `function_phase_rereseed(end_of_elaboration_phase)\
    `function_phase_rereseed(start_of_simulation_phase)\
    `task_phase_rereseed(run_phase)\
    `function_phase_rereseed(extract_phase)\
    `function_phase_rereseed(check_phase)\
    `function_phase_rereseed(report_phase)\
    `function_phase_rereseed(final_phase)\

  class multi_seed_component extends uvm_component;
    `component_extension
  endclass

  class multi_seed_env extends uvm_env;
    `component_extension
  endclass

  class multi_seed_sequencer #(type REQ=uvm_sequence_item, RSP=REQ) extends uvm_sequencer#(REQ,
RSP);
    `component_extension
  endclass

  class multi_seed_driver #(type REQ=uvm_sequence_item, RSP=REQ) extends uvm_driver#(REQ, RSP);
    `component_extension
  endclass

  class multi_seed_test extends uvm_test;
    `component_extension
  endclass

endpackage
```