# Improving Constrained Random Testing by Achieving Simulation Verification Goals through Objective Functions, Rewinding and Dynamic Seed Manipulation

Eldon Nelson M.S. P.E.
Intel Corporation ( eldon_nelson@ieee.org )

*Abstract*–**Constrained Random simulation is so critical to modern verification environments that it is a major component of the SystemVerilog language itself. This paper proposes a method that improves how UVM Constrained Random simulations are run. By abstracting the purpose of a simulation to be achieving "Objective Functions" (nominally coverage goals), it is possible to have the simulation autonomously explore deep possibilities from multiple points in time of a standard UVM testbench governed by feedback. This method has a number of benefits including: faster automated coverage closure, an efficient final stimulus solution and proposed higher quality of coverage.**

## I. Introduction

The current approach for applying Constrained Random [1, p. 467] to SystemVerilog verification environments is to apply a single seed at the beginning of the simulation. Through a feature called "thread stability" [1, p. 503] it can be shown that a single seed is then passed down in a deterministic way through the entire verification environment. When simulation time starts, every object uses its given seed to govern its own internal SystemVerilog RNG [1, p. 501] (Random Number Generator). The randomization is deterministic from time zero in the simulation, which of course, is a requirement for reproducibility. However, it is possible to dynamically re-seed a SystemVerilog object during simulation time with the built-in SystemVerilog function `srandom(int)` [1, p. 502]. Dynamically re-seeding a SystemVerilog object would alter its next `randomize()` function call.

If the goal of verification is to exercise the design, then a way to measure how well it is exercised is through the use of coverage – both functional coverage and code coverage. Coverage allows introspection into the design to see if certain conditions are met. Perhaps less used in practice are the SystemVerilog functions to look at these conditions dynamically during the simulation. Built-in functions of SystemVerilog covergroups such as `get_coverage()` [1, p. 547], return a floating point value representing the coverage percentage reached of a particular covergroup. It is then possible to dynamically measure the progress of certain verification goals during the simulation.

The concept of the "Objective Function" is introduced here, which is a measure of a user-specified function queried from within the SystemVerilog environment. An "Objective Function" could simply be defined by the user as: calling the covergroup built-in function `get_coverage()` on a particular covergroup. A more ambitious goal would be adding the resultant `get_coverage()` values from a number of related covergroups and using that as the "Objective Function". The "Objective Function" could be as general as calling the SystemVerilog built-in function `$get_coverage()`, which returns a floating point number between `0` and `100` representing the aggregate value of all covergroups in the design. The "Objective Function" could be any function that returns a number that increases in value as it approaches its goal. Therefore, it is possible that using something as abstract as line coverage of the DUT could be used as the "Objective Function", which could be useful to demonstrate a particular stimulus is activating lines of code not yet exercised.

All three major simulators [2] [3] [4] support the concept of checkpointing which is typically used for saving design state in long running jobs or to go back to a point before interactive changes were made. Checkpointing is the final required feature that opens up an intriguing method to improve Constrained Random. It is possible to dynamically re-seed the environment during the simulation, run for a simulation time "Interval" and then compare the increase in the "Objective Function". If the "Objective Function" has not increased over the time "Interval", then simulation time is moved back to the last checkpoint to start the feedback loop again. If the seed resulted in an improvement to the "Objective Function", then simulation time continues using the latest seed (Figure 1). "Objective Functions" combined with dynamic seeding and rewind has a number of compelling benefits over how Constrained Random is implemented today.

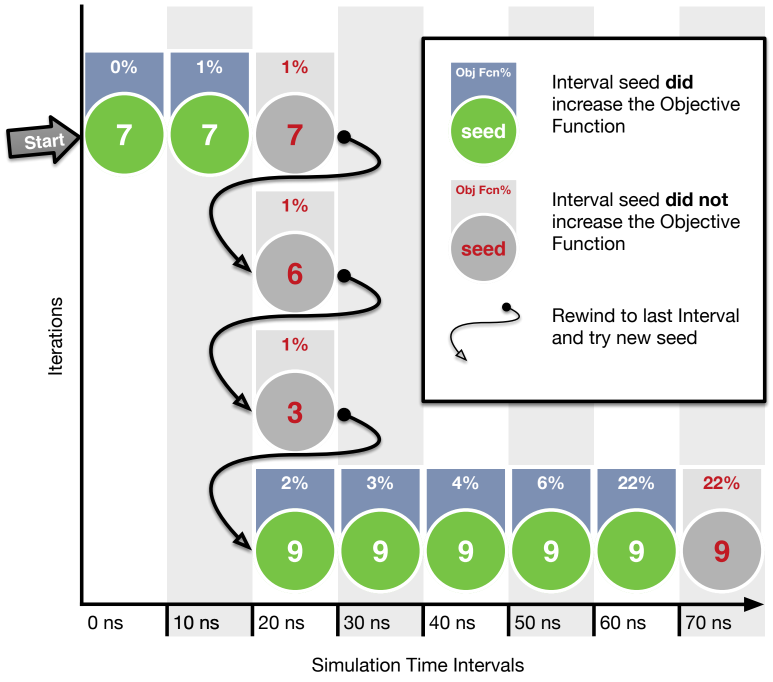
Start
0%
7
1%
7
1%
7
1%
6
1%
3
2%
9
3%
9
4%
9
6%
9
22%
9
22%
9
Obj Fcn%
seed
Interval seed **did** increase the Objective Function
Obj Fcn%
seed
Interval seed **did not** increase the Objective Function
Rewind to last Interval and try new seed
Iterations
0 ns
10 ns
20 ns
30 ns
40 ns
50 ns
60 ns
70 ns
Simulation Time Intervals

FIGURE 1. DYNAMIC RESEEDING WITH REWIND GOVERNED BY OBJECTIVE FUNCTION OVER INTERVAL

## II. APPROACH

One of the major influences for this effort was a paper by Tom Murphy VII that was presented in 2013 at the Sigbovik conference in Pittsburgh. In his paper "The First Level of Super Mario Bros. is Easy with Lexicographic Orderings and Time Travel . . . after that it gets a little tricky" [5] [6] Murphy demonstrates a method to teach a computer to autonomously play the Nintendo game Super Mario Brothers. Unlike other approaches to creating an artificial intelligence to play the game, Murphy chose to only observe RAM memory of the running game in order to find patterns that would tell him that the game was progressing towards its goal. He doesn't try to understand what is going on with the game mechanics or even look at the output video or audio. Murphy's method looked at registers during every frame while the game was running and developed algorithms to infer it was progressing without knowing what the game looked like - or even what the goal of the game really was.

This paper proposes a method that uses inspiration of the Murphy paper to optimize the exploration of a hardware design being stimulated by a constrained random test environment. The parallels are numerous. Murphy proposes using an "objective function" [5, p. 2] based off of values in RAM locations read at many points in time to figure out if the game was progressing. The author also uses "Objective Functions" and applies it simply to covergroups for the purpose of the paper, but other arbitrary functions can be applied as well. Murphy uses multiple random stimulus attempts over periods of time he calls steps [5, p. 8] or iterations [5, p. 9] choosing one attempt for that iteration that is the most promising. This method uses the term "Intervals" to explore discrete possibilities in an expanding state space. Murphy does use ideas already established in Verification; such as, constrained sequences which he calls motifs [5, p. 6] in his solution.

The full source code to implement this method is released under the GNU GPLv3 License [7] and is available on GitHub [8] under the project "improving-constrained-random" [9]. There is a makefile that can be used to build and run the same simulations as demonstrated in the paper. Running `make help` will describe the makefile targets that are available. While the current implementation is achieved in one simulator [4], it should be easy to port to other simulators.

`https://github.com/tenthousandfailures/improving-constrained-random`

## III. Benefits

When considering this approach, we can think about the impact on the quality of coverage results. For example, if there is a set of related features that need to be exercised that are grouped into a particular “Objective Function”, this method will iterate until it satisfies that set of related features in a single test. In current Constrained Random environments that set of related features might be partially covered over many independent tests. Then the union of that coverage would indicate that the feature was sufficiently covered. Rigorously exercising a feature can be done with directed testing or careful manipulation of constraints to get to a certain goal. But if the goal is stated simply in terms of coverage, the “Objective Function”, this method can find a complete stimulus solution autonomously in a single test that is not diluted by being a union of many tests.

The second benefit of this method is that it does not require any knowledge of the design to operate; only the creation of user defined “Objective Functions” to guide each test toward a particular goal. A method with a similar final result as this method is Graph-Based Stimulus [10], which seeks to drive-by-planning all possible stimulus. The contrast of this method and Graph-Based Stimulus is that this method can work without knowing the path to reach a verification goal within the design; this method relies only upon blind feedback to explore the state space to attempt to reach the verification goal. There are “Objective Functions” that could be defined that have no direct relationship to the stimulus which would challenge Graph-Based Stimulus solutions. For example, directing external stimulus to drive a FIFO deep within the DUT (Design Under Test) to be empty or full is not something that Graph-Based Stimulus may be able to solve easily. This method is the simplest feedback loop [5] that actually works, requiring possibly less effort than calculating stimulus.

The third benefit of this method is that an efficient stimulus solution to satisfy the “Objective Function” is produced. There are parallels with FPV (Formal Property Verification) and this method, with a contrasting property being the ability to reuse existing UVM and VIP (Verification Intellectual Property) assets. The trick that enables much of the power of FPV is also the one that disallows any class based code from being part of the verification environment since it would handicap the FPV solver to simplify the problem. In this method, many possible seeds and paths are evaluated and only the final feedback created path is presented. This method produces a way to repeat a simulation by providing a file that details what dynamic seed is used at a particular simulation time. During the simulation, all of the expected self-checking and scoreboarding of the UVM environment is still occurring. And if an error is encountered during any possible path, the test can report that error and exit; allowing for conventional bug hunting and debugging activities.

## IV. RESULTS AND IMPLEMENTATION

The DUT chosen for demonstration and discussion is a two input parameterized design that compares its two input vectors: `a` and `b`. If `a` and `b` are equal, the single output `c` is raised. There is a covergroup inside the DUT that measures that all possible matching values on `a` and `b` have been seen on the positive edge of the clock. This simple DUT (Figure 2) will help us understand how to apply this method to improve the application of constrained random stimulus.

```
module dut #(parameter width=2) (
       input [width-1:0] a,
       input [width-1:0] b,
       input             clk,
       input             reset,
       output            c
                                 );

   reg [width-1:0] match;

   covergroup objective_cg;
      coverpoint match;
   endgroup

   objective_cg objective;

   assign c = (a == b);

   always @(posedge clk) begin
      if (c) begin
         match <= a;

         #1;
         objective.sample();
      end else begin
         match <= '0;
      end
   end

   initial begin
      objective = new();
   end

endmodule
```

FIGURE 2. COMPLETE SOURCE CODE OF DUT [11]

The test environment (Figure 3) is a simplified conventional UVM test environment. In order to simplify the code, no UVM environments or agents are used to make the code as succinct as possible, although we do use the conventional `uvm_test` and `uvm_sequence_item` for our test environment.

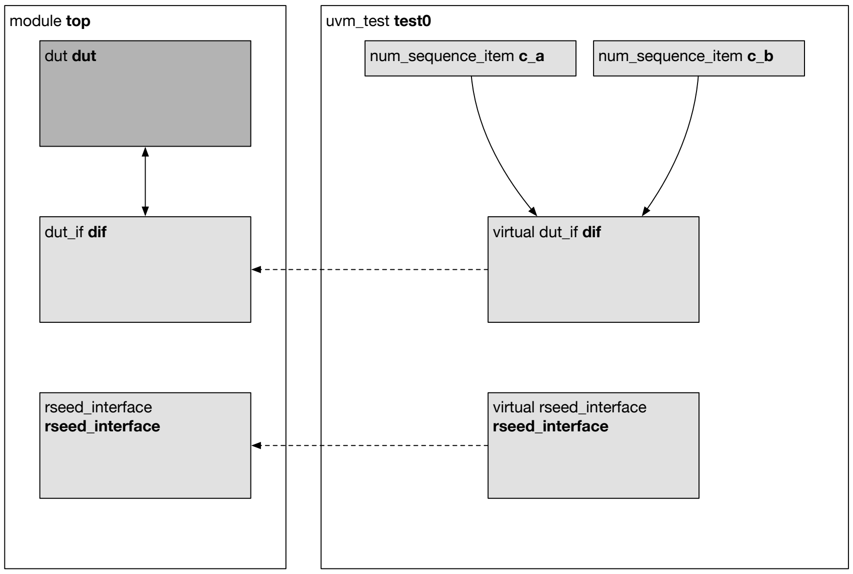
module **top**
dut **dut**
dut_if **dif**
rseed_interface **rseed_interface**
uvm_test **test0**
num_sequence_item **c_a**
num_sequence_item **c_b**
virtual dut_if **dif**
virtual rseed_interface **rseed_interface**

FIGURE 3. TESTBENCH DESIGN

The interesting feature in Figure 3 is the inclusion of the `rseed_interface` [12] (randomization seed interface) which provides a static testbench location that provides the simulator a known path to call SystemVerilog functions that dynamically interrogate or modify the verification environment. This reusable component has methods and variables within it that will inform the simulator what the test wanted to accomplish. A decision was made that the SystemVerilog would be the master for all configuration and goals in order to facilitate porting this method to other simulators. Figure 4 is an excerpt of the `rseed_interface` which illustrates some of the variables and functions that are within `rseed_interface`.

```
interface rseed_interface (
                     input clk,
                     input reset
                     );

   bit                      trigger         = 0;    // breakpoint trigger for eval loop
   time                     start_time      = 7;    // sim time to start evaluation loop
   time                     iteration_time  = 10;   // the time between evaluation loops
   bit                      final_report    = 0;    // breakpoint trigger for end of test
   int                      max_target      = 100;  // the desired Object Function final value
   int                      unsigned seed;          // the global seed

   irand::master_seed       ms;                     // a class with helper functions

   string                   server  = "top_default_server";  // TCL server name default
   int                      port    = 9999;                  // TCL server port default

   function void set_seed(int unsigned s);          // function to set the seed
      seed = s;
      ms.set_seed(s);
   endfunction

   initial begin                                    // grab values from ms at time 0
      #0;
      ms           = irand::master_seed::get_instance();
      server       = ms.server;
      port         = ms.port;
      max_target   = ms.max_target;
      seed         = ms.return_seed();
   end

   initial begin
      $value$plusargs("start_time=%d", start_time);
      $value$plusargs("loop_time=%d", loop_time);

      #(start_time);
      forever begin
         #(loop_time);

         // replacing the below line with a composition based function
         // will allow changing the Objective Function at runtime TBD
         coverage_value                 = dut.objective.get_coverage();
         ms.set_coverage_value(coverage_value);

         // set off the trigger for the simulator TCL to process
         trigger                        = ~trigger;

         if (ms.get_coverage_value() >= ms.max_target) begin
            ...
            final_report   = 1;
         end
      end
   end
```

FIGURE 4. EXCERPTS FROM RSEED_INTERFACE.SV [12]

Many SystemVerilog simulators can use TCL commands (Figure 5) to query values dynamically from the running simulation, execute functions calls of the design and pass new values into the simulation. By giving a known hierarchical location (`top.rseed_interface`) to call functions and query variables, the design becomes more portable and easier to implement on new designs. Not all simulators can call class objects functions dynamically, so a SystemVerilog interface with wrappers for those object functions was chosen to provide the software communication layer from the simulation to the simulator.

```
set server [get top.rseed_interface.server]
set port [get top.rseed_interface.port]

call top.rseed_interface.set_seed(32'd${last_seed})
```

FIGURE 5. EXCERPTS FROM RCLASS.TCL [13]

The decisions that drives the combined SystemVerilog and TCL system is a feedback loop. Which evaluates over a particular "Interval" testing if the generated stimulus improved the "Objective Function" or not. The appropriate action is then executed and the loop is repeated until the "Objective Function" is satisfied. Figure 6 describes how the process works.

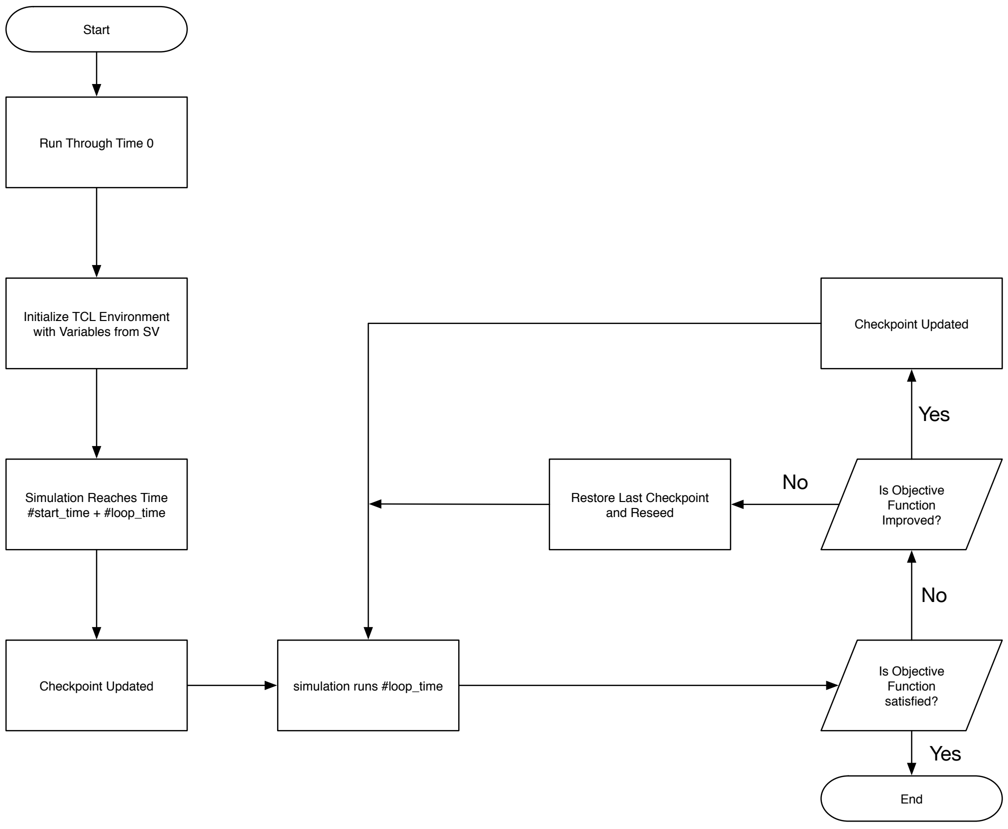
Start
Run Through Time 0
Initialize TCL Environment with Variables from SV
Simulation Reaches Time #start_time + #loop_time
Checkpoint Updated
simulation runs #loop_time
Is Objective Function satisfied?
Yes
End
No
Is Objective Function Improved?
Yes
Checkpoint Updated
No
Restore Last Checkpoint and Reseed

FIGURE 6. DECISION FLOWCHART FOR PROPOSED METHOD

Figure 7 is an excerpt from a simulation log using this implementation. It shows two seeds rejected at simulation time `17 ns` because these seeds did not increase (red) the "Objective Function" – the progress was zero in both cases. But one seed (yellow), did increase the "Objective Function" and was accepted by the simulation and time was allowed to continue.

```
UVM_INFO sv/dut.sv(14) @ 20: reporter [dut_if] AFTER drive regs a: 0 b: 3
------------------------- START eval_loop
DEBUG current simulation time is ctime : 27 ns
INFO STATUS : TCL : LOCAL REJECTED seed: 3552075441 at time: 17 ns
INFO STATUS : TCL : 27 ns : NO PROGRESS : false: 0.000000 > 0.000000 REWINDING TO CHECKPOINT {2} at 17 ns
All the Checkpoints created after checkpoint 2 are removed...
------------------------- END eval_loop
UVM_INFO sv/dut.sv(14) @ 20: reporter [dut_if] AFTER drive regs a: 1 b: 3
------------------------- START eval_loop
DEBUG current simulation time is ctime : 27 ns
INFO STATUS : TCL : LOCAL REJECTED seed: 3981500775 at time: 17 ns
INFO STATUS : TCL : 27 ns : NO PROGRESS : false: 0.000000 > 0.000000 REWINDING TO CHECKPOINT {2} at 17 ns
All the Checkpoints created after checkpoint 2 are removed...
------------------------- END eval_loop
UVM_INFO sv/dut.sv(14) @ 20: reporter [dut_if] AFTER drive regs a: 0 b: 0
------------------------- START eval_loop
DEBUG current simulation time is ctime : 27 ns
INFO STATUS : TCL : LOCAL ACCEPTED seed: 1493068099 at time: 17 ns
INFO STATUS : TCL : 27 ns : GOOD : 25.000000 > 0.000000
------------------------- END eval_loop
UVM_INFO sv/dut.sv(14) @ 30: reporter [dut_if] AFTER drive regs a: 1 b: 3
```

FIGURE 7. LOG EXCERPT OF LOCAL RUN SHOWING REJECTION OF BAD SEEDS AND ACCEPTANCE OF GOOD SEED AT TIME INTERVAL

The output of the simulation (Figure 8) is a plaintext file that documents the chosen seed applied at particular simulation times and progress of the "Objective Function" over each interval. The output file named `replicate` (Figure 8) can be fed back into the simulation to replicate a run or provide an efficient method to rerun an "Objective Function" test with its solution. Notice that the accepted seed ending in `099` (yellow) in the `replicate` file matches the time and value in the log excerpt.

```
0 ns : -1 -> 0.000000 : seed 2
17 ns : 0.000000 -> 25.000000 : seed 1493068099
27 ns : 25.000000 -> 50.000000 : seed 765542315
37 ns : 50.000000 -> 75.000000 : seed 1532361113
47 ns : 75.000000 -> 100.000000 : seed 893445949
```

FIGURE 8. EXAMPLE REPLICATE FILE TO REPRODUCE A RUN

In our simple example with the two input vectors `a` and `b` being two bits each, there is a 1 out of 4 (or $4\ of\ 16$ possibilities) chance that `a` and `b` randomize to the same number. To hit all 4 distinct matching probabilities in a row in any order is:

$$\frac{4!}{(2^4)^4} = \frac{4!}{16^4}$$

which is just below 4%. Meaning that 4% of all possible initial seeds should match the effectiveness of our optimal solution. It takes about 70 incremental iterations to find one of these solutions experimentally. Having an optimized solution could save significant time if a rerun for quality is needed or if a bug that involves an “Objective Function” is needed to be replicated in an efficient testcase. The percentages of a seed matching the optimal solution would quickly diminish with more complex goals or designs. For example, with `a` and `b` being 5 bits each, there is a 1 out of 32 or ($32\ of\ 1024$ possibilities) chance that `a` and `b` randomize to the same number. And to hit all 32 distinct matching possibilities in a row in any order is:

$$\frac{32!}{(2^{10})^{32}} = \frac{32!}{1024^{32}}$$

It is likely that no simulation seed would optimally solve the example DUT sequence of stimulus by chance with that probability. But, it is possible with a relatively small number of iterations to solve.

## V. Scaling

The intriguing part of this idea is that this approach is scalable. Since the goal is abstracted to be an “Objective Function” and the feedback is abstracted to be the dynamic seed, it is possible to coordinate the “Interval” feedback loop to include multiple simulations working on the same test in parallel. The only information that has to be communicated between the parallel simulations is the optimal dynamic seed over a time “Interval”. Once an optimal seed is discovered, all of the parallel simulations adopt that seed for the “Interval” and start the next “Interval” (Figure 9). Multiple simulations and servers can be working on the same test each exploring different paths cooperatively searching for the solution to the “Objective Function”. The implications of an entire server farm exploring the stimulus state space could yield significant performance gains towards achieving the “Objective Function”.

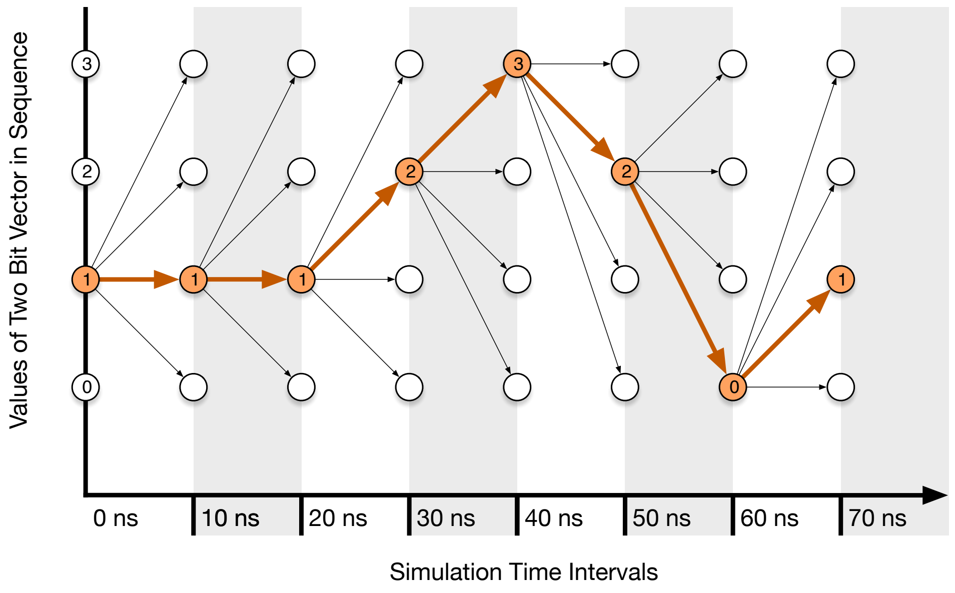
Values of Two Bit Vector in Sequence
3
2
1
0
0 ns
10 ns
20 ns
30 ns
40 ns
50 ns
60 ns
70 ns
Simulation Time Intervals

Figure 9. Choosing Optimal Seed Dynamically During Simulation and Synchronizing Parallel Attempts

In order to facilitate multiple simulators working on finding the optimal seed for a time interval, there needed to be a process outside the simulation to coordinate the evaluation and acceptance of new seeds. The solution chosen (Figure 10) was to use a lightweight TCL socket server that would accept proposed seeds from the simulations and then immediately reply if that seed was accepted or not. This server is operational and is invoked with `make server`. The default port is `9000`, but can be changed by adding an argument to the makefile command line such as `make server PORT=9001`.

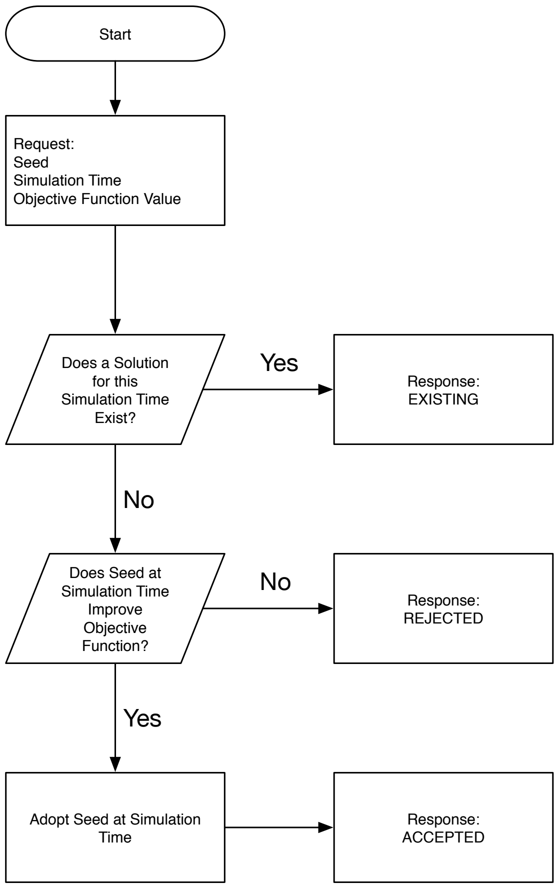
Start
Request:
Seed
Simulation Time
Objective Function Value
Does a Solution for this Simulation Time Exist?
Yes
Response: EXISTING
No
Does Seed at Simulation Time Improve Objective Function?
No
Response: REJECTED
Yes
Adopt Seed at Simulation Time
Response: ACCEPTED

FIGURE 10. SERVER DECISION TREE

An `EXISTING` response tells the simulator that a seed for its current time step has been chosen already based off of previous inputs. This tells the simulator it should rewind back to the previous simulation time and apply the provided seed given in the socket response.

A `REJECTED` response tells the simulator that no acceptable seed for its proposed time has been found. This tells the simulator it should rewind back to the previous simulation time and attempt a new seed of its own choosing.

An `ACCEPTED` response tells the simulator that the seed it proposed was beneficial towards the "Objective Function" and that the simulation should continue with that seed. The `ACCEPTED` seed is then saved in the TCL server and given to other simulators that request a decision at that simulation time.

Focusing simulation resources on reaching hard to reach verification goals requires special attention to the active constraints and an intimate knowledge of how that stimulus is interpreted by the design and testbench. The level of abstraction is high and requires in-depth knowledge. More simulation cycles help and more tests help to reach hard to reach verification goal. But there are limits. While it is conceivable that a single test running for infinite time would eventually hit all possibilities; it would make a situation that would require infinite time to recreate that failure for debugging, which is problematic. Tests that take days to run are not generally practical, in the sense of not being able to quickly debug them or to understand how they got to a particular state. Therefore, the alternative is to launch a large number of reasonable length random tests to try to get the verification goals satisfied – which is the approach used today.

Figure 11 shows the probability histogram of a model [14] of this algorithm showing how many iterations it took to find the optimal solution for our example DUT. This histogram is normalized, meaning that the area integrated under the curve is equal to one. Running the model linearly with no rewind and the proposal of this paper to use rewind if a solution does not advance the objective function, has the same iteration count and processing overhead in this case. We can see most solutions were in a range centered around 4000 iterations with a wide range of solution iterations between 2000 and 6000. Figure 12 show the normalized cumulative distribution of the same data; it is possible to see that with a probability of 0.5 (50%) the median simulation model achieved its "Objective Function" in 4000 iterations and 80% achieved the "Objective Function" in 5000 attempts.

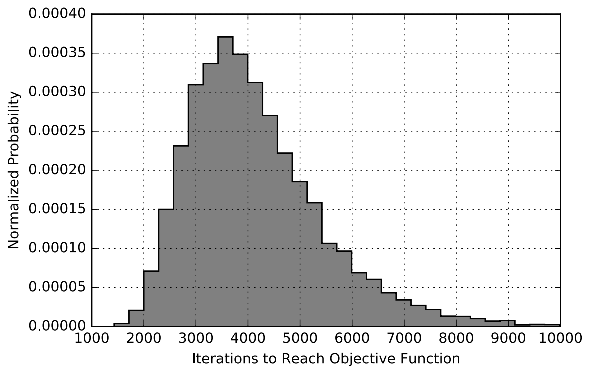

FIGURE 11. NORMALIZED PROBABILITY HISTOGRAM OF ITERATIONS TO SATISFY THE DUT WITH WIDTH=5 LINEARLY

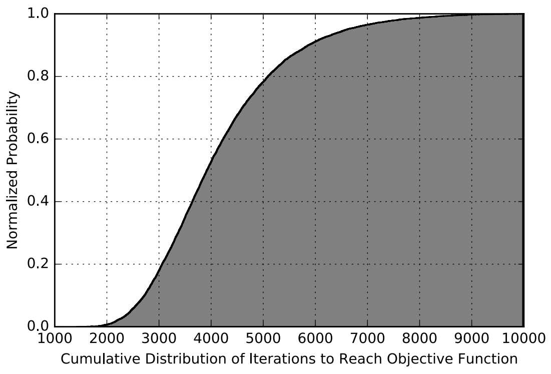

FIGURE 12. CUMULATIVE DISTRIBUTION OF ITERATIONS TO SATISFY THE DUT WITH WIDTH=5 LINEARLY

Figure 13 shows the probability histogram of a model of this algorithm using the same setup but using 5 parallel simulations to find an optimal solution. We can see that most solutions were centered around 1000 iterations – with less variance than the linear histogram. The solution can be found faster by coordinating the optimal found seeds using this proposed method. The method also reduces the variation in finding a solution, which can give assurances that a solution is likely to be found within a certain amount of time.

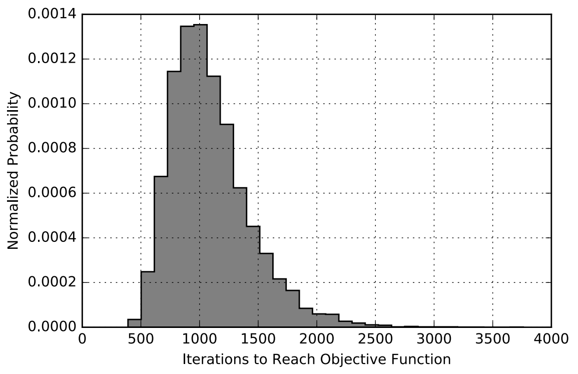

FIGURE 13. NORMALIZED PROBABILITY HISTOGRAM OF ITERATIONS TO SATISFY THE DUT WITH WIDTH=5 WITH 5 PARALLEL SIMULATIONS

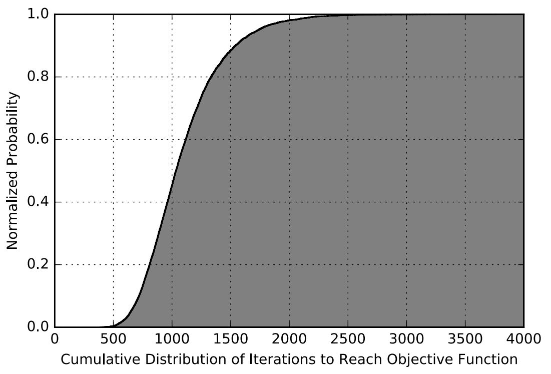

FIGURE 14. CUMULATIVE DISTRIBUTION OF ITERATIONS TO SATISFY THE DUT WITH WIDTH=5 WITH 5 PARALLEL SIMULATIONS

The proposal is that a smaller number of tests should be run, but those tests can be more exhaustive and succinct in their approach to the problem. The smaller number of tests can use the same number of simulation resources as before, but they are being used to explore in parallel many possibilities. There still exists bug hunting in this process – as tests will still fail in the conventional way and can be debugged with conventional methods. If this method is extended to 20 parallel simulations the number of attempts in the final solution is further reduced (Figure 15 and Figure 16).

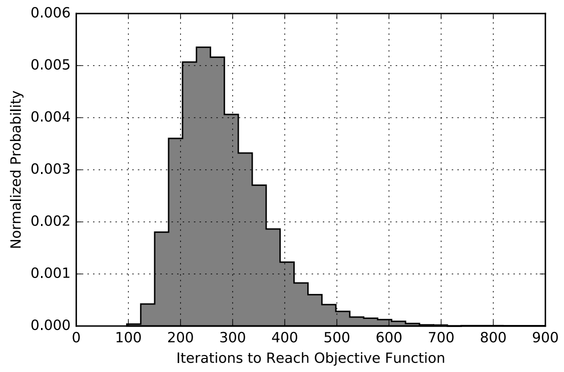

FIGURE 15. NORMALIZED PROBABILITY HISTOGRAM OF ITERATIONS TO SATISFY THE DUT WITH WIDTH=5 WITH 20 PARALLEL SIMULATIONS

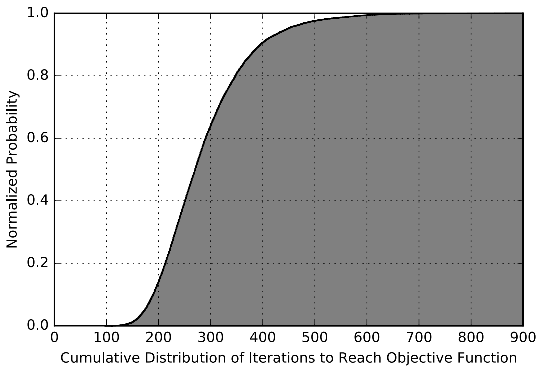

FIGURE 16. CUMULATIVE DISTRIBUTION OF ITERATIONS TO SATISFY THE DUT WITH WIDTH=5 WITH 20 PARALLEL SIMULATIONS

Comparing the normalized probabilities shows how many parallelized iterations it takes to solve the example DUT. Figure 17and Figure 18 show the composites of the different parallel and linear approaches. In each case, the median number of parallel iterations is reduced and the distribution of iterations is reduced. The reason why adding more parallel simulations improves the odds of solving this DUT is that we are simply guessing more with each iteration. Each iteration brings with it not just a single guess, but five or twenty attempts. These parallel attempts are present for every time step and add up to fewer iterations needed and also less wall-clock time to find a solution. The computing resource is the same however, we have just parallelized the search by allowing multiple simulations to work together to try alternative solutions to the same test.

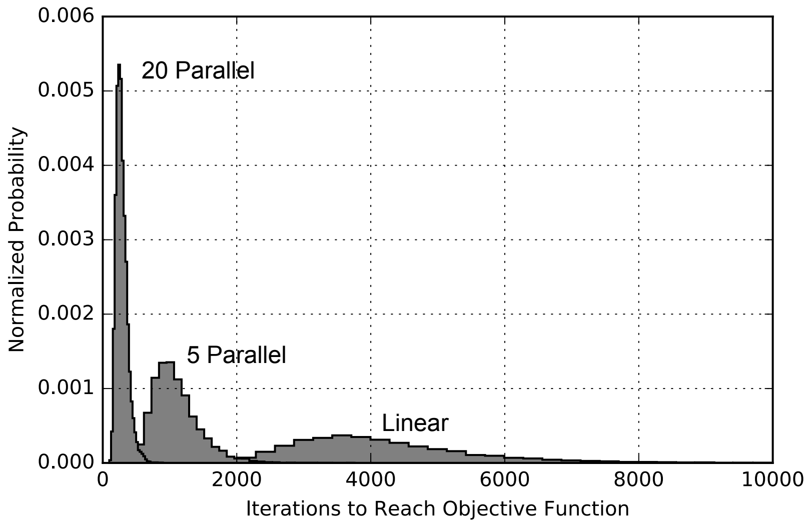

FIGURE 17. NORMALIZED PROBABILITY HISTOGRAM COMPARING NUMBER OF PARALLEL SIMULATIONS

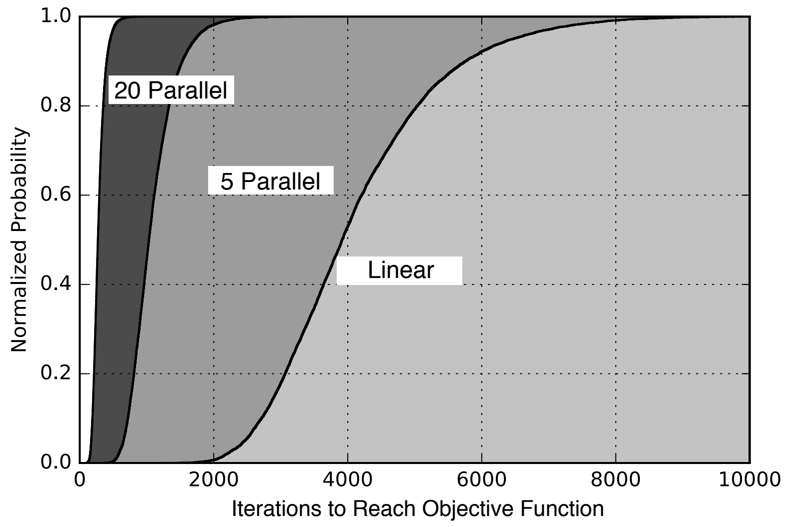

FIGURE 18. CUMULATIVE DISTRIBUTION COMPARING NUMBER OF PARALLEL SIMULATIONS

In order to validate our algorithm modeling, we can run provided makefile and produce actual SystemVerilog simulator results. The deciding factor is how many attempts per unit time we can produce and validate. Our simulation model predicted a 4x speed up when moving from "Linear" to "5 Parallel" as demonstrated in Figure 18 when comparing the 0.5 probability. When using the SystemVerilog simulator and the provided implementation [4] we can demonstrate a 5.7x speed up between "Linear" and "5 Parallel". The information gained from Figure 19 is that the implemented method demonstrates promising scaling when done with a SystemVerilog simulator. There is correlation between the SystemVerilog simulation performance results and the model results.

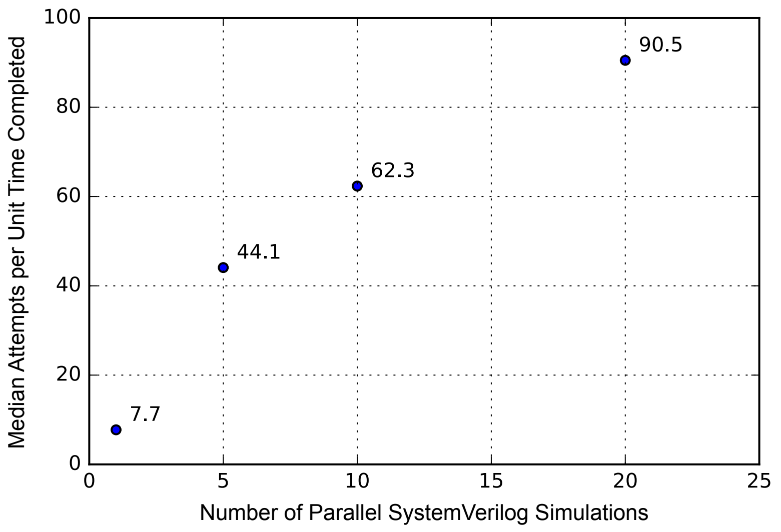

FIGURE 19. MEDIAN ATTEMPTS PER UNIT TIME COMPLETED WITH INCREASING PARALLEL COOPERATIVE SIMULATIONS

## VI. FACILITATING UVM RESEED

```
class ms_sequence_item extends uvm_sequence_item;

   irand::master_seed ms;
   bit ms_enable = 1;

   function new();
      ms = irand::master_seed::get_instance();
   endfunction

   function void pre_randomize();

      if (ms_enable) begin
         ms_run();
      end

   endfunction

   function void ms_run();

      string inst_id;
      string type_id;
      string type_id2;

      if (get_full_name() == "") begin
         inst_id  = "__global__";
      end else begin
         inst_id           = get_full_name();
      end

      type_id            = get_type_name();
      type_id2           = {uvm_instance_scope(), type_id};

      if (uvm_pkg::uvm_random_seed_table_lookup.exists(inst_id)) begin
         if (uvm_pkg::uvm_random_seed_table_lookup[inst_id].seed_table.exists(type_id2)) begin
             uvm_pkg::uvm_random_seed_table_lookup[inst_id].seed_table.delete(type_id2);
         end
      end

      reseed();

   endfunction

endclass
```

FIGURE 20. SETTING UP UVM RESEED() IN ENV_PKG.SV

The built-in UVM function `reseed()` (Figure 20) will reseed an object based off of a hash created from: `uvm_pkg::uvm_global_random_seed`, `get_type_name()` and `uvm_instance_scope()` which are all built in UVM functions and variables. Whenever we want to insert a new seed into the system, we first set `uvm_pkg::uvm_global_random_seed`. Then, we rely on the `pre_randomize()` method we added to our sequence item to run the function `ms_run()` (*short for master seed run*) that does the necessary work to purge an existing entry, if present, in the UVM `seed_table`. UVM tries to avoid collisions in seeds for objects because if two objects had the same seed they would randomize the same way, and would not truly be random anymore, but correlated. The `seed_table` insures that an attempt is made to randomize every object in a deterministic way based off of some intrinsic properties of the UVM object. The method described allows for the built-in UVM method `reseed()` to function as we want – as a method to dynamically reseed an object. If we did not remove the entry for the object in the `seed_table` we would not reseed the object, but simply return its original seed since `reseed()` looks to see if something is extant in its database before regenerating its seed hash. The other feature to note, is that it is possible to avoid the `reseed()` behavior. In Figure 20, a control bit `ms_enable` is used to allow the user to exclude an object from reseeding.

## VII. Limitations

This method has a number of limitations. For example, with infinite possibilities parallelizing concurrent exploration will not necessarily improve the probability of finding a low probability path. This method can improve the chances of finding those probabilities, but does not guarantee it.

This method relies on the simulation to be in control of its state when a checkpoint is created. If the simulation relies on external C programs controlled via a SystemVerilog DPI (Direct Programming Interface) or some other VIP that does not obey the simulation checkpoint, this method will not work. If there are external programs that do not obey simulator checkpointing, the state will not be consistent when a checkpoint is restored and the simulation will be invalid.

The choice of the simulation interval is up to the user to determine the optimal value. Should it be one clock cycle, as in our example DUT? For this example, one clock cycle was the optimal value for the interval, since we also had no state within the DUT. The optimal value maybe be many clock cycles and vary per design. The interval is controlled by the user with the SystemVerilog plusarg `+interval_time=X`; which in the proposed implementation, is in simulation time units. The other value that the user must determine is the the `+start_time=X` which marks when the method starts comparing coverage and engaging the rewind behavior. The choice of when to start the algorithm is also dependent on the design and must be chosen by the user.

This method also can get stuck in simulation loops where the "Objective Function" does not achieve it target value. To accommodate for this, there are two plusargs to guide the simulation. One is `+max_attempts=x` (default 1000000) which is the total number of attempts to try before exiting the simulation. Since the conventional UVM timeout mechanisms are based on simulation time, we need something to watch out for infinite loops that occur over a single time period. The second is `+max_objective=x` (default 100) which is a number that is the acceptable value of the "Objective Function" for the test to finish. Early in a design the "Objective Function" goal may be reduced on a per test basis. At the end of the design the goal can be brought up to its true final value. Also, note that the "Objective Function" can be any number and this plusarg allows for altering that value at runtime.

The method is susceptible to falling victim to local maxima. The algorithm provided simply looks at possibilities from a point in time and chooses the first one that improves its "Objective Function". The first solution that improves the "Objective Function" with that small interval time window may put the design in a state that has no simple exit. The algorithm would then get stuck, perhaps even systemically.

The resultant dynamic seed solution, which is provided as a `replicate` file (Figure 8), documents the seeds and times those randomization seeds are to be applied, is likely fragile to design and environment changes. While it is possible to replicate a simulation using the same inputs and the dynamic seed solution, it is susceptible to DUT changes and environment changes. The extent relies very much on what was changed. It is more fragile than simple randomization stability that UVM tries to accommodate to the best of its ability. Since the method currently is based off of simulation time, changing how long the reset is applied would invalidate the dynamic seed solution – wherein a UVM environment, that only relied on a single seed, may be able to still produce the same result.

The simulation log file produced during the process is not useful in its current form; as the simulation log contains many possibilities that were never chosen. It is possible to post-process the simulation log to remove log entries relating to paths that were not taken. This has not been done, but is left as a future exercise. However, once the process is complete a simulation may be rerun with its `replicate` file which will produce an accurate log.

## VIII. SUMMARY

Constrained Random simulation and coverage collection today relies on large numbers of tests and merging coverage across those tests. This paper made the argument that there may be an alternative to this approach that may result in faster automated coverage closure, an efficient final stimulus solution and proposed higher quality of coverage. This paper documented an existing reference implementation of this proposed method and shared some of its current and future design choices – as well as its limitations. The performance benefits for reaching verification goals on a simple example testcase was compared to conventional methods. The results propose that this method can improve the time to reach an "Objective Function" in a single test, probabilistically, by parallelizing and coordinating multiple trials. It was shown that the method demonstrated scaling from a single simulation, to five parallel simulations, and twenty parallel simulations each improving the iterations and wall clock time needed to achieve "Objective Function" closure. A series of graphs illustrated probability density of the modeled and simulated possibilities versus the different approaches. The model and simulator results had a high degree of correlation.

One end result of this method was a novel approach the treatment of seeds in UVM testbenches. A file was created during the proposed process that documented SystemVerilog randomization seeds that are applied to the simulation at specific times that resulted in an efficient solution. This file allows for an efficient test that covers the "Objective Function" and provides an efficient way to replicate the simulation. It was mentioned that this recipe may be more fragile for design changes than the current SystemVerilog seed methodology as a drawback.

The goal of proving that this method provides higher quality of coverage is not proven. It is only shown that this method can generate stimulus that may mimic the results of Graph-Based Stimulus, but in a very different way. It works in perhaps the opposite direction of Graph-Based Stimulus, in that it sets the design goal and then directs the simulation to try to reach that goal in an iterative approach. While when following Graph-Based Stimulus, the design goal may be set and then the user tries to figure out what stimulus would reach that goal.

The source code was provided and attempts to be as simulator agnostic as possible, making it easier to port to all simulators. Limitations of the approach were enumerated.

## IX. FUTURE WORK

There are solutions to some of the limitations to this method that are readily available. At its heart, this is an optimization algorithm that strives to find stimulus that would improve its "Objective Function". The approach was to expand upon randomization techniques which are used in a number of fields. Absolutely solving a problem is ideal, but is not possible in many fields and randomization optimization techniques are common tools to explore a problem perhaps even deeper than possible with exact methods. For example, the classical traveling salesman problem does have algorithms that absolutely solve it for small numbers of nodes – simply use a brute force search to evaluate every possibility. But the complexity of the problem quickly expands and full proofs can no longer be proved.

One known approach to attempt to avoid local maxima – that this method can have problems with – is to utilize a method known as "Optimization by Simulated Annealing" [15]. This method allows more randomization of the solution up front and gradually reduces the randomization as it gets closer to the solution. In other fields of study the random variable may be distance or temperature. One way to apply simulated annealing to our verification problem is to treat our randomization variable as simulation time. At the beginning of the simulation, make the interval be a large simulation time and let many simulations be evaluated. Then take the best available solution to the "Objective Function". Reduce the interval time for the second iteration to a smaller value and repeat. Continue reducing the interval time as the "Objective Function" approaches its goal. This could lead to a solution less susceptible to local maxima problems.

## X. ACKNOWLEDGEMENT

A bit thanks goes to Tom Murphy VII for an interesting paper [5] on a novel approach to creating an AI for playing the Nintendo game Super Mario Bros. I thank my employer Intel Corporation for sponsoring my attendance at DVCon to share this paper.

## XI. References

[1] IEEE Computer Society and the IEEE Standards Association Comporate Advisory Group, IEEE Standard for SystemVerilog 1800-2012, New York, NY: IEEE, 2013.

[2] Cadence, "Incisive Version 14.10," [Online]. Available: http://www.cadence.com.

[3] Mentor Graphics Corporation, "Questa Sim Version 10.4c," [Online]. Available: https://www.mentor.com.

[4] Synopsys, "VCS Version L-2016.06-1," 2016. [Online]. Available: http://www.synopsys.com.

[5] T. Murphy VII, "The First Level of Super Mario Bros. is Easy with Lexicographic Orderings and Time Travel . . . after that it gets a little tricky.," in *Sigbovik*, Pittsburgh, 2013.

[6] T. Murphy VII, "Computer program that learns to play classic NES games," 1 4 2013. [Online]. Available: https://www.youtube.com/watch?v=xOCurBYI_gY. [Accessed 8 11 2016].

[7] Free Software Foundation, "GNU General Public License," 29 6 2007. [Online]. Available: https://www.gnu.org/licenses/gpl-3.0.en.html. [Accessed 8 11 2016].

[8] GitHub, Inc., "GitHub," 2016. [Online]. Available: https://github.com.

[9] E. Nelson, "Improving Constrained Random Github Project," 2016. [Online]. Available: https://github.com/tenthousandfailures/improving-constrained-random.

[10] N. Le and M. Andrews, "Efficient Bug-Hunting Techniques Using Graph-Based Stimulus Models," in *Design and Verification Conference*, San Jose, 2016.

[11] E. Nelson, "dut.sv," 2016. [Online]. Available: https://github.com/tenthousandfailures/improving-constrained-random/blob/master/sv/dut.sv.

[12] E. Nelson, "rseed_interface.sv," 2016. [Online]. Available: https://github.com/tenthousandfailures/improving-constrained-random/blob/master/sv/rseed_interface.sv.

[13] E. Nelson, "rclass.tcl," 2016. [Online]. Available: https://github.com/tenthousandfailures/improving-constrained-random/blob/master/tcl/rclass/rclass.tcl.

[14] P. Jupyter, "Jupyter Notebook," 2016. [Online]. Available: http://jupyter.org. [Accessed 14 12 2016].

[15] S. Kirkpatrick, C. D. Gelatt Jr. and M. Vecchi, "Optimization by Simulated Annealing," *Science,* vol. 220, no. 4598, pp. 671-680, 13 May 1983.