Want Functional Coverage Closure? Don't kneel to the Almighty Random Constraint Solver

Jeremy Ridgeway

Broadcom, Inc.

Fort Collins, CO USA

ABSTRACT

Random variable constraints are the crux to both constrained random simulation and functional coverage. The SystemVerilog standard defines random variable constraints as declarative syntax. That is, the constraint on any variable, once elaborated, is not expected to change significantly during simulation. As such, functional coverage goals are at the mercy of the random constraint solver. We present an approach for an expressive and dynamic random constraint library to nudge control back to the verification engineer. With our library, we achieve dynamic whole constraint replacement (e.g., change from inside to dist), constraint augmentation (e.g., logically AND a new constraint (! inside { [1:9] }) with existing constraint inside { 0, [1:9], 10 }), infinitely nested constraints (inside { inside { ... }, dist { ... }, ... }), as well as a new sequence constraint to automatically iterate over value or nested constraints (seq [4, inside { ... }, dist { ... }, ...]). Furthermore, because constraints are composed as a string, all constraints may be provided either during simulation or a priori on the command-line. Finally, each random variable in our approach maintains a constraint stack to allow for push/pop of constraints during critical scenarios. With this methodology, randomization can work alongside functional coverage to achieve project goals.

Table of Contents

Table of Figures

1. Introduction

In simpler days a testplan matched one-for-one to the testbench: one testing scenario in the plan equaled one test run in regression, and a passing test meant the testing scenario had been verified. Today, constrained random testbenches are the standard for verification of larger, more complex IP and subsystems. This has led to a reversal of correlation: one test run equals many testing scenarios selected during simulation through constrained random analysis. We ascertain verification status by test pass/fail rate and summation of functional coverage groups hit -- where, roughly, one testing scenario from the testplan matches one functional coverage group reporting, from within the testbench, that the scenario was simulated. We have a gap, however, between testing scenarios we want to cover (testplan) and testing scenarios actually covered through random analysis (reality). The current solution is simply to run the test more times hoping randomization will eventually achieve the testplan.

With our SystemVerilog generic random variable container classes, we aid coverage by nudging the random analysis towards our goals first. The random variable itself and/or the testbench can easily manipulate the current random constraints. Essentially, we can block random streaking by reactively changing the constraint on a random variable based on previous random selection. If scenario A was selected, we can block scenario A from being selected again. Then, once all desired scenarios have been randomly selected, we can reverse the incremental blocking to allow for full random range. This approach differs from the common random constraint usage models today that focus on instantaneous random selection with no history. Since the SystemVerilog coding language itself provides little capability for constraint manipulation during simulation, the common approach prevails, loading server farms with regressions hoping to cover the testplan. Our approach overcomes many of these hurdles in order to help drive the constrained random testbench towards functional coverage closure and testplan completion without loss of random selection and its benefits.

2. Background

As verification engineers, we are constantly fighting quality testing versus testplan completion time. Of course we want both. We cannot feasibly deliver RTL without covering every scenario in our testplan. But, every scenario in our testplan may not uncover potentially high quality bugs found by constrained random simulation. Then again, constrained random verification comes with its own costs. Therefore choosing the appropriate verification approach or approaches is key to a successful verification program. We see three testing approaches commonly used in verification: directed, directed-random, and constrained random.

Directed testing enumerates all testing scenarios in the testplan and correlates them one-for-one to tests implemented in the test suite. When a directed test is run and passes we know the feature is verified. When a directed test fails we may have uncovered a RTL bug. When a directed test is not run then we know *nothing* about the feature. That is, the directed test must be run in order to exercise the test feature. As such, functional coverage is not required in directed testing. The testplan and correlated test suite is sufficient. Verification status can be surmised from a single regression. And, regression time is correlated to size of test suite and server and license resources.

Directed-random testing is similar in architecture to directed testing but expanded to cover a class, or set, of testing features. Similar to directed testing, when a directed-random test is not run then we know *nothing* about that class of features. Functional coverage may be necessary to report specific features simulated within the class. In this case, the test itself may sufficiently provide the context for the cover point coverage. That is, cross coverage may not be necessary because the

cover point is implicitly crossed with the context of the directed-random test. Regression time is still correlated to the size of the directed-random test suite and server and license resources. However, merging functional coverage between regressions may be required to achieve complete coverage.

Constrained-random verification (CRV), however, requires a completely different architecture than directed or directed-random. In CRV, nearly all testplan scenarios are targeted for just one test. The focus is on testbench development with autonomous drivers and checkers and monitors, also known as universal verification components (UVCs). Each testing scenario from the testplan is allocated within individual UVCs. The test, itself, just bootstraps the environment as a whole.[1]

The CRV architecture negatively affects both functional coverage and regression time. Without functional coverage we cannot know what scenarios from the testplan are simulated. Furthermore, we have little to no control over the random constraint solver to achieve the testplan goals. As such, it is possible to run regression for days without achieving desired functional coverage. It is, therefore, critical to develop a functional coverage model that reports no more or less than the testplan, including appropriate coverage context. In other words, cross coverage is necessary to show that a scenario is hit within the context of other simulation state variables (simulation configuration). There *is* value in reporting possible valid combinations have been simulated (and verified) – indeed that is what CRV will provide. However, functional coverage focus must remain on the testplan otherwise timetables can become untenable.

In this paper we propose an aid to regression time, and functional coverage model completion, by

(a) Constraining random variables according to functional coverage, and
(b) Blocking covererage bins hit for subsequent randomizations.

This approach is based purely in a class representation of the random variable. We do not query any functional coverage group or bin. Instead, we have developed an approach to SystemVerilog constraints that enables easy and autonomous manipulation of the constraint itself. We will develop a working knowledge of our approach in Section 3, Theory. In Section 4, we present details about our generic random constraint container class library available. In Section 5, we dissect an example from the abstract using the LVM random constraint library. In Section 6, we discuss some work yet to be done on the class library, both within our company and with individuals within the greater verification community. We summarize and conclude in Section 7.

3. Theory

Consider the following generic Universal Verification Methodology (UVM) sequence item with four kinds of random variables (addr differs from size because it requires an additional constraint to ensure the lower two bits are always assigned zero), Listing 1.

Listing 1: Generic sequence item nugget_of_data.

```
1  class nugget_of_data extends uvm_sequence_item;
2    rand enum { CONTROL, SIDEBAND, DATA } code;
3    rand bit[31:0] size; // used to constraint data queue size
4    rand bit[31:0] addr; // addr[1:0] == 0
```

[1] We advocate the asymptotic ideal of one good test and one good plus error injection test. Of course, constraining the testbench via directed-style testing is still required but not the focus.

```
5     rand byte data[$];    // assume max of 1024 bytes
6   endclass
```

There are four kinds of random variables in the sequence item. The inline enumeration limits the number of selectable values in the code variable. However, the size variable is too large of a random range to feasibly cover in simulation. This variable must be broken into significant contiguous ranges for randomization and functional coverage. Suppose that the size of the data queue affects the operation of the device under test (DUT). Random constraints on size may guide the variable to exercise these differences. The address variable is similar in scope to the size variable, except that it has an added constraint on the lower two bits. However, adhering to the CRV guideline avoiding verification in the covergroup, we may disregard the constraint for addr's covergroup – a stimulus checker or assertion may ensure valid address assignment. Finally, we will assume all byte values in the data queue are equal. We will have neither random constraints nor functional coverage on the data values, themselves.

Let us consider the following simplistic testplan governing nuggets of data generation, Table 1.

Table 1: Initial disjoint testplan for nuggets of data.

Scenario ID	Scenario Description
NUGGET_1	All code selections must be verified.
NUGGET_2.1	Nugget sizes may be of length zero (no data) or any size up to a maximum of 1024 bytes.
NUGGET_2.2	For nugget sizes less than 512 the DUT will apply no backpressure.
NUGGET_2.3	For nugget sizes 512 bytes or larger, the DUT may apply backpressure in order to process the nugget data.
NUGGET_3	Nugget addresses must be DWord aligned.

Notice that Table 1 implies no correlation between the three random variables in a nugget of data. For example, all nuggets of data may be of any size [0:1024] bytes regardless of the control code. We will apply CRV architecture on the DUT interface for nuggets of data and develop functional coverage to match.

3.1 Randomization on a restricted range – nugget code

The random variable in Listing 1, line 2, nugget code, is limited to one of three options, as defined by the inline enumeration. If the testplan in Table 1 *had indicated* an overriding protocol on code, sideband, or data nuggets, then class nugget_of_data should really extend to nugget_code, nugget_sideband, and nugget_data, respectively, constraining the code according to the class type. However, the testplan did not imply a protocol. As such we may reasonably assume all control codes are equally valid and disjoint from the other variables in the class. A feasible covergroup is shown in Listing 2, below. We have intentionally labeled the covergroup with the scenario ID from the testplan, Table 1.

Listing 2: SystemVerilog functional covergroup for variable nugget code coverage goal.

```
1   covergroup cg_nugget_1;
2     coverpoint code {
3       bins code__0 = { CONTROL };
4       bins code__1 = { SIDEBAND };
```

```
5       bins code__2 = { DATA }; }
6   endgroup
```

When cg_nugget_1 is instantiated within the class nugget_of_data, the coverpoint label, code, will bind to the random class variable, code. At covergroup sample, the current value stored in the code will be sampled and appropriate bin count incremented.

No random constraint on code is required due to its limited random range. With no constraint, randomization will consider each value uniformly probably – a flat curve. An alternative approach is to explicitly indicate each value in an inside set, as in Listing 3, lines 1-2.

Listing 3: Two SystemVerilog constraints for determining statistical curves on variable nugget_code.

```
1   constraint nugget_code_uniform {
2     code inside { CONTROL, SIDEBAND, DATA }; }
3   constraint nugget_code_user {
4     code dist { CONTROL := 10, SIDEBAND := 20, DATA := 70 }; }
```

In Listing 3, lines 3-4, the user-defined distribution is shown to control random selection on the selected code value. It is also valid to control the code selection during simulation by enabling one constraint and disabling the other, then reversing the constraint mode. Constraint mode enable/disable is one way SystemVerilog affords dynamic random manipulation. However, the nugget_code_user constraint is not required as per the testplan in Table 1. As such, we will only consider the nugget_code_uniform constraint.

An equal and alternate composition of the uniform constraint on the nugget code variable is shown in Listing 4, where we employ a queue of values within our inside set constraint.

Listing 4: Inclusion constraint formed from dynamic set membership rather than explicit values.

```
1   int nugget_code_values[$] = { CONTROL, SIDEBAND, DATA };
2   constraint nugget_code_uniform {
3     code inside { nugget_code_values }; }
```

In Listing 4, line 1, we declare the nugget_code_values queue as a *non-rand* class member variable and populate it with all nugget code enumeration values. The queue as a non-rand variable will ensure the queue size will not change during randomization. According to SystemVerilog standard, the nugget_code_uniform constraint converges on one value within the queue with equal probability.

Two questions emerge with the queue within an inside constraint. What happens when we populate the queue with duplicate entries? What happens when the queue contains fewer than all enumeration values?

The nugget_code_uniform constraint with duplicate values in the queue has no bearing on eventual outcome of randomization. Consider, first, the population from Listing 4, line 1.

```
((code == CONTROL) || (code == SIDEBAND) || (code == DATA))          (1)
```

From Eq. (1), it is clear that a single free variable, code, exists in three disjoint Boolean predicates. Furthermore, if Eq. (1) is to resolve to true, then that free variable must eventually resolve to one of the three enumerated values: CONTROL, SIDEBAND, or DATA. Now, consider if the queue were to contain a duplicate entry, SIDEBAND as shown below.

Listing 5: Duplicate value set membership has no bearing on random outcome.

```
1  int nugget_code_values[$] = { CONTROL, SIDEBAND, DATA, SIDEBAND };
```

Transliterating Listing 5 to Boolean equation (2), below, we can clearly see that the overall constraint has not changed.

```
((code == CONTROL) || (code == SIDEBAND) || (code == DATA) ||
                      (code == SIDEBAND) )                        (2)
```

In Eq. (2), a single free variable, code, still exists, though it now exists in four disjoint Boolean predicates. However, if Eq. (2) is to resolve to true, then that free variable must eventually resolve to one of the *three* enumerated values. The fact that one of the three values is duplicated is of no consequence. In fact, the random constraint solver will build a Boolean equation in conjunctive-normal-form (CNF) and that excludes *all* duplicates. That is, from Eq. (2), only a single unique instance of code will exist and only a single unique predicate representing (code == SIDEBAND) will exist. Eq. (3), below, labels each predicate from Eq. (2).

```
((      a       ) || (      b       ) || (      c      ) ||
                     (      b       ) )                          (3)
```

Clearly, two predicates labeled **b** is not required. As such, Eq. (3) will simplify to Eq. (1) and the uniformity in nugget_code_uniform constraint in Listing 4 is preserved[2]. Duplicate entries in the queue have no bearing on eventual random outcome.

Non-existence of enumerated value in the inside set queue *will* have an effect on the random outcome. Consider if the queue population below.

Listing 6: Non-existance of value in set membership does affect random outcome.

```
1  int nugget_code_values[$] = { CONTROL, DATA };
```

In Listing 6, the SIDEBAND code value is not listed at all. As such, the Booelan equation composed by the random constraint solver will ignore the SIDEBAND code value and its Boolean predicate representative label, b.

```
((code == CONTROL) ||                    (code == DATA))          (4)
((     a          ) ||                    (     c      ))          (5)
```

This is a desired manipulation of the constraint. With the inside set constraint on a queue of values we may easily and dynamically affect the overall random outcome. For example, if SIDEBAND were selected during randomize then during post_randomize we may simply remove SIDEBAND from the queue to block it from consideration in a subsequent randomize. With this approach we may achieve cg_nugget_1 coverage closure, from Listing 2, in just three randomizations. All subsequent randomizations would uniformly select any value for code as an empty constraint, Ø, represents an unconstrained variable.

An alternative approach is to not remove valid values but, in parallel, use a not inside constraint with its own queue.

[2] The spacing in Eq. (3) is kept to align with original formula in Eq. (2); this appears throughout the paper.

Listing 7: Exclusion constraint formed by dynamic set membership.

```
1  int nugget_code_exclude_values[$] = { };
2  constraint nugget_code_exclude {
3    ! ( code inside { nugget_code_invalid_values } ); }
```

Since we are looking to exclude values only after they have been chosen, the initial state of the invalid values queue is empty, representing an empty constraint. Suppose, in the first randomize call the SIDEBAND code value is chosen from the nugget_code_values queue based on the nugget_code_uniform constraint. Suppose, too, that we push the SIDEBAND code value into the nugget_code_exclude_values queue in the post_randomize() function. In the case that both nugget_code_uniform and nugget_code_exclude constraints are enabled in the next randomize call, the constraint solver will compose Boolean equation (6), where the first clause is from nugget_code_uniform constraint while the second is from nugget_code_exclude constraint.

```
((code == CONTROL) || (code == SIDEBAND) || (code === DATA))
                                &&                                  (6)
                    (!(code == SIDEBAND))
```

A CNF *clause* is the disjunction of Boolean predicates (i.e., product-of-sums). Eq. (6) has two clauses. We have labeled each Boolean predicate from Eq. (6) in clause Eq. (7a) and clause (7b).

```
((      a      ) || (      b      ) || (      c      ))      (7a)
                          &&
                    (     !b      )                          (7b)
```

Clause (7b) is a *unit clause* and has special meaning to the constraint solver. In order for the overall formula to resolve to true, unit clause (7b) must also evaluate to true. That is, **b** == false and, thus, the variable code cannot be selected as SIDEBAND, only predicates **a** or **c**, from Eq. (7a), are possible. In this fashion we can continue to exclude the random selection until constraint solver failure.

3.2 Randomization on a contiguous inclusive range – nugget size

Unlike the restricted random range on the nugget code variable, the 32-bit size random variable's random range is too large to possibly cover every selection. Instead, often contiguous ranges of values within the larger range are uniformly selected.

Again, we wish to compose as random constraint to mimic the covergroup on the same variable. Assuming the nugget size is limited to the absolute range 0-1024 bytes, we might compose cover bins as in Listing 8. Here, both extremities, 0 and 1024, are represented in unique cover bins. For the remaining range, we have broken them into less-than-512-bytes and 512-bytes-or-greater, bins size__1 and size__2, respectively. We make the assumption here that these ranges are significant to the design RTL under verification.

Listing 8: SystemVerilog functional covergroup for variable nugget size coverage goal.

```
1  covergroup cg_nugget_2;
2    coverpoint size {
3      bins size__0 = { 0 };            // minimum valid value
4      bins size__1 = { [1:511] };      // lower range
5      bins size__2 = { [512:1023] };   // upper range
6      bins size__3 = { 1024 }; }       // maximum valid value
7  endgroup
```

Unfortunately, we have no facility to indicate these contiguous ranges in the same positive and negatives that we used for nugget code, from Listing 4 and Listing 7. That said, there is no requirement that the inside set queue must be on integral values. Instead, we could represent the each predicate as a class instance reference. The remainder of this section is devoted to that construction.

From the covergroup for nugget_code, above, we *want* the following constraint.

Listing 9: Nugget size desired constraint with contiguous value ranges.

```
1  constraint nugget_size_valid {
2    size dist { 0 := 1, [1:511] :/ 1, [512:1023] :/ 1, 1024 := 1 }; }
```

First we consider the transliteration of Listing 9 into the Boolean equation (8).

```
((size == 0) || (size >= 1 && size <= 511) ||
 (size >= 512 && size <= 1023) || (size == 1024))
```
(8)

For each of the predicates, above, we may represent them using a class instance. The bool_pred base class in Listing 10 contains only a random flag indicating Boolean result. We will extend this class into Boolean operations later. For now, whenever the result flag is true consider the predicate being represented to also resolve to true. The nugget_size_ok positive result queue ensures at least one member's result flag to resolve to true using a logical-OR array reduction operation. The not_ok_vals constraint ensures that all member's result flag resolves to false using a logical-AND array reduction operation.

Listing 10: Boolean predicate base class used in positive/negative inside set queues.

```
1  virtual class bool_pred; // base class
2    rand bit result;
3  endclass
4  rand bool_pred nugget_size_ok[$];       // positive result queue
5  rand bool_pred nugget_size_not_ok[$];   // negative result queue
6  constraint okvals {
7      nugget_size_ok.or() with (item.result == 1); }
8  constraint not_ok_vals {
9      nugget_size_not_ok.and() with (item.result == 0); }
```

We may rewrite Eq. (8) as a logical-OR of predicates in Eq. (9).

```
(d || e || f || g)
```
(9)

Here, predicates **d** and **g** represent the assignment on the extremities while **e** and **f** represent the two contiguous inside ranges. For each Boolean predicate, we have abstracted out the underlying operation and represented them simply as Boolean equation (9).

The constraint solver works the in much the same way. First, the formula is rewritten into CNF. Then the solver makes a Boolean assignment on all predicates without regard to underlying operation; this is Boolean satisfiability, or simply SAT. Second, the Boolean assignment is validated within the underlying operation, known as the Theory solver; this is SAT modulo Theory, or SMT. For example, if, in Eq. (9), the Boolean solver assigns predicates **d** == true, **e** == false, **f** == true, and **g** == false, the Boolean formula results in true, a valid *Boolean-only* assignment. However, the theory solver knows that nugget size cannot be both 0, from predicate **d**, *and* within in the contiguous range 1-511, from predicate **f**, simultaneously. Therefore, the theory solver will punt back to the

Boolean solver for a new assignment. Structuring the constraint into a Boolean set and a theory set (the underlying operations) and iterating between the two is known as the Davis-Putnam-Logemann-Loveland (DPLL) procedure [1]. In Listing 10, and in our constraint library, we mimic this structure. The okvals constraint, line 6, *is* a Boolean clause, a disjunction of predicates. The predicates in question are instances of class extensions of the bool_pred base class, lines 1-3. We need only define each underlying operation as a bool_pred class extension, instantiate those and populate them in the nugget_size_ok positive result queue.

Listing 11: Boolean predicate class extensions for constant equivalency and uniform range and the free variable.

```
1   class bool_eq extends bool_pred;  // assignment
2     const int unsigned const_val;
3     rand free_var rhs;
4     constraint valid { solve result before rhs.value;
5       result -> const_val == rhs.value; }
6   endclass

7   class bool_rng extends bool_pred; // assignment on contiguous range
8     const int unsigned const_min_val;
9     const int unsigned const_max_val;
10    rand free_var val;
11    constraint valid { solve result before val.value;
12      result -> const_min_val <= val.value &&
13                val.value <= const_max_val; }
14  endclass

15  class free_var;
16    rand bit[31:0] value; // both size and addr are bit[31:0]
17  endclass
```

We have defined three new classes in Listing 11. First, the bool_eq class represents an equivalency on a constant value (e.g., size == 0). Second, the bool_rng class represents a contiguous range with uniform probability (e.g., size >= 1 && size <= 511). Finally, the variable under consideration is, itself, represented by an instance of the class free_var. For each constraint in a bool_pred class extension, the result flag must be resolved prior to the applying the operation. When result is true the operation is valid. When the result is false, the operation may be ignored. The following code constructs the nugget_size_uniform constraint for Eqs. (8) and (9).

```
1   free_var size = new();
2   bool_eq  size__0 = new(size, 0);          // size == 0
3   bool_rng size__1 = new(size, 1, 511);     // 1 <= size <= 511
4   bool_rng size__2 = new(size, 512, 1023); // 512 <= size <= 1023
5   bool_eq  size__3 = new(size, 1024);       // size == 1024

6   nugget_size_ok = { size__0, size__1, size__2, size__3 };
7   this.randomize();

8   // assume resolves to size.value == 0, then:
9   nugget_size_not_ok.push_back(size__0);
```

After executing the randomize call, on line 7, the nugget size free variable class instance has been selected according to the constraints in Listing 10. If randomize selects size.value == 0, then we may append the size__0 instance to the nugget_size_no_ok negative result queue to ensure 0 is not selected a second time, line 9. At the point of constraint solver failure, and given unique instances of predicates in the nugget_size_ok queue, then

- The queue sizes, nugget_size_ok and nugget_size_not_ok, are the same, and
- Values from all coverpoint bins have been generated in simulation.

The last point is most important as an aid to functional coverage and testplan completion. Once the solver has failed, it is feasible to simply allow randomization to again take over. There is no longer a need on this variable for blocking bins already hit.

3.3 Randomization with an added constraint – nugget address

Constraint construction for the 32-bit nugget address random variable is similar to the size random variable. However, an added constraint ensures generated addresses are double-word aligned (modulo 2). We may represent this constrained by nesting predicate operations and putting the clauses within a queue, the formula queue.

First, in Listing 12, we define the classes require for the CNF structured formula.

Listing 12: Class definitions for full CNF formula.

```
1   class bool_literal extends bool_pred; // positive or negative pred
2     const bit negative;
3     rand bool_pred pred;
4     constraint valid { solve result before pred.result;
5       result -> pred.result != negative;
6     }
7   endclass
8   class bool_clause extends bool_pred; // disjunction of literals
9     rand bool_literal literals[$];
10    constraint valid {
11      literals.or() with (item.result == 1);
12    }
13  endclass
14  class bool_formula extends bool_pred;// conjunction of disjunctions
15    rand bool_clause clauses[$];
16    constraint valid {
17      clauses.and() with (item.result == 1);
18    }
19  endclass
```

The formula only contains unique random free variable instances and unique predicate instances.[3] It is imperative to identify predicates that have resolved to true and negate them to block further selection.

The Boolean literal class performs the positive versus negative action that the nugget_size_not_ok queue and corresponding not_ok_vals constraint performed in the previous section, from Listing 10. When the Boolean literal is assigned a true value, then it propagates that value to its predicate based on its positive or negative identity. A positive literal propagates a true flag to its predicate, the underlying operation is valid. A negative literal propagates a false flag to its predicate, the underlying operation is ignored. A predicate instance may exist in separate positive and a negative

[3] Though, in our construction, we need not be concerned with uniqueness of predicates; that is something the constraint solver must and will maintain separately during random constraint evaluation.

literal instances simultaneously, thereby blocking the predicate's operation altogether.

The bool_clause class constructs a single clause of the constraint formula. This class employs a logical-OR array reduction operation to ensure at least one Boolean literal resolves to true. Recall that when a negative Boolean literal resolves to true, the underlying predicate resolves to false.

Finally, the bool_formula class enables construction of multiple clauses simultaneously. The formula is resolved during randomization with its logical-AND array reduction operation. All clauses within a formula must always resolve to true for the formula to be satisfied. It is the action of the constraint solver to find a valid assignment on all literals, predicates, and free variables to ensure the formula is satisfied.

A satisfied formula means that there exists a valid assignment on our variable. The simulation continues with valid stimulus. An unsatisfied formula means the constraint solver failed and no valid assignment exists for our variable. If,

a) The original formula construction is satisfiable, and
b) Clauses have been appended to the formula to block previous predicate selection, then

the covergroup for this variable is complete.

For the nugget address variable we must define a new class to employ the modulo and equivalency operations.

Listing 13: Boolean predicate class extension for modulo operation.

```
1   class bool_mod extends bool_pred;
2     const int unsigned const_divisor;    // val % divisor
3     const int unsigned const_remainder; //( modulo == remainder)
4     rand int unsigned mod_value;    // intermediate result op storage
5     rand free_var val;              // variable under evaluation
6     constraint valid {
7       solve result before mod_value;
8       solve mod_value before val.value;
10      result -> ( (mod_value == (val.value % const_divisor)) &&
11                  (mod_value == const_remainder) ); }
12  endclass
```

From the nugget class definition in Listing 1, the 32-bit random address variable required a second constraint to ensure the lower two bits are zero, thus making the address dword aligned. The covergroup need not ensure this requirement. Instead, it may still focus only on the address ranges that are the testbench is required to cover during simulation. The nugget address covergroup in Listing 14 is structured much like the size covergroup, for simplicity in this paper. A separate checker is assumed to ensure, when applicable, the lower two bits of address are zero. We perform no checking in the covergroup.

Listing 14: SystemVerilog functional covergroup for variable nugget address coverage goal.

```
1  covergroup cg_nugget_3;
2    coverpoint addr {
3      bins addr__0 = { 0 };                 // minimum valid value
4      bins addr__1 = { [1:0xffff] };              // lower range
5      bins addr__2 = { [0x10000:0xffff_fffe] };  // upper range
6      bins addr__3 = { 0xffff_fff8 }; }  // maximum valid value
7  endgroup
```

The new CNF structure to our constraint formula requires a different construction for simulation. In lines 4-7 below, we construct the uniform random range instances that mimic the covergroup for addr. These predicates are instantiated as positive literals in lines 10-13, and then set in one clause in lines 16-17. In line 8 we construct the dword alignment instance. This predicate is encapsulated in a positive literal, line 14, and within its own unit clause in lines 18-19. The c1 unit clause will ensure that regardless of formula change, the address will continue to be dword aligned. Both clauss are set in a Boolean formula in lines 21-22.

```
1   // construct the nugget address variable
2   free_var addr = new();
3   // construct the predicates
4   bool_eq  addr__0 = new(addr, 0);
5   bool_rng addr__1 = new(addr, 1, 'hffff);
6   bool_rng addr__2 = new(addr, 'h10000, 'hffff_fffe);
7   bool_eq  addr__3 = new(addr, 'hffff_fff8);
8   bool_mod dword_align = new(addr, 4, 0); // (addr % 4) == 0
9   // construct the literals
10  bool_literal a0 = new(addr__0, 1);          // positive literal
11  bool_literal a1 = new(addr__1, 1);          // positive literal
12  bool_literal a2 = new(addr__2, 1);          // positive literal
13  bool_literal a3 = new(addr__3, 1);          // positive literal
14  bool_literal dw_algn = new(dword_align, 1); // positive literal
15  // construct the two clauses, with dword-alignemnt as a unit
16  bool_clause c0 = new();
17  c0.literals = { a0, a1, a2, a3 };
18  bool_clause c1 = new();
19  c1.literals = { dw_algn };
20  // construct the formula
21  bool_formula f = new();
22  f.clauses = { c0, c1 };
23  f.randomize();
```

Predicate blocking may begin following the first randomization on this variable, from line 23 above. First, we identify the predicate or predicates that resolve to true and instantiate each in a new negative literal. Second, we instantiate each negative literal in their own unit clause and append to the formula. For example, if we identify that Boolean literal a1 has been hit in the first randomization, the address value is within the contiguous range of 1-0xffff, we may then block predicate addr__1 from further consideration.

Listing 15: Instantiating negative literals on the formula to block recurrence of the predicate.

```
1   // construct the blocking unit clause for literal a1
2   bool_literal block_pred = new(a1.pred, 0); // negative literal
3   bool_clause block_clause = new();
4   block_clause.literals = { block_pred }; // unit clause
5   f.clauses.push_back(block_clause); // append new clause
6   f.randomize();
```

In line 2 above, we take the bool_pred instance from the literal a1 in order to construct the negative

literal. Recall that predicate instances may exist in both positive and negative literals simultaneously. The end result, however, is that the predicate is simply blocked from consideration. The unit clause consisting of a single negative literal ensures that the predicate does not resolve to true.

Randomization, in line 6 above, will converge on a different literal than a1. Literal a1 is excluded from consideration. Continuing randomization in this manner will, eventually, result in constraint solver failure. Once that occurs, the formula's clause queue need only to revert to its initial state to continue simulation. In this example, the formula initially contained two clauses. Once all coverpoint bins have been generated and constraint solver fails, we may pop off all new clauses until only the two initial clauses remain.

4. Generic Random Container Class Library

Our generic random container class library, lvm_rand, builds on the work discussed in Section 3. Furthermore, this class library expands the functionality from [2]. Whereas, in [2] we allowed for dynamic construction of various kinds of constraints with constant values, now we construct for infinitely deep constraints and in a manner that may be manipulated.

```
lvm_rand#(int) myrand = new("MYRAND", this);
myrand.push("inside { 4, 5 }");
```

The constraint formula supplied above, following algorithms described in [2], would result in a single constraint class container implementing a simple queue with constant int value, much like Listing 4. This constraint may be dynamically interchanged (i.e., replaced) but not manipulated. The above constraint could only always select constant values 4 or 5 with uniform probability.

However, now we construct the formula for infinitely deep constraints, more constraint types, including some arithmetic operations, and for manipulation.

```
lvm_rand#(int) myrand = new("MYRAND", this);
myrand.push("value inside { inside { 0, 1 }, [5:10],
                            dist { 15 := 1, 20 := 80 } }");
myrand.AND("value != 0");
```

The pushed constraint above randomly selects, with uniform probability, an inside constraint, a contiguous range constraint, or a user-defined distribution constraint. Once the sub-constraint is selected, randomization continues to find an assignment for myrand.value. However, the ANDed constraint, above, ensures that that assignment is not equal to 0.

The complexity of construction, however, does increase with the expressiveness of the supplied constraint. It is, therefore, critical that, regardless of the internal construction, the constraint may be reported in a user-friendly way (i.e., as supplied by the user). In the subsections that follow, we show how we may manipulate the instantiated constraint in manual (via AND function call) and automatic ways (via automatic predicate promotion).

4.1 Class Structure

Two separate class trees are implemented in the class library, one tree for the lvm_rand container class, itself, and one tree for the applied constraints. The lvm_rand container class provides the user programming interface. This interface is the only interface the testbench is required to access. The constraint class tree is only accessed behind-the-scenes by the constraint factory class to instantiate and manipulate constraint containers.

4.1.1 lvm_rand

All lvm_rand classes are extensions to the uvm_object class, as in Figure 1. This extension is not actually necessary as there is nothing in the lvm_rand classes that *strictly require* the UVM library. Nonetheless, as UVM is increasingly common in modern testbenches, we have opted to extend from the UVM library. This gives the added benefit of access to the UVM configuration and resource databases as well as the UVM command-line processor. Furthermore, when working with parameterized classes it is helpful to maintain an un-parameterized base class, lvm_rand_base, which may be used for generic class reference casting. All classes from lvm_rand_param to lvm_rand must maintain their type parameter for proper access.

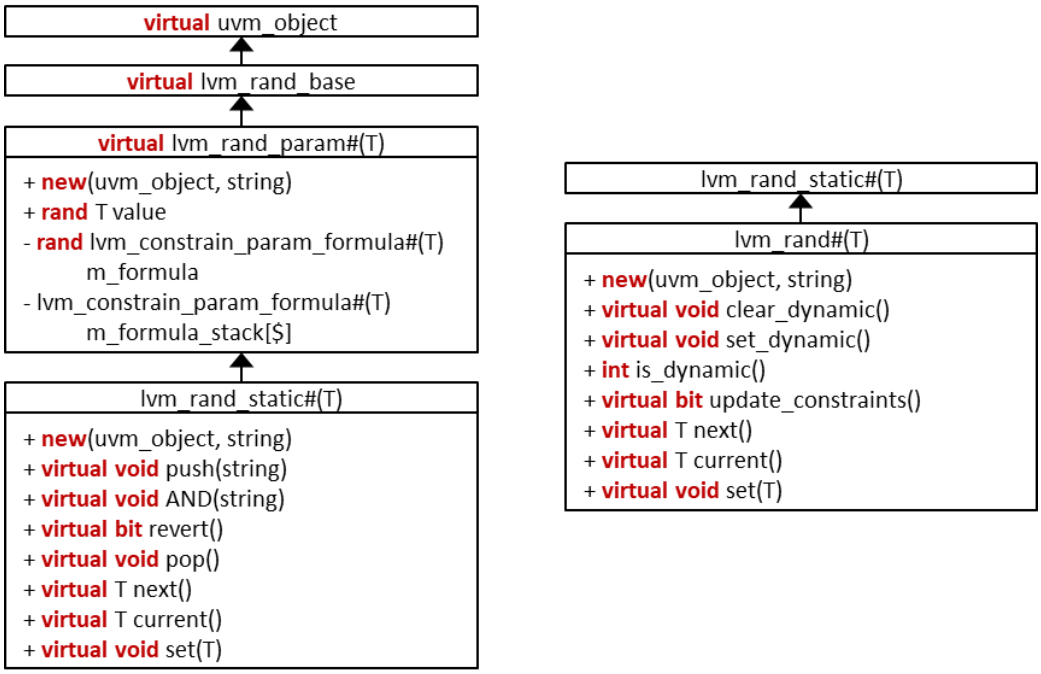

Figure 1: LVM random container class inheritance hierarchy.

The type-parameter for the lvm_rand class hierarchy, Figure 1, may be any randomizable SystemVerilog integral type. The lvm_rand_param#(T) base class contains the actual randomized value. It is this value that will be assigned during randomization. The lvm_rand_static class disallows external dynamic constraint instantiation. That is, while the constraint may be pushed onto an instance of the lvm_rand_static#(T) class it will not access any command-line or UVM configuration or resource database. The constraint is "static" in that it will not change on its own. However, in the lvm_rand#(T) class extension, the command-line and UVM configuration and resource database are consulted for randomization during the next() function call.

We have noted in previous work that *many* successive UVM configuration and/or resource database accesses result in simulator performance degradation [2]. As such, the our guidance is to update the constraints from these interfaces once just before the UVM common run-time phases with a call to the update_constraints() function. In this manner, the command-line, testbench, tests, and UVM sequences may all have a chance to set a constraint change prior to first use during simulation. Then, we advise disabling the lvm_rand's inherent dynamism with a call to clear_dynamic().

In all lvm_rand_static#(T) or lvm_rand#(T) instances, regardless of other parameters, the push(), AND(), and pop() functions are accessible. The lvm_rand_param#(T) class, Figure 1, maintains both the active constraint, m_formula, and the inactive constraint stack, m_formula_stack. Note that the active constraint is labeled rand while the inactive stack is non-rand. Constraint reference may move between the two to enable/disable constraints.

In Figure 2, at lvm_rand construction, no constraint exists and the active constraint is null. Thus, at lvm_rand construction and until a push() function call, the random variable is *unconstrained* (A). Calling push()parses a *new* constraint string, instantiates its formula, assigns that to the m_formula reference. In Figure 2(B), for example, the formula container class, lvm_constrain_param_formula, has been instantiated containing references to two Boolean clauses. This now becomes the active constraint on this variable. A call to pop() now removes the active constraint by assigning the m_formula variable to null.

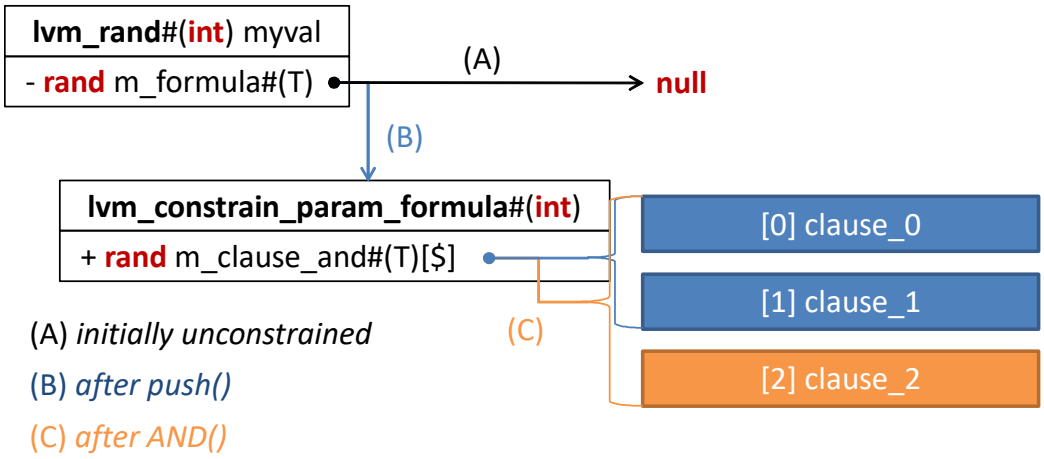

Figure 2: LVM random variable reference to its formula during simulation, initially unconstrained, then after push, after AND.

If the m_formula variable had already been non-null, then calling push(), in Figure 2(B), would first push the current m_formula reference to the inactive constraint stack, m_formula_stack. Now, two formulas exist on this lvm_rand: an active formula in the m_formula reference, and an inactive formula in the m_formula_stack queue. A call to pop() now removes the active constraint in m_formula *and* pops the last constraint from the m_formaul_stack queue and assigns that back to m_formula. The lvm_rand returns to its previous state prior to push.

The lvm_rand_static#() class, in Figure 1, also defines the AND() and revert() functions. Referring to Figure 2(C), calling AND() parses a new constraint string but, instead of replacing the current constraint in m_formula, the new constraint is *appended* to the active formula in m_formula. At the next randomize() call, all three clauses will be in effect on the random variable. Now, the original constraint (blue clauses, 0 and 1) on this variable has been modified (with the orange clause, 2). It is possible to remove all ANDed clause with a call to revert(). The formula container class keeps track of the original formula for easy reversion.

Finally, because the random variable is a class member of the instantiated lvm_rand, Figure 1, both a next() and current() function are provided for easy access to the value, post-randomization. In the lvm_rand_static#(T)::next() function call, the formula is randomized and converged value returned. In lvm_rand#(T)::next() when is_dynamic() is true, first the command-line and UVM configuration and resource databases are queried for an updated constraint. If a constraint exists

in those locations (where command-line has highest priority but only accessed once, the first time next is called), then the constraint is parsed and updated, then the formula randomized and converged value returned. However, the current() function will only return the last randomized value or, if the variable has not yet been randomized, will return the value of next().

4.1.2 lvm_rand_constraint

The LVM random constraint class hierarchy is where all the constraints available come together. It is expected that the SystemVerilog Constraint Factory will instantiate these classes as required from the constraint string provided. No user intervention is necessary to construct the well-formed CNF constraint formula for randomization. Each constraint constructed appears as instantiated Boolean clauses in the lvm_rand_param#(T).m_formula class reference, as shown in Figure 2 and described in more detail in Section 4.2 .

Figure 3 indicates the necessary classes for constructing the CNF formula. These classes are very similar to the classes presented in Section 3.3. The formula is a Boolean predicate and contains a queue of clauses. The clause is a Boolean predicate and contains a queue of literals. The literal is a Boolean predicate and either propagates a true flag (positive literal) or false flag (negative literal) to its contained predicate.

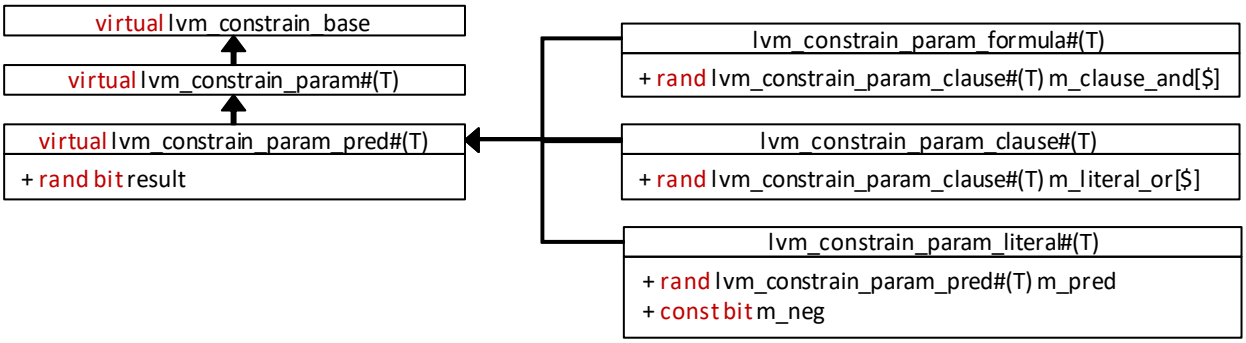

Figure 3: LVM random constraint formula class inheritance hierarchy.

The remaining classes are split into value parameter constraints and binary operation parameter constraints, Figure 4.

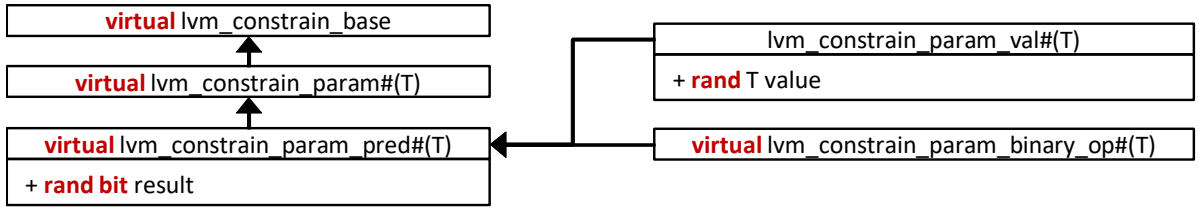

Figure 4: LVM random constraint operation class hierarchy.

The free variable, defined as an instance of the lvm_rand_static#(T) or lvm_rand#(T) class, is represented in the constraint formula as an instance of the lvm_constrain_param_val#(T) class. A reference to this class is required for binary operations on the random variable (e.g., value > 4). The binary Boolean operations indicated in Table 2 are available in the class library.

Table 2: Binary logical operations in the LVM Random Library.

Class name	Operation	Definition
lvm_constraint_param_binary_op_eq#(T)	a == b	Equal
lvm_constraint_param_binary_op_neq#(T)	a != b	Not equal
lvm_constraint_param_binary_op_gte#(T)	a >= b	Greater-than-or-equal
lvm_constraint_param_binary_op_gt#(T)	a > b	Greater-than
lvm_constraint_param_binary_op_lte#(T)	a <= b	Less-than-or-equal
lvm_constraint_param_binary_op_lt#(T)	a < b	Less-than

Both sides of the Boolean binary operation are expected to be some sort of Boolean value predicate. Often, this is simply a reference to the random variable, lvm_constrain_param_val#(T). However, it may alternatively be a reference to any of available the lvm_constrain_param_val#(T) class extensions, Figure 5.

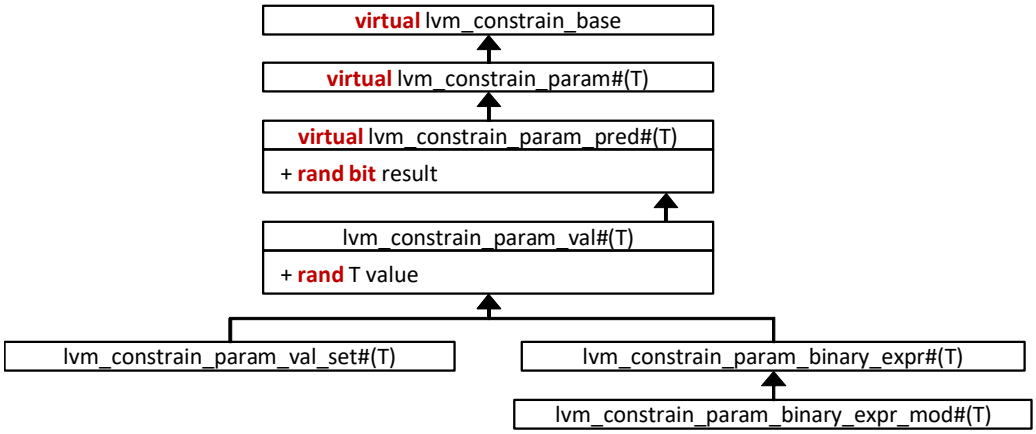

Figure 5: LVM random constraint value and binary expression class hierarchy examples (only a subset shown).

For example, in Figure 5, the parameterized value inside set class is an extension to the parameterized value class. In fact, all the parameterized value classes indicated in Table 3 are supported in the library.

Table 3: Parameterized value class containers supported in LVM Random library.

Class name	Definition
lvm_constrain_param_val_const #(T)	Simple constant value
lvm_constrain_param_val_range #(T)	Contiguous range of value prediates
lvm_constrain_param_val_set #(T)	Uniform distribution on set of value predicates
lvm_constrain_param_val_dist #(T)	User distribution on a set of value predicates
lvm_constrain_param_val_seq #(T)	A deterministic selection of value predicates
lvm_constrain_param_binary_expr #(T)	A binary arithmetic expression

The sequence constraint differs from SystemVerilog constraints in that the set of value predicates are not bound by random selection, uniform or otherwise. Instead, the first value predicate in the list is selected during the first randomize call, the second value predicate on the second randomize, etc., until the end of the list. Then, the list may be rewound to the beginning, a loop, or simply stick at the last value predicate.

In this structured CNF formula we cannot use arithmetic operators directly. Instead, we must build them into the instantiated formula as classes. Therefore, these are implemented as extensions to the lvm_constrain_param_binary_expr#(T) class, from Figure 5. While only the modulo operator is shown in the figure, all the expression types indicated in Table 4 are supported.

Table 4: Binary arithmetic expressions in LVM Random library.

Class name	Operation	Definition
lvm_constrain_param_binary_expr_plus #(T)	a + b	Addition
lvm_constrain_param_binary_expr_minus #(T)	a - b	Subtraction
lvm_constrain_param_binary_expr_times #(T)	a * b	Multiplication
lvm_constrain_param_binary_expr_divide #(T)	a / b	Division
lvm_constrain_param_binary_expr_lshift #(T)	a << b	Bit-wise left shift
lvm_constrain_param_binary_expr_rshift #(T)	a >> b	Bit-wise right shift
lvm_constrain_param_binary_expr_bit_or#(T)	a | b	Bit-wise OR
lvm_constrain_param_binary_expr_bit_and(#T)	a & b	Bit-wise AND
lvm_constrain_param_binary_expr_bit_xor#(T)	a ^ b	Bit-wise Exclusive-OR
lvm_constrain_param_binary_expr_bit_xnor#(T)	!(a ^ b)	Bit-wise Exclusive-NOR

Of course, the user will not need to manipulate any of the constraint classes in Table 2, Table 3, or Table 4, directly. Instead, a string parser and SystemVerilog constraint factory will perform the work automatically.

4.2 C++ Parser Front-end and Factory Back-end

As discussed in Section 3.3, it is quite tedious to construct the full constraint by hand. As such, we have defined a proprietary grammar, based on SystemVerilog constraint syntax, to allow the constraint to be presented to the lvm_rand class as a string. The constraint string, itself, may be composed during simulation.

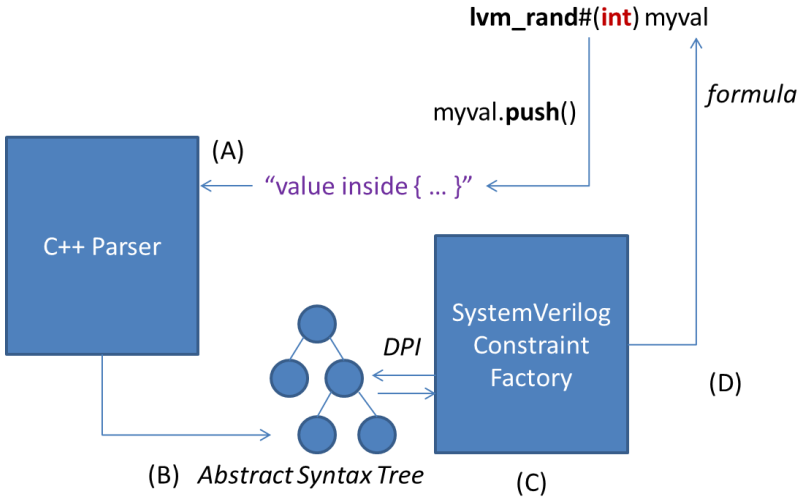

Figure 6: Parse and construct a new constraint.

In Figure 6, the parser is initiated from an lvm_rand_static#(T) or lvm_rand#(T) instance push() function call. The string provided to the push function is passed to the C++ parser, Figure 6(A), via a direct programming interface (DPI) function. The parser ensures the string is well-formed and creates an internal abstract syntax tree, an intermediate model of the constraint, (B). Once the parse and resulting model are complete, the SystemVerilog Constraint Factory walks the tree, (C), constructing the necessary SystemVerilog classes to represent the new constraint. Both the initial push and the abstract syntax tree walk are facilitated from SystemVerilog via DPI functions. Finally, the constraint factory returns the completed formula, (D), now as a collection of instantiated constraint container classes that represent the original constraint string. Randomization may then commence employing the reference to the new constraint container, m_formula, specialized for the user's constraint string.

Similar to push, the AND() function parses a constraint string and generates an internal abstract syntax tree. However, instead of overriding the current formula, the clauses of the new formula are appended to the existing formula. Randomization will now be affected by both the original clauses and the appended clauses.

With the constraint supplied as a string, it may be supplied directly on the command-line using the lvm_rand instance name as its plusarg identifier. For example, below an lvm_rand of int parameter type is created with then name MYRAND.

```
lvm_rand#(int) myrand = new("MYRAND", this);
```

At simulation runtime, a constraint may be supplied on the command-line as a plusarg.

```
> simv +MYRAND="value inside { [0:1000] } && value % 2 == 0" …
```

The plusarg may simply be the lvm_rand variable name or a full UVM testbench path instance.

```
> simv +tb.env.MYRAND="value inside { [0:1000] } && value % 2 == 0"
```

Alternatively, the uvm_config_db or uvm_resource_db may be accessed from the command-line or from within the testbench proper using string datatype.

```
uvm_config_db#(string)::set(uvm_top, "", "MYRAND",
          "value inside { [0:1000] } && value % 2 == 0");
```

4.3 Automatic Predicate Promotion

In the lvm_rand_static#(T)::post_randomize function call (not represented in Figure 1), we traverse the instantiated active formula to determine which predicates have matched the selection. Generally, only one predicate matches, but it may be possible to match multiple predicates if there exists an overlap on the selected value and defined predicates (e.g., value == 0 && value inside { 0 } would result in two predicates matching when value converges on zero). These predicates are then instantiated negatively directly on the constraint formula, as shown in the example code in Listing 15. Each predicate is instantiated again within a negative literal and appended to the formula as a unit clause. At the next randomization, the negative literal ensures the predicate will be excluded from consideration because the predicate's result flag cannot be assigned as true.

The consequence of automatic predicate promotion (to negative unit clause) is that, at some point, the constraint solver *will* fail. Failure is desirable because it indicates all possible random values have been selected. At constraint solver, the formula must be reverted to its original construction. Therefore, the lvm_rand_static#(T)::next function ensures that upon constraint solver failure, revert() is called and randomization occurs a second time. Failure a second time implies the original formula was unsatisfiable (unable to converge on any value), even if it were well-formed.

Listing 16: Next function implementation with auto-predicate promotion

```
1   function lvm_rand::T lvm_rand::next();
2     if(is_dynamic())
3       void'(update_constraints());
4     return super.next();
5   endfunction
6   function lvm_rand_static::T lvm_rand_static::next();
7     int flag = this.randomize();
8     if(flag == 0) begin
9       if(is_constrained() &&
10         enabled_auto_predicate_promotion()) begin
11        void'(m_formula.revert_formula(1));
12        flag = this.randomize();
13        if(flag == 0)
14          `uvm_fatal("RAND/NEXT", "Randomization post-revert failed")
15      end else begin
16        `uvm_fatal("RAND/NEXT", "Randomization failed")
17      end
18    end
19  endfunction
```

In Listing 16, the next function is shown for both lvm_rand#(T), lines 1-5, and lvm_rand_static#(T), lines 6-19. When automatic predicate promotion is enabled, line 10, and the constraint solver fails, the current active formula is automatically reverted back to its original state, line 11, and randomization occurs again, line 12. In order to take advantage of this procedure, the simulator must be instrumented to allow simulation to continue upon random constraint failure. In Synopsys VCS, that means +ntb_stop_on_constraint_solver_error=1 *must not* be enabled [3].

5. Example

Consider the following example formula, as presented from the abstract.

```
inside { 0, [1:9], 10 }
```

This constraint is well-formed in the LVM Random grammar. There is an implication that the random variable, value, is assigned to, with uniform probability, one of the values indicated in the inside set. After first randomization, we wish to block the contiguous inside range 1 – 9. In other words, we wish to logically AND the following new constraint with the original.

```
! ( inside { [1:9] } )
```

The combination of the two constraints ensures, on the following randomization, the value will be assigned either 0 or 10, with uniform probability. First we construct the original constraint with the class objects discussed in Section 4. Then, we augment the original constraint with the blocking constraint. For this discussion, we assume a type-parameter of int.

5.1 Constructing the constraint

All lvm_rand constraints are built in a CNF-like structure.[4] Our original constraint formula has a single clause with a single positive literal (a unit clause) assigning the lvm_rand value to the result of the inside set, a binary Boolean operation. The set, in turn, is a construction of separate value predicates. The inside set construction is depicted in Figure 7. Notice that the m_set queue refers to additional type-parameterized value class instances, regardless of the actual leaf type.

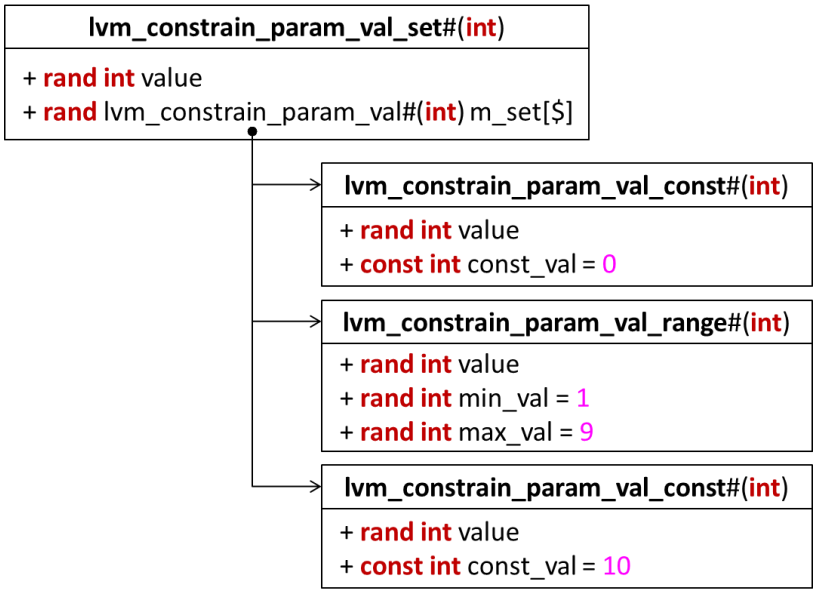

Figure 7: Inside set construction for example constraint.

The lvm_constrain_param_val_const class contains a SystemVerilog constraint that ensures the random "value" class member is assigned to the constant value class member (value == const_val).

[4] Though, we may cut corners in our construction because, ultimately, the constraint solver will perform the final CNF translation for solving

The lvm_constrain_param_val_range ensures value is within bounds (min_val <= value <= max_val). Each of these value constraints has equal probability according to the SystemVerilog constraint on m_set. The value from the chosen sub-constraint propagates up to the set's "value" class member.

A binary Boolean operation is employed to tie the random variable's representative with the result of the inside set. Figure 8 depicts the binary Boolean operation instance with references to the two lvm_constrain_param_val classes. The lvm_rand's representative is referenced at the binary operation lhs class member, while the inside set is at the rhs class member. The Boolean equivalency operator's SystemVerilog constraint ensures the values within those references are equivalent. Thus, the random value chosen from the inside set is propagated to the random variable's representative in the constraint formula.

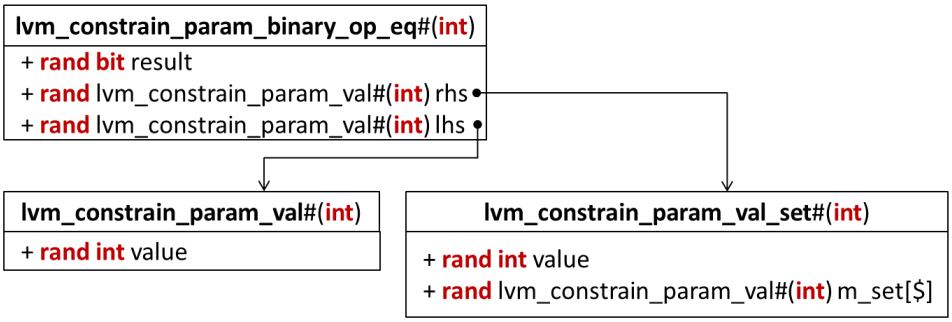

Figure 8: Binary Boolean operation ties together the instantiated value representative with the inside set.

The binary Boolean operation *is* the predicate in this formula's lone literal and clause. Therefore, the binary Boolean operation is instantiated within a positive literal (m_neg = 0), within a unit clause, and within the formula, as shown in Figure 9.

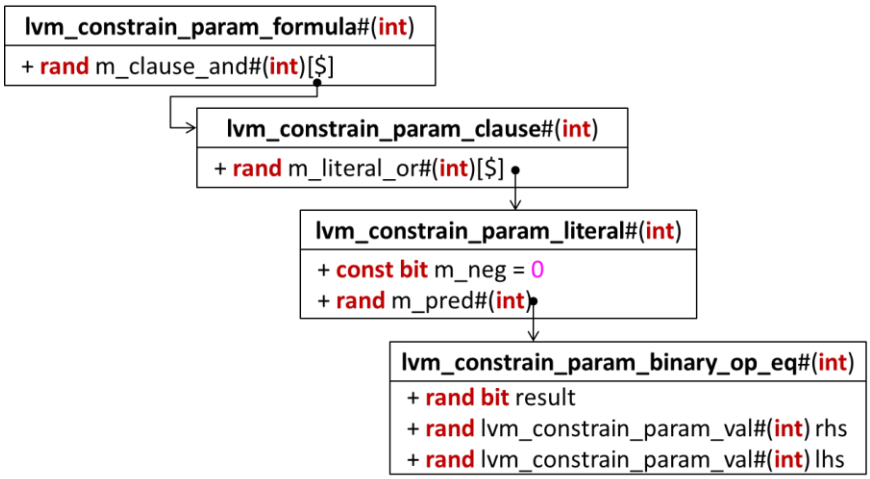

Figure 9: Formula for original example constraint.

Finally, the lvm_rand class, itself, randomizes according to the instantiated formula while propagating up the chosen value. The lvm_rand's representative class is also referenced in the lvm_rand itself, as shown in Figure 10. This is the same class instance used throughout the formula. At randomization, a SystemVerilog constraint in the lvm_rand class ensures the local value is equivalent to m_val.value. Thus, the lvm_rand.next() function can safely call randomize() and immediately return the local value.

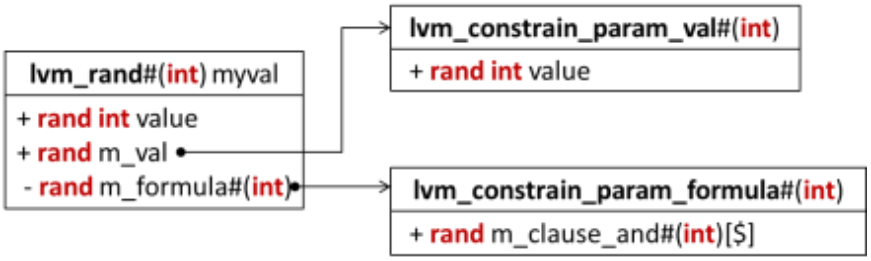

Figure 10: LVM Rand for the example. The value and formula representative are referenced in the lvm_rand class.

5.2 Augmenting the constraint

After the first randomization, we need to construct a new unit clause to block the inside contiguous range 1 – 9. This can be constructed in one of two ways. Either we may use a binary Boolean not-equivalent operation to negate the range: value != inside { [1:9] }. Or, we may use the same binary Boolean equivalency operation within a negative literal. We choose the latter in Figure 11.

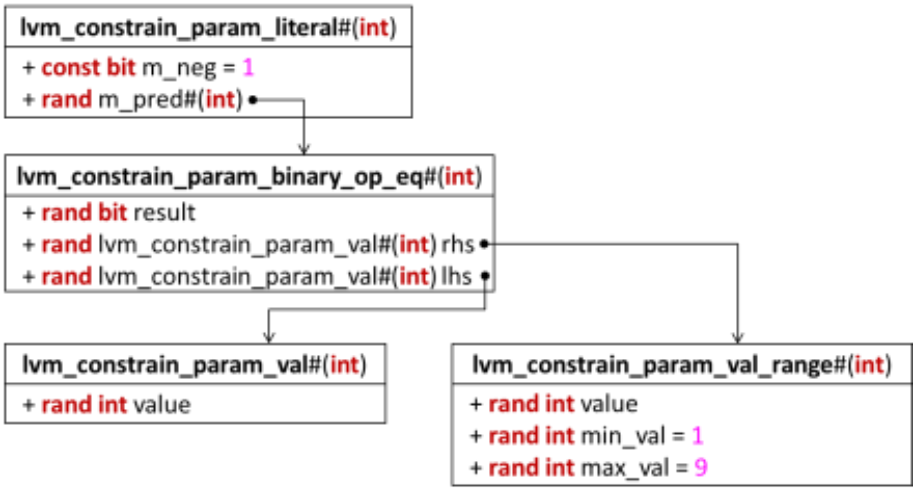

Figure 11: Negated literal for formula augmentation.

The new negated literal is instantiated in a second (unit) clause and appended to the m_clause_and queue in Figure 9. The result is that the inside contiguous range is blocked from randomization because the value cannot be both inside [1:9] AND outside [1:9].

This second step, the augmentation, occurs in the lvm_rand variable when automatic predicate promotion is enabled. That is, when the predicate inside range [1:9] matches the selected random value, then it is automatically promoted to a negated literal and instantiated as a new unit clause in the formula. In this way we can isolate selected predicates from consideration in future randomizations until the formula is reverted to its original state.

When each predicate in the formula matches a coverpoint bin from our testplan, then each successive randomization nudges verification progress. Once all predicates in the formula are selected (i.e., at constraint solver failure) then we revert the formula and let randomization take its course.

5.3 User Interface

The user does not need to know anything about the internal construction of the formula. Instead, the push()/next() interface is sufficient. When coupled with a covergroup and automatic predicate promotion then all points are always covered in three randomizations. The order of coverage, of course, changes with the random seed.

Listing 17: User interface for example constraint construction.

```
1   class mytest extends uvm_test;
2     covergroup cg_myval(ref int value);
3       cp_values: coverpoint value {
4         bins range__0 = { 0 };
5         bins range__1 = { [1:9] };
6         bins range__2 = { 10 }; }
7     endgroup
8     lvm_rand#(int) myval;
9     function new(string name = "mytest",
10                uvm_component parent = null);
11      myval = new("MYVAL", this);
12      cg_myval = new(myval.value); // always available
13      myval.push("inside { 0, [1:9], 10 }"); // original constraint
14      void'(myval.next()); // assume 1 <= value <= 9
15      cg_myval.sample();   // range__1 is now covered AND blocked
16      void'(myval.next()); // assume value == 0
17      cg_myval.sample();   // range__0 is now covered AND blocked
18      void'(myval.next()); // now value == 10
19      cg_myval.sample();   // range__2 is now covered AND blocked
20      // Now at next() the constraint reverts and all three ranges in
21      // the inside set are again available with uniform probability
22    endfunction
23  endclass
```

The myval random variable is instantiated in Listing 17. From strictly the random variable perspective, the user needs only to declare (line 8), instantiate (line 11), push a constraint (line 13), and use the random variable directly (lines 14, 16, and 18). Everything else is added functionality available to the user when necessary.

In line 11, the random variable is instantiated with the name "MYVAR" and, because this is an uvm_test class extension, the full UVM testbench name is "uvm_test_top.MYVAR". This string may be used on the command-line and/or the UVM configuration or resource databases to override the default constraint indicated on line 13. Both Synopsys VCS simv command-lines below are valid (careful with terminal interpretation of special characters).

```
> simv +uvm_test_top.MYVAR='dist {0 := 10, [1:9] := 40, 10 := 50}'
> simv +MYVAR=0
```

Furthermore, in Listing 17, we have specified a covergroup for the three inside set values. Note that, in line 2, the sample value is taken as a reference port to the covergroup. The only requirement on the coverpoint cp_values, line 3, is that it may bind to a locally accessible variable. As any lvm_rand#(T) or lvm_rand_static#(T) always has its value publicly accessible, we could use that directly to bind the coverpoint to myval.value. Here, we opted to bind the coverpoint to the reference port. Then at covergroup instantiation, line 12, the lvm_rand#(T) instance's value is bound to the covergroup port. In lines 14, 16, and 18, we ignore the myval.next() return value because the covergroup port is already bound to the same. As such, in lines 15, 17, and 19, sampling the covergroup will pick the current random variable's value, post-randomization.

It is also feasible to extend the lvm_rand#(T) class in order to directly incorporate the covergroup into the random variable class definition. In this case, the lvm_rand#(T)::next() function could be overridden to also incorporate the covergroup sample function.

The expressiveness of the random variable container classes demands some complexity in structure. However, this complexity should not burden the user of the container class. In most, if not all, cases, the user will only consider the push/pop/next/current programming interface, and the command-line access, available to lvm_rand.

6. Future Work

Several avenues of work are necessary in our random contain class library. We enlist the help of the verification community resolve these challenges and expand the library's usefulness.[5]

6.1 Random Variable Dependencies

As discussed in Section 4, we employ a parser on the front-end and a factory on the back to ease the constraint description and instantiation. However, while the existing CNF structure may include multiple random free variable instances, the parser has difficulty with that. Consider the example below with two random variables with unidirectional dependency.

```
lvm_rand#(int) myrand0 = new("MYRAND0", this);
lvm_rand#(int) myrand1 = new("MYRAND1", this);
myrand0.push("value inside { 1, [3:10] }");
myrand1.push("value inside { [3:10] } && value != MYRAND0.value");
```

The parser and constraint factory, themselves, do not have immediate connection between the string "MYRAND0" and the SystemVerilog symbolic myrand0 lvm_rand instance. As such, while the formula in myrand0 could be constructed to perform randomization over these constraints, the parsaer and constraint factory cannot.

6.2 Cross Coverage Dependencies

As a corollary to the variable dependency, functional covergroup crosses are not directly supported for automatic predicate promotion. That is, when two or more variables are crossed a set of cross bins are created. While the current approach could support more complex variable dependencies, with an enhanced parser and constraint factory, the identification of cross-bin related dependencies would need to be addressed. In this case, we would want to block the individual cross-bin while allow other cross-bins to be available. Consider the following covergroup A.

[5] Refer to the website www.verificationhack.com for the lvm_rand class library.

```
covergroup cg_A;
  coverpoint a { // lvm_rand#() a = push("value inside { 0, 1 }");
    bins a0 = { 0 };
    bins a1 = { 1 };
  }
  coverpoint b { // lvm_rand#() b = push("value == 0");
    bins b0 = { 0 };
  }
  cross a, b {
    c0: binsof(a.a0) && binsof(b.b0);
    c1: binsof(a.a1) && binsof(b.b0);
  }
endgroup
```

Assume coverpoints a and b are tied to two lvm_rand.values of the same name, as indicated in the comments. On the first randomization of these dependent variables suppose cross bin c0 is covered, that is a.value == 0 and b.value == 0. At this point we wish to block a.value == 0 via automatic predicate promotion while *not* blocking b.value == 0. In this example, we could enable promotion on lvm_rand a while disabling it on lvm_rand b. But, as dependencies become more complex and crosses expand it won't be so straightforward.

6.3 Random Variable Constraint Files

While the lvm_rand class variable and background support classes have access to the command-line, parsing a file is not yet supported. The Synopsys VCS simulation *can* take a filename with simulation-time command-line arguments. However, this can get convoluted, especially within regression scripts. Instead, we would like to have argument for a parse-able file (with comments) that can `include other files.

6.4 Enumeration String-to-Symbolic Name Correlation

Within a single random variable, at present time, there is no facility to link enumeration symbolic names with their string equivalent within the constraint string.

```
typedef { ONE, TWO, THREE } myenum_t;
lvm_rand#(myenum_t) myrand = new("MYRAND", this);
myrand.push("value inside { ONE, THREE }"); // not available
```

We may compose a string using literal values converted from the symbolic equivalent, as below.

```
myrand.push($sformatf("value inside { %0d, %0d }", ONE, THREE));
```

Within the constraint factory, only numeric values are used, regardless of enumeration. This approach does not aid in good programming techniques.

7. Conclusions

We presented our random constraint library that takes a string and constructs a dynamically-modifiable random constraint. The top-level lvm_rand class is type-parameterized, has access to the command-line and uvm_config_db and uvm_resource_db databases, as well as an API for dynamic instantiation and changing of constraints. Its constraint stack allows for pushing simultaneous independent constraints where the stack head is always enabled. In this manner, temporary constraints may be pushed then, later, popped to return to the original constraint. Our

constraint syntax largely follows SystemVerilog standard with the enhancement of infinite nesting.

We have employed this approach in its current iteration in multiple verification projects with success. We look forward to expanding the library's capabilities and integration with functional coverage.[6]

8. References

[1] C. Barrett, R. Sebastiani, S. Seshia and C. Tinelli, "Satisfiability Modulo Theories," *Handbook of Satisfiability,* vol. 185, 2008.

[2] J. Ridgeway, "Interchangeable SystemVerilog Random Constraints," in *Synopsys User Group (SNUG)*, San Jose, 2014.

[3] Synopsys, Inc., "VCS MX/VCS MXi User Guide," 2019.

[4] IEEE Computer Society, SystemVerilog, 2012.

[6] The library and all theory code are available via the website: www.verificationhack.com.