Random Stability in SystemVerilog

Doug Smith

Doulos
Austin, Texas, USA
www.doulos.com
doug.smith@doulos.com

ABSTRACT

A common problem that arises with constrained random verification is reproducing random stimulus for verifying RTL bug fixes and locking down test stimulus for regressions. In SystemVerilog, this is referred to as *random stability,* which is both a function of thread locality and hierarchical seeding. This paper discusses random stability, especially the use of good random seeds and locking down random number generator (RNG) seeding for test reproducibility. In addition, the RNG seeding employed in the leading verification methodologies like VMM, OVM, and UVM will be examined, tested, and critiqued, highlighting the strengths and gotchas.

Table of Contents

Table of Figures

1 Introduction

Over the years, random test stimulus has proven itself to be very effective at rapidly uncovering design bugs; in particular, exhaustively exploring a design's state space to hit those hard-to-find, corner-case scenarios. So the quality of the random stimulus is important in order to achieve a high degree of confidence that a design has been well tested. As a side effect of using random stimulus, there is the potential that test cases may become irreproducible.

Random stimulus relies upon the use of a *random number generator* (RNG). Of course, all RNGs strive to produce statistically proven, truly random numbers, but the reality is that not all RNG algorithms are created equal. They do share in common a starting point, which we call a *random seed*. While the algorithms may produce statistically proven random numbers, the algorithms themselves are quite deterministic. This is necessary if the random sequence of numbers is to be reproduced—an essential requirement for any verification effort for testing design bug fixes. Provided the same seed is given to the RNG, the same sequence of random numbers will be generated. Upon generating a random number, the seed is updated with the next value in the sequence so subsequent calls generate new random values.

Where engineers encounter difficulty in a random stimulus-based testbench environment is ensuring that the random number generation is seeded in a consistent and reproducible way. Unfortunately, modifications to a testbench often have the unpleasant side effect of changing the seeding, and therefore, altering the test stimulus so that the original failure is no longer reproducible.

In SystemVerilog, this behavior is referred to as *random stability*. Random stability relies upon two factors: (1) *thread locality*, and (2) *hierarchical seeding*. Thread locality refers to the fact that calls to the random number generator in each thread or process are independent of all other calls in other threads. In other words, each SystemVerilog process (thread) has its own independent random number generator so that the sequence of random numbers generated in a process is only affected by its own calls to the random number generator, and all other calls for random numbers have no effect on it. Likewise, each process receives its own individual seed through the process of hierarchical seeding, which means that seeds are passed down from one simulation element to the next as each element is created.

While thread locality solves most of the random-stimulus reproducibility issues since each thread is independent of each other, hierarchical seeding is crucial to get right if test cases are going to be reproducible. Some elements in a SystemVerilog simulation always receive the same seed, but most have the potential for unexpected seed changes. How SystemVerilog and the leading verification methodologies handle their seeding, is, in fact, the subject of this paper. In the following sections, a brief look at random number generators will be presented along with an in-depth explanation of the random seeding in a SystemVerilog environment and how to ensure it does not change. In light of this information, the leading SystemVerilog verification methodologies like VMM and UVM will be evaluated, highlighting their random stability shortcomings and strengths, and offering some helpful suggestions to make the methodologies even more robust.

2 Random Number Generators

The need for random numbers has been around since the beginning of computing for such things as simulating natural phenomena, random sampling, Monte Carlo simulations, etc. so it is no surprise that random number algorithms date back as far as 1939 to mechanical machines used to create tables of random numbers. John von Neumann suggested one of the first computer algorithms in 1946 called the "middle-square" method. This method squared a previous random number and took the middle digits out of it. While this seems reasonable, it has the problem that it tends to repeat elements and propagate them from one number to the next like zero, which continues to perpetuate itself in a sequence. [4]

Later, a simple algorithm known as the *multiplicative linear congruential generator (MLCG or LCG)* was introduced by Lehmer in 1948. It could be summarized as,

$$X_{n+1} = (a\ X_n + c) \bmod m$$

where X_0 is the starting value (the *seed*), a is the multiplier (usually a large prime number), c is the increment, and m represents the range the random number should fall in. For years, the LCG algorithm was considered adequate for most applications, and it became the standard implementation for most operating system's random function calls, including the standard C and C++ libraries. Unfortunately, the numbers it generates tend to correlate too closely to each other. In fact, instead of hitting all the random numbers in a specified state space, the numbers tend to lie upon correlated "planes." For example, generating numbers in the range space of 2^{32}, there are only 1600 planes where the numbers will tend to lie, meaning that you may only be generating test scenarios that only hit a small fraction of the total space. Likewise, the least significant bits are less random than the most significant bits[7]. Lastly, poorly written LCGs tend to repeat numbers in the sequence, usually with a short periodicity.

More recently, algorithms based on a linear feedback shift register like the Mersenne twister algorithm have proven much more robust without the short repeating period of the LCG and LSB issues. Such algorithms should be used when testing critical applications that depend on good quality random stimulus such as cryptology. Interestingly, the GNU library uses a LCG algorithm for numbers 32-bits or less, and a LFSR algorithm for all others. In their book, Numerical Recipes 3rd Edition, William Press et ali. present what they call their "suspenders-and-belt, full-body-armor, never-any-doubt-generator" in just a few short lines of code, which offers an alternative quick and robust solution[7].

When Verilog was introduced as an IEEE standard in 1995, a random number generator was provided by the system call to `$random`. At the time, `$random` was platform dependent because it invoked the C library's `rand` function. In Verilog-2001, the uniform random number function was published as part of the standard, and it is now included in appendix N of the IEEE 1800-2009 standard[1]. The source for `$random` and all the `$dist_*` functions are included. This provides a standard way of generating random numbers so that test results can be replicated

across simulators. Unfortunately, the algorithm used is not very sophisticated; in fact, it is the same one used in Verilog-XL and it appears to be a form of some kind of linear congruential generator, having vestiges of by-gone days such as using a small prime number like the one used in the old Vax glibc MTH$RANDOM function and an antiquated use of a floating point to integer conversion in the algorithm.

A better approach would be to use one of the several new random functions provided by SystemVerilog—`$urandom`, `$urandom_range`, and `randomize`. Unlike the old Verilog `$random`, the standard does not dictate the implementation for these functions. Instead, engineers rely on the EDA vendors to provide suitable and robust random number generators. For applications particularly sensitive to random stimulus (such as encryption), linking in high-quality RNG algorithms through the DPI would probably offer better test results and verification confidence. See [7] for suggestions on what RNGs to never use, and [5] for a very high quality random number generator.

3 Random Seeding

If we want to use all the random constraint syntax in SystemVerilog, then there is little we can do to control the quality of the random number generator algorithm since that is implemented in the simulator. What is within our control is how well we seed the randomization. While all seeds should be valid (except for perhaps 0) and produce equally random results, it does not mean that all seeds are good to use. For example, repeating the same seeds over and over again only limits the amount of state space tested, and using successive seeds or seeds based off of each other tend to produce results that are correlated to each other in some way.

3.1. Seeds to avoid

3.1.1. Zero

The simplest seed to avoid is 0. Linear congruential random number generators get stuck using zero[2]. The Park and Miller RNG algorithm proposed in [6] has been the minimal standard for random number generators for years, and their algorithm does not work with a seed of 0. Therefore, it is probably wisest to steer clear of zero since EDA vendors do not typically disclose their RNG algorithms.

3.1.2. Limited range of seed values

A limited range for seed values will produce a limited range of random sequences. For example, using a 1-byte seed limits the possible seeds values to 256. Consequently, there are only 256 random sequences that would ever be generated.

SystemVerilog uses an ‘int’ or 32 bits to seed its randomization (see `srandom` in [1]). That means there are 2^{32} or 4294967296 possible seed values. Often seeds are chosen using a limited range such as a Unix process ID (pid). While 32-bit operating systems typically use an int for their process IDs, most kernels limit the pid range to something much more human readable like 2^{15}-1 or 32767. Once the max value is hit, the process ID rolls back around to 0. Recall the last time you ever saw a process ID reach 4294967296?! Using a limited pid like this means there

are only a potential 32768 random sequences, or 4294934530 possibilities never explored! Therefore, avoid using a seed generator source that generates numbers less than 32 bits.

3.1.3. Time and date

By far, the most common method for seeding a simulation is to use the time and date. While this creates what appears to be a unique number, there is very little variation in the seeds. For example, consider the following seed created from the time 16:31:51 and date March 28, 2013. A seed can be generated several ways from this, but for illustration purposes consider the straightforward appending of all the digits:

16 31 51 03 28 2013

This is a large number, but consider how often during the course of running regressions for a project do the values actually change. It takes 365 days before the year changes, which means every seed used on a project will have the same year in the seed unless the project rolls into the next year. Even then, there is typically only a 1-digit variation in the number. Likewise, the month takes 30-31 days to vary, the days of the month repeat every 30-31 days, and as creatures of habit or by virtue of cron jobs, our simulation regressions tend to execute near the same time every day. In other words, the seeds used are not really random, they all relate to each other in some way, and typically have repeating or practically fixed digits in them. Likewise, there are only 60 seconds in an minutes, 60 minutes in an hour, 24 hours in a day and so on, so what about the values from 61-99 for seconds and minutes, and 25-99 for hours? Even if the standard C time is used—number of seconds since January 1, 1970—then that is even worse since only the least significant bits of the seed ever vary.

Using the time and date greatly limit the random state space exploration. Only a very small subset of the possible seeds are ever touch, and they all correlate to each other, which further reduces the state space explored. Sometimes, they may even overlap if just the time of day is used. These issues remain even if the time and date are appended with another "random" value such as the process ID.

Lastly, in this example the time and date generated a 14 digit number—always. What about seeds with other digit lengths? Recently, an engineer admitted to me that his regressions were not finding many bugs. Then it dawned on him and his team that their regressions always executed at the same time of day and they used the time and date for a seed. Once they moved to a truly random seed, all sorts of bugs were uncovered as tests began to hit new corner cases!

3.1.4. $RANDOM

One of my favorite ways to generate a random seed for a regression is using the bash shell's built-in `$RANDOM` variable. Reads of this variable trigger a call to bash's implementation of the Park and Miller algorithm[6], which is a tried and true RNG. It could not be easier to pass a different random seed for each regression test to the simulator than passing in `$RANDOM`.

While bash's algorithm is adequate, its seed generation is not. Bash uses the time of day (seconds and microseconds) plus the process ID. Subshells are seeded with just the process ID. Issues with these two methods are described in the previous two sections. Even if the seed were better, or the seed is overwritten by assigning to `RANDOM`, the results are very limited. Bash masks off the upper 32 bit number it generates into a 15-bit value between 1 and 32767.

Korn shell also provides a `$RANDOM`, but it relies on the poorly written C library functions `rand` and `srand`; therefore, it is not advisable to use it.

3.1.5. $random to seed $random

One of the problems with linear congruential generators is that when seeded with another random call to the same RNG, the results show a strong correlation with each other [3]. Using a statistical test suite that tests a billion bits of random data against random number generators used in cryptographic applications, test results show that using a LCG to seed another LCG gives statistically poor results[5]. Therefore, using SystemVerilog's `$random` to seed another call to `$random` would make a poor choice of seeds.

3.2. What makes for a good seed?

A good seed should really have the following characteristics:

(1) Generated from a random source
(2) *The random source should not be from the generator being seeded* (so the results do not correlate with each other)
(3) *Not limited to a small subset of values* (like time and date)
(4) Ideally, from a source that relies on physical randomness
(5) *As many bits as possible* (SystemVerilog uses 32 bits so not limited to 2^{15} bits)

The easiest way to generate a good random seed is to use a random number generator that uses randomness from an actual physical source, which is generally referred to as a "*truly" random number generator* (however, it is hard to get true randomness in a computer since they are designed to be predictable). For example, clock jitter in a system (though clocks tend to not be the best source), white noise on an audio port, or sporadic timings between keyboard and mouse entries[2]. A readily available source for random numbers in a Unix system is /dev/urandom. Urandom uses entropy pools inside of the system kernel and regularly injects into the pools random jitter measurements from the kernel. While not a purely random source, it does provide statistically good results for random numbers.[1] Urandom can be read with the following Unix commands:

```
1  % head -4 /dev/urandom | od -N 4 -D -A n | awk '{print $1}'
```

[1] Statistical tests performed by [5] show that /dev/urandom produces medium quality results since /dev/urandom is subject to uniformity flaws. However, this is should be adequate enough for most regression testing.

Another decent source for a seed would be calling the BSD glibc function `random`. `Random` uses a LSFR approach and should produce acceptable seeds for verification purposes. Such a call to `random` could be invoked from the DPI and passed into the SystemVerilog `srandom` function.

3.3. Hierarchical seeding in SystemVerilog

Beyond the choice of initial seeds, how the elements in a SystemVerilog simulation are seeded will determine the random stability of our testbench environment. SystemVerilog ensures that each thread has its own random number generator so that calls to the RNG are independent from each other. Again, this is referred to as *thread locality*. The tricky part is ensuring that every thread always gets seeded in the same way so changes to the testbench environment do not make test results irreproducible.

By default, seeds are passed down from structural elements to their processes. Statically elaborated constructs, referred here as *structural elements*, include modules, interfaces, programs, and packages[2]. This seeding approach is termed *hierarchical seeding*. The initial seed of the simulator is either passed in as a command line argument or set to an arbitrary default value. This initial seed becomes the seed of each structural element as shown in Figure 1.

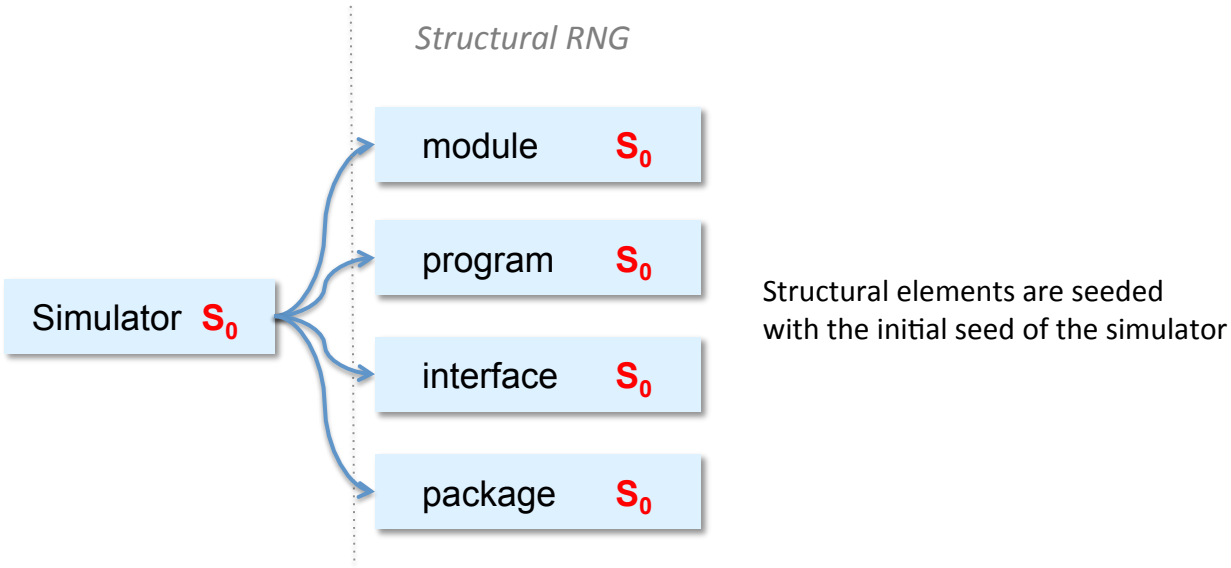

Figure 1: Hierarchical seeding of structural elements in a SystemVerilog simulation.

Hierarchically seeding protects the simulation from being irreproducible when adding new modules to a testbench or design.

[2] Packages are not technically "structural" since they are neither instantiated nor have hierarchy.

Within structural elements (save packages), processes exist like `always` or `initial` blocks. The SystemVerilog standard requires that each process (or thread) has its own independent random number generator and is seeded with the next random value from its parent—either the parent module or its parent process as in the case of a dynamic fork statement[1] (see Figure 2).

Practically, this means adding a new process inside a module could potentially alter the order that the processes are seeded. This is where the random "instability" enters into a testbench. If someone alters the order that random calls are made within a process, then the random sequence will be changed, which is the expected behavior of a RNG. However, if someone decides to simply add a new `initial` block, even though each process is theoretically independent, the new `initial` block could change previous test results because the seeding of the processes may change, altering the previous random sequence. This is, in fact, a common gotcha that many engineers encounter. Fortunately, the standard mandates that seeding must happen deterministically, effectively from top to bottom inside of a module. Therefore, if a new process is added at the bottom of a module, then it should not affect the seeding of any previously defined process blocks.

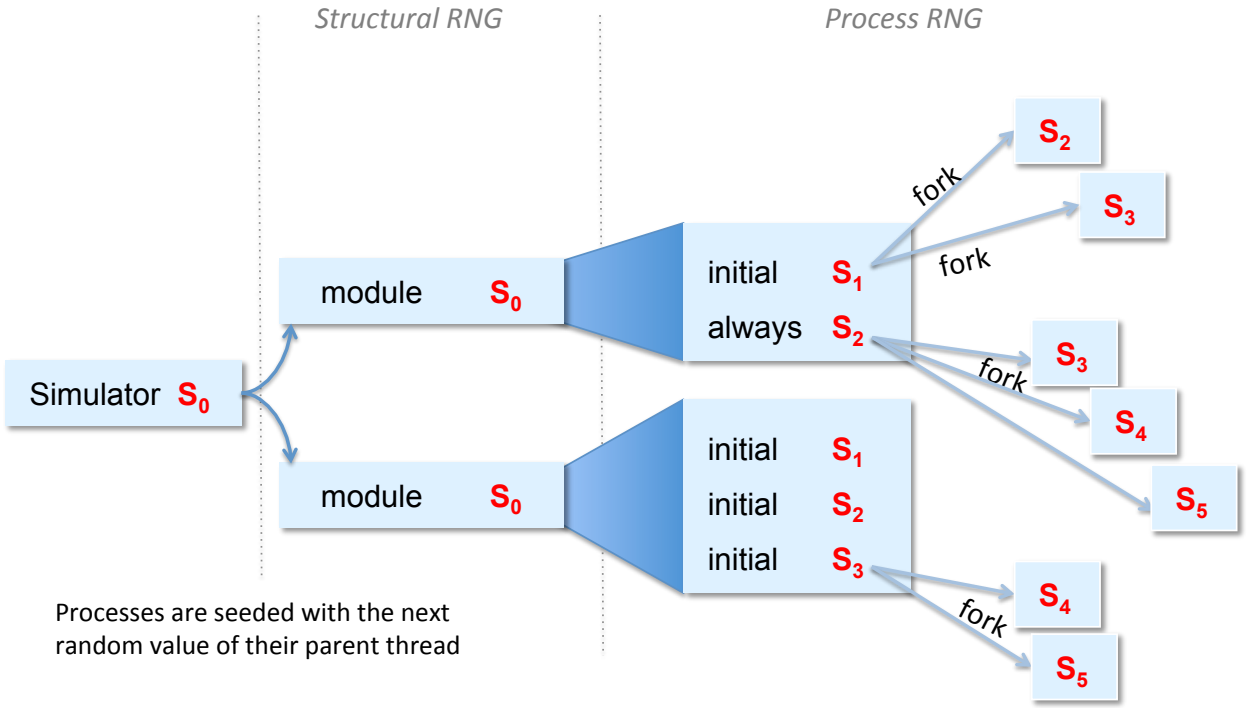

Figure 2: Seeding of processes inside of a structural element.

In class-based testbenches, objects will represent the components in the testbench and generate the random stimulus that passes into the design. In order to ensure random stability, each object also must have its own independent random number generator, and consequently, its own seed so previous random stimulus can be reproduced. The SystemVerilog standard requires that objects be seeded with the next random value from their parent thread as shown in Figure 3.

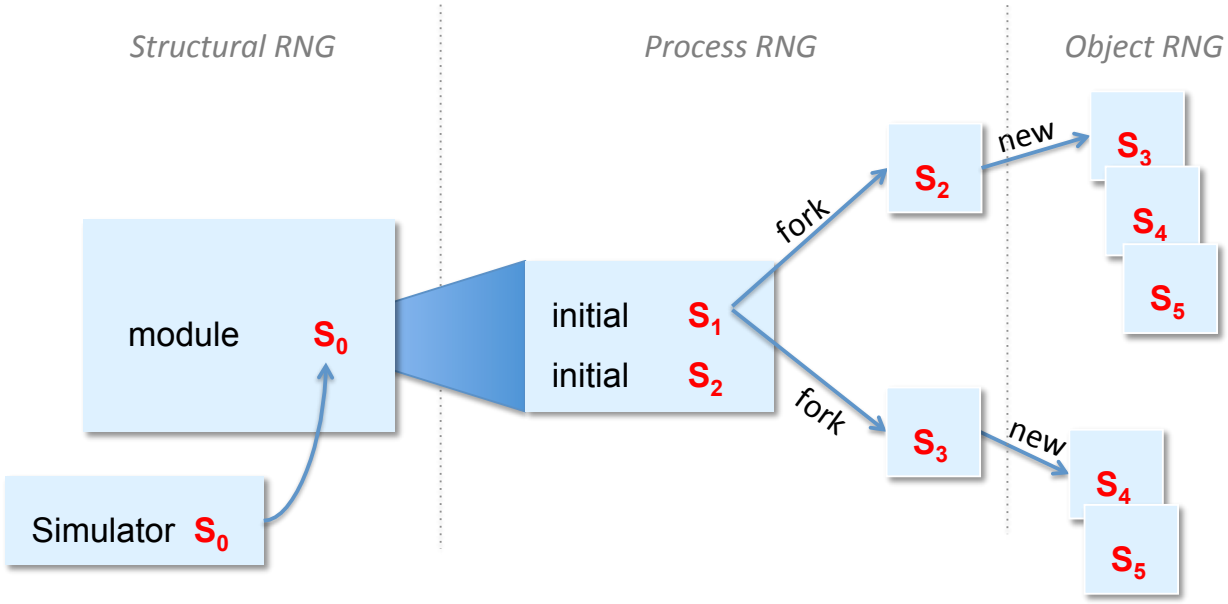

Figure 3: Hierarchical seeding of objects.

Again, where random "instability" enters into a testbench is when changes to the order that objects are created happens within a thread of execution due to modifications to the testbench. For example, instantiating new components before existing ones could cause the objects' initial seeding to change, resulting in a new sequence of random stimulus. Likewise, creating new transaction objects could also affect the seeding of subsequent objects.

3.3.1. Simulator differences

Unfortunately, the SystemVerilog standard[1] is slightly ambiguous on how hierarchical seeding works. For example, the standard says in section 18.14.1, "An *initialization RNG* shall be used in the creation of static processes and static initializers," and that, "Each *initialization RNG* is seeded with the default seed." However, it subsequently says that static processes (e.g., initial, always, fork..join) are seeded with the "next value from the initialization RNG of the module instance, interface instance, program instance, or package containing the thread declaration." For that reason, it is not clear if static processes should be initialized using the default seed (i.e., initialization RNG) or seeded from the parent module? Likewise, is the "*next value* from the initialization RNG" interpreted as the next random value generated by the module's RNG or simply the seed that the module was seeded with?

The author interprets the standard as intending static processes (initials, always, and fork..joins) to be seeded with the next random value of the containing module's RNG. Sadly, none of the major simulators interpret the standard quite the same way. SystemVerilog's `get_randstate` function reveals each simulator's hierarchical seeding:

```
2  module test;
3
4     process m = process::self();
```

```
5
6     initial begin
7         process p;
8         p = process::self();
9
10        $display("Module  randstate = ", m.get_randstate());
11        $display("Process randstate = ", p.get_randstate());
12    end
13 endmodule
```

For example, the latest version of Questa initializes all static threads to the same initial value instead of the next random value of the containing module. VCS initializes static threads with the next random value of the module's RNG, but initializes all statically declared objects with the same seed instead of the next random value in the module's RNG. Incisive simulator initializes all static processes with different random values. None of these implementations follow the standard exactly. While it is frustrating that simulators are not more consistent for portability purposes, these minor inconsistences should have little affect on test reproducibility provided the simulators consistently follow their chosen method.

3.4. Locking down the seed

With so many initial seeds throughtout a testbench, there is little wonder why testcases can become irreproducible. Random instability typically creeps in from one of the following causes:

(1) A change in the order of random calls with a process
(2) Insertion of new processes before previously defined ones
(3) A change in the order of creation of forked processes
(4) A change in the order of object creation

In practice, this means that even a small or subtle change can affect test results. Unfortunately, there is no absolute way to guarantee that random test results will never become irreproducible. The reason is because the random seeding can always be messed up. Except for certain SystemVerilog random functions like `$random`, `$dist_uniform` (and all `$dist_*` functions), which have a user provided seed as an argument, the other random functions like `$urandom`, `$urandom_range`, and `randomize` are hierarchically seeded. Therefore, nothing stops a user from inserting additional RNG calls before existing calls, thereby altering the seed and the resulting random sequence.

So while we cannot guarantee reproducibility if the *test stimulus is modified*, we can guarantee reproducibility if the *testbench is modified*. To do so, we need to somehow lock down the seeding of each hierarchical element in the testbench so that the same random sequence can be recreated. The simplest way to accomplish this is to either: (1) carefully control the creation of testbench elements, (2) use a template generator and deterministically seed it, or (3) manually seed each hierarchical element.

3.4.1. Controlling the creation of hierarchical elements

To control the creation of simulation elements, you could prohibit adding new processes (usually `initial` blocks) or new object instantiations. Such a policy would be impractical, of course, since testbench changes are inevitable. However, as long as new processes and object instantiations are added after existing code, then simulation elements should continue to be seeded in the same order and preserve the random stimulus. Enforcing this policy may be difficult with multiple testbench developers so one of the following approaches may be more useful.

3.4.2. Using a template (atomic) stimulus generator

One way to control SystemVerilog's hierarchical seeding is to create template objects, which are used to generate all the random stimulus. As long as all calls to the random number generator are only made via the template object(s), then the order of random seeds will be preserved since SystemVerilog guarantees object random stability. Likewise, if the template object is initially seeded in a controlled manner, then changes to the rest of the testbench will not affect it. An easy way to control the seeding of a template object is to restrict the verification environment to one initial block, which always gets the same seed, and then create the template object first thing at time zero before any other calls to `new` or the RNG. This will ensure that the template will always be seeded with the same random value as shown in Figure 4.

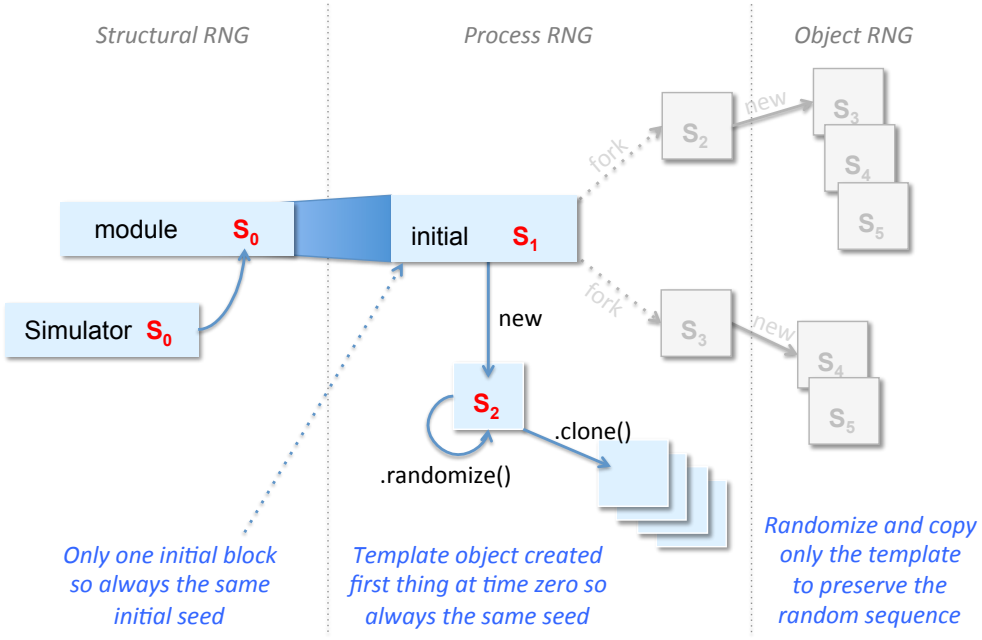

Figure 4: How to lock down the random seed using a template object.

An even easier approach would be to use a statically declared initializer. A statically declared initializer is an object created at time zero by using a call to `new` in a variable initialization. For example,

```
14 package my_pkg;
```

```
15    class test;
16       …
17    endclass
18
19    test t = new();       // Statically declared initializer
20
21 endpackage
```

The standard requires that these objects be seeded with the "next random value of the initialization RNG of the module instance, interface instance, program instance, or package in which the declaration occurred." Provided no additional static processes or initializers are added before the variable declaration, then the object will always be initialized with the same seed (Figure 5).

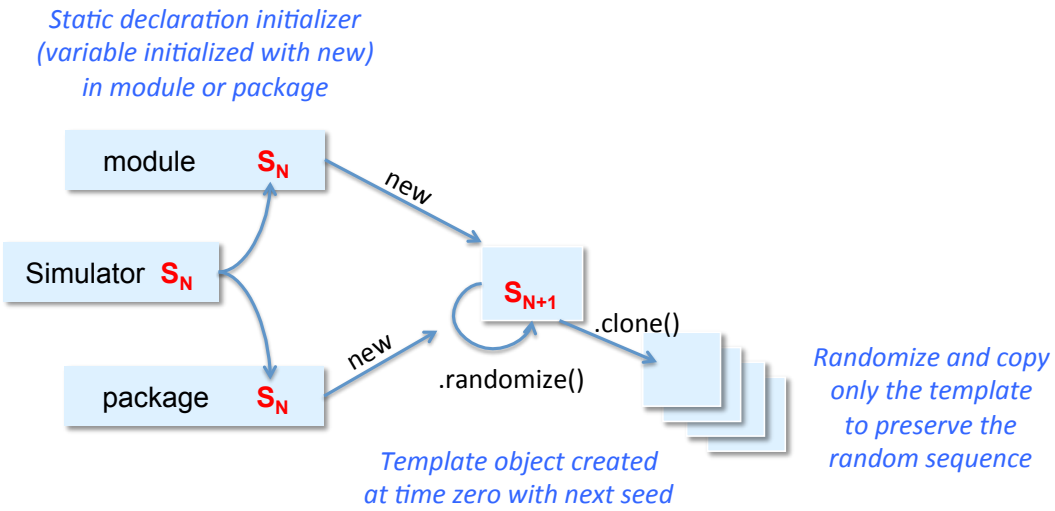

Figure 5: How a statically declared initializer is seeded.

3.4.3. Manual seeding

The only way to really guarantee that a testbench environment is impervious to changes is to manually seed all processes and created objects. SystemVerilog provides two ways to control the random seeding: (1) `srandom`, and (2) `set_randstate`. To seed a process, `srandom` is called on a process object with a 32-bit integer seed:

```
22 initial
23 begin
24    process::self.srandom( 1234 ); // initial block seed=1234
25
26    fork
27    begin
28       int x;
29       process::self.srandom( 1 ); // Forked process seed=1
```

```
30         x = $urandom;
31      end
32      …
33      join
34 end
```

Seeding an object simply requires invoking `srandom` on the object:

```
35 initial
36 begin
37      my_test t = new;
38      t.srandom( 6789 );      // Test object seeded with 6789
39 end
```

The second approach to seeding an object is using `get_randstate` and `set_randstate`. These methods are invoked just as `srandom` on a process object. The `get_randstate` method returns the current RNG state for a given object, returning an implementation-specific string. For example, VCS returns a string like this:

```
00Z1ZZZ1Z0ZZ11XZX11Z1ZX1XZ1Z0X01XZZZZZXZXZXZZZZXXXXZZXZZXXZZXXZZ
```

or

```
00000000000000000000000000000000011010101000101100110111111010O
```

The `set_randstate` function receives this implementation-specific string. The disadvantage of using the `get_/set_randstate` methods is that every simulator uses a different format for its string. Practically, it is only useful within a simulation to save the RNG state at the beginning of a test run and then later restoring it when the test case is restarted.

4 Evaluation of random stability in the leading methodologies

When developing a verification methodology, it is important to consider the random stability and how to ensure test reproducibility. Both the initial seeding and the hierarchical seeding are important for reproducibility and achieving good test coverage across the design's state space. Therefore, it is important that the leading industry verification methodologies follow a robust approach for random seeding. The following section will evaluate the strengths and weaknesses of the random seeding methods for VMM and OVM/UVM.

4.1. VMM

4.1.1. Strengths

Disappointingly, VMM provides little support for random stability within a verification environment. However, it does use the `get_randstate` and `set_randstate` functions so that test

cases can be reset and restarted with the same initial seed within the same simulation run. While this helps during a simulation, it does nothing to guarantee random stability across different simulation runs where the testbench environment has been changed.

VMM also favors the idea of a *data factory* where a template can be used to generate repeated stimulus. Data factories or template generators can help avoid reproducibility issues provided the template object is always seeded first in a known order. If, however, the order that objects are created is accidentally changed, then random stability of the environment is compromised.

4.1.2. Weaknesses

As mentioned, VMM offers little with regards to random stability. Noticeably missing is the use of `srandom` or a methodology that ensures test stimulus reproducibility. So naturally, VMM 1.2.2b provides no built-in mechanism for setting an initial seed. Instead, the initial seed is set using a simulator specific option on the command line and then passed into the testbench components according to SystemVerilog's defined hierarchical seeding method. Since `srandom` is not used on any of the testbench objects, simple changes in the order that processes are created or objects are instantiated easily causes a VMM environment to become irreproducible due to the change in the seeding.

Furthermore, while `get_/set_randstate` is used for restarting tests within a simulation, it is only used in the constructor calls of `vmm_xactor`s and `vmm_env`. In fact, the random state of `vmm_data` objects are not reset, which may pose a problem with reproducing test stimulus if the template generator approach is used as described in Section 3.4.2.

4.1.3. Suggested enhancements

VMM could significantly enhance its random stability by manually seeding all VMM objects in their constructor calls using `srandom`. A reproducible and robust seeding algorithm would be required in order to generate the initial seeds. Until changes are made to VMM, users can manually seed their own objects, which would lock down their testbench seeds.

> ***VMM Random Stability Guideline:*** *Manually seed all testbench objects using* `srandom`*.*

Likewise, VMM would benefit from a command line argument such as VMM_SEED so that the seed can be easily passed into the testbench and then used for seeding all the other VMM objects. A static initializer in a package could be used to capture that initial seed and then used to seed all other objects in the testbench.

4.2. OVM / UVM

OVM and UVM share an identical random seeding methodology. OVM/UVM use a global random seed that is set inside a package by a call to `$urandom`:

```
40 int unsigned uvm_global_random_seed = $urandom;
```

Since packages always get initialized with the default initial seed and this is the only RNG call in the package, the global seed can be consistently set to the same value every simulation run. When an OVM/UVM component is created, a flag called `use_uvm_seeding` is checked to determine whether SystemVerilog's hierarchical seeding is used or OVM/UVM's seed generation hashing algorithm. By default, components are seeded using OVM/UVM's hashing function, which generates a unique seed based on a component's (1) *data type*, (2) *full instantiated path*, and (3) the *global seed*. The hashing function uses a CRC algorithm to avoid collisions and uniquely seed each instance. UVM's hashing algorithm can be found in `src/base/uvm_misc.svh` of the source code and the CRC algorithm is included here for consideration of the reader:

```
41  // Function- uvm_oneway_hash
42  //
43  // A one-way hash function that is useful for creating srandom seeds. An
44  // unsigned int value is generated from the string input. An initial seed can
45  // be used to seed the hash, if not supplied the uvm_global_random_seed
46  // value is used. Uses a CRC like functionality to minimize collisions.
47  //
48  parameter UVM_STR_CRC_POLYNOMIAL = 32'h04c11db6;
49  function int unsigned uvm_oneway_hash ( string string_in, int unsigned seed=0 );
50    bit          msb;
51    bit [7:0]    current_byte;
52    bit [31:0]   crc1;
53
54    if(!seed) seed = uvm_global_random_seed;
55    uvm_oneway_hash = seed;
56
57    crc1 = 32'hffffffff;
58    for (int _byte=0; _byte < string_in.len(); _byte++) begin
59       current_byte = string_in[_byte];
60       if (current_byte == 0) break;
61       for (int _bit=0; _bit < 8; _bit++) begin
62          msb = crc1[31];
63          crc1 <<= 1;
64          if (msb ^ current_byte[_bit]) begin
65             crc1 ^=  UVM_STR_CRC_POLYNOMIAL;
66             crc1[0] = 1;
67          end
68       end
69    end
70    uvm_oneway_hash += ~{crc1[7:0], crc1[15:8], crc1[23:16], crc1[31:24]};
71
72  endfunction
```

4.2.1. Strengths

The strength of OVM/UVM's random stability methodology is that it manually seeds every object created, and uses a robust algorithm to ensure quality seeding throughout a testbench. While

neither fool-proof or unbreakable, OVM/UVM provides a solid methodology for ensuring random stability.

4.2.2. Weaknesses

Nonetheless, OVM/UVM's random seeding methodology is not without weakness. While each component is manually seeded, seeding can still be affected by changes to the environment. Changes to a component's instance name will change the generated seed for the component and any component hierarchically beneath it. OVM/UVM's seeding algorithm uses a component's full hierarchical pathname so any hierarchical or instance name modifications at the top will affect pathnames throughout the entire testbench environment. Likewise, changes to the component's class name will change the generated seed. Realistically, however, component class names will probably never change once created, though instance names may. *Therefore, it is advisable to not touch the hierarchical naming once a testbench environment has been setup*. Adding additional component instantiations has no affect on existing components, which accomplishes the goal of random stability in the testbench environment.

> ***OVM/UVM Random Stability Guideline:*** *Do not change the hierarchical naming once a testbench environment has been setup.*

Transactions (sequence items) and sequences are also manually seeded, but they have no hierarchical path so their instance name is just the simple name passed into the constructor. Of course, making changes to a testcase's stimulus makes the test scenario potentially irreproducible; however, with OVM/UVM's seeding method, changes to test stimulus on different interfaces, which use different transaction objects, should remain independent of each other.

4.2.3. Suggested enhancements

There are a couple improvements that could be made for OVM/UVM's random stability methodology. For starters, OVM/UVM could provide a command line argument for passing in an initial random seed. Currently, all the simulators have proprietary command line options so there is no standard way for setting the initial seed. Likewise, there is no standard way in SystemVerilog to determine the simulator's initial seed since `get_randstate` provides an implementation-specific string. Having a command line option would not only provide a portable option, but it would allow OVM/UVM to display the initial seed in a standard way into the log file so seeds can be easily extracted and passed in again to reproduce the test stimulus. The OVM/UVM global seed could be set using `$urandom(UVM_SEED)` instead of just `$urandom`.

Another possible improvement would be to separate the random seeding in OVM/UVM from an object's hierarchical name. While this method helps to uniquify the random seed, any changes to the instance names can affect test case reproducibility. However, without using the instance name, the hashing algorithm uses the order of creation to unquify the seeds. So either way, specific changes to a testbench can affect the hierarchical seeding. Unfortunately, there is no easy workaround for this except to seed each object of the same type with the same seed. While this is a valid approach, it may not result in the best random results.

5 Conclusion

Random stability is important in any testbench environment. In SystemVerilog, there are three simple methods that can guarantee random stimulus reproducibility: (1) control the order of process and object creation, (2) isolate the random seeding by using a template stimulus generator, or (3) manually seed each testbench element. Controlling the order of creation is not only difficult to manage and regulate, but all the simulators seed testbench elements in a non-standard way. Using a template stimulus generator works well, but real-world stimulus typically requires more than just a stream of random data; rather, designs require specific sequences of transactions such as provided by VMM's scenarios or OVM/UVM's sequences. The best way to ensure random stability is to manually seed each simulation element. The only concern with this approach is ensuring that a robust seeding algorithm is used.

There are several improvements in the SystemVerilog standard that could be made for random stability. First, section 18.14.1 on random stability needs clarification so all the simulation vendors implement the same hierarchical seeding. The archaic RNG implementations for the `$random` and `$dist_*` functions should really be updated with a more robust algorithm. Also, it might be worth considering enhancing the random functions in SystemVerilog to use 64-bits instead of 32-bits.

Considering the leading industry methodologies, a standardized way to pass in an initial seed on the command line would help with portability and for recording the random seed to a log file for reproducing test cases. VMM in particular, does a good job locking down the seeds within a simulation for restarting a testcase, but does not offer any other kind of random stability between separate simulation runs. VMM would greatly benefit from using `srandom` to seed the testbench processes and objects. OVM and UVM do a good job at random stability.

Another possible improvement to help all the methodologies is to provide some additional random functions via the DPI interface. DPI functions could offer 64-bit RNG functions as well as provide better RNG algorithms like those provided with the C++'s Mersenne twister engine or any of C++'s 20 random distribution functions. Another useful DPI function would be to generate a truly random seed, which could be used to initially seed a VMM/OVM/UVM simulation if nothing is provided on the command line. As discussed in Section 3.2, /dev/urandom provides a good source for random data in a Unix environment and could be easily read in C:

```
73 int myRandomInteger;
74 int randomData = open("/dev/urandom", O_RDONLY);
75 read(randomData, &myRandomInteger, sizeof myRandomInteger);
76 close(randomData);
77 return (myRandomInteger);
```

With some basic policies, using an industry methodology like UVM essentially eliminates concerns for random stability in a testbench environment. Separate from the methodology is the initial seed passed into the simulation. Without a good initial seed, test stimulus only hits a limited part of the design's state space. Limited seeds like date and time should be avoided while

seeds from a good random source are essential for good test coverage. A robust methodology combined with a good source for initial seeds provides for a verification environment that is change-resistant and apt to produce high-quality results.

6 Acknowledgements

Any trademarks or other proprietary names mentioned in this paper are acknowledged as the property of their respective owners.

7 References

[1] "IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language," IEEE Std 1800-2009, 2009.

[2] Garfinkel, Simpson, Gene Spafford, and Alan Schwartz. Practical Unix and Internet Security. 3rd Edition, O'Reily Publishers, 2003.

[3] Jain, Raj. "Random Number Generation", Presentation, University of Washington, 2006. Available online at http://www.cse.wustl.edu/~jain/cse567-06/.

[4] Knuth, Donald. Seminumerical Algorithms: The Art of Computer Programming. Vol. 2, Addison-Wesley Publishing Company, Reading Massachusetts, 1969.

[5] LavaRnd, http://lavarnd.org.

[6] Park, Stephen H. and Keith W. Miller. "Random Number Generators: Good One Are Hard To Find," Communications of the ACM, Vol. 31(10), October 1998, p. 1192-1201.

[7] Press, William H., Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery. Numerical Recipes 3rd Edition: The Art of Scientific Computing. Cambridge University Press, New York, 2007. Available online at http://www.nr.com.

[8] -----. Numerical Recipes in C: The Art of Scientific Computing. 2nd Edition, Cambridge University Press, New York, 1992. Available online at http://www.nr.com.

[9] "UVM Reference Manual Version 1.1c", Accellera, 2012.