Engineered SystemVerilog Constraints

Jeremy Ridgeway
Avago Technologies, Inc.
Fort Collins, CO 80525
jeremy.ridgeway@avagotech.com

***Abstract*-SystemVerilog constraints are declarative--they must be elaborated prior to simulation. There is some flexibility at run-time when class members are used in the constraints as bounds or sets. Constraints and random variables are otherwise limited to enabled or disabled. This poses problems for verification engineers (and their managers) when completion means 100% functional coverage. The verification cycle is too short to rely solely on the random number generator to achieve all coverage goals. We present a set of engineered constraints that may be manipulated, augmented, and/or replaced on-the-fly and without requiring SystemVerilog tasks. The constraint formula is transformed to a set of container classes structured in conjunctive-normal form (CNF), similar to the constraint solver's representation. Then, clauses may be added or subtracted as needed to guide randomization. Engineered constraints can drive simulation towards functional coverage complete.**

I. Introduction

Current constrained random verification (CRV) environments rely on randomness with valid stimulus generation to verify the device under test (DUT). It is generally accepted that this strategy uncovers unforeseen (and often high quality) bugs and corner cases more efficiently than directed testing approaches alone [1, 2, 3, 4]. Manually enumerating test scenarios cannot determine all the interesting interactions between complex DUT state machines. Constrained randomness assists by not only removing much of the verification bias but also will eventually generate these interesting scenarios.

Even as CRV has taken hold, reporting strategies have had to catch up. Previously, a suite of tests to be implemented and ensure passed was a straightforward way to determine the current status of verification according to plan and schedule. This has largely been replaced with functional coverage. The combination of constrained random stimulus and functional coverage is the mainstay of coverage driven verification (CDV). While the best way to plan, implement, and report functional coverage remain open to debate, for example [5, 6, 7, 8, 9], the CDV methodology is accepted as an efficient way to achieve and report verification goals, [1, 2, 3, 10] among others.

At the heart of CRV and CDV is the random number generator and the accompanying constraint solver embedded in the simulator chosen for functional verification. There is very little guarantee that, in any given simulation or regression, random bins in a constraint will be hit. For example, a uniform probability SystemVerilog set, *val* **inside** {0, 1, 2}, offers no guarantee that all values will be generated. Instead, coverage is left to the whim of the random generator and random seed. In the meantime, verification engineers (and their managers) are looking for optimizations to CRV to more efficiently hit cover goals.

Constraints for CRV in SystemVerilog are declarative in nature. The language does provide for some limited manipulation of already elaborated constraints during simulation [11]. For example, local variables may be used instead of numbers for a random variable's constraint:

$$val\ \textbf{inside}\ \{0,\ [1{:}max]\}. \qquad (1)$$

Also, random variables and their constraint blocks may be individually enabled or disabled during simulation. However, only a SystemVerilog task may perform this action (functions only report current random mode). Constraint blocks may be overridden (or removed) through class inheritance. However, current guidance recommends against this practice on randomized data descriptors [3].

In [12], we presented a limited library of constraint types that could be instantiated, then totally replaced and instantiated again, all on-the-fly. We were able to alleviate some of the constraint concerns, notably inheritance in data descriptors, with the approach presented, but it required a wholesale replacement of the constraint. In this paper, we present an approach for engineering constraints to not only instantiate new constraints but also modify existing constraints, all on-the-fly and purely with a set of function calls. A type-parameterized random variable container class can then modify the constraint automatically—in its post-randomization function—by closing off individual values or whole ranges based on the last randomization. For example, engineered constraints may close off the contiguous range from Eq. (1), above, by augmenting the constraint to:

$$val \textbf{ inside } \{0, [1{:}max]\} \;\&\&\; !(val \textbf{ inside } \{[1{:}max]\}).$$

Constraints may be dynamically added or removed as required by the current state of simulation and according to functional coverage goals.

This paper is organized as follows: In section II, we discuss SystemVerilog constraints, some issues and theory behind the constraint solver. In section III, we employ the theory to present our engineered constraints. We tie constraints to a random variable container class in section IV. Limitations and future work is discussed in section V. Finally, we conclude in section VI.

II. SYSTEMVERILOG CONSTRAINTS

Consider a constrained random verification environment for some packet based transmission device. The packets are data transactions in the environment and are modeled as a chain of inherited classes. At the base class, random length is defined to control packet sizes.

```
class base_packet;
    rand int len;
    constraint valid {
        len >= 0 && len <= 1024;
    }
endclass
```

Without further refinement, the `valid` constraint, above, defines a uniformly probable random space. The values chosen by the constraint solver at simulation-time, in the aggregate, should be uniformly distributed over the entire range, 0 to 1024. There are two implications here:

Implication 1.	All packet lengths are uniquely considered important to verification, and
Implication 2.	At a minimum, 1025 packets must be generated to *possibly* hit all (important) cases.

Of course, the first implication is probably not true. That is, two packets of length 33 and 35 bytes, respectively, are probably handled similarly by the DUT. The hardware design team working with the verification team can identify ranges of values that are considered uniquely important. Values from these ranges must be verified in simulation, but any one value is not particularly important (unless the range is exactly one value). For example, consider the following five ranges for the packet length:

a) Minimum length (=0),
b) Within the lower half (0 < len < 512),
c) Middle length (=512),
d) Within the upper half (512 < len < 1024), and
e) Maximum length (=1024).

The implication, here, is that different DUT logic is activated by each of the ranges above. Therefore, verification should focus on generating packets of sizes over the mixture of important ranges. The packet class used in simulation, then, may extend from the base and set the constraints according to the plan:

```
class packet extends base_packet;
    constraint imp {
        len inside {
            0           // a) min length
            [1:511],    // b) lower half
```

```
                512,        // c) mid length
                [513:1023], // d) upper half
                1024        // e) max length
            };
        }
    endclass
```

The constraint `imp`, above, defines five ranges, or *bins*, with uniform probability. The constraint solver is free to consider each of the five bins equally. Then, once a bin is chosen, it is free to consider any value within the range with equal probability. Thus, the constraint defines two levels of probability. The first is the bin choice and the second is the value within the bin. Each of the five bins are given equal weight. Each of the values within the bins are also given equal weight. The hardware and verification teams could further refine the bin choice by specifying weights on each bin. However, once the bin is chosen, the constraint solver chooses any value within the bin with equal weight.

Functional coverage can be added to the `base_packet` class to report important bin lengths observed during simulation.

```
    class base_packet;
        rand len;
        constraint valid { … }
        covergroup imp_cg;
            coverpoint len {  // point is implicitly bound to len
                bins len_a = { 0 };           // a) min length
                bins len_b = { [1:511] };     // b) lower half
                bins len_c = { 512 };         // c) mid length
                bins len_d = { [513:1023] }; // d) upper half
                bins len_e = { 1024 };        // e) max length
            }
        endgroup
        function new();
            imp_cg = new;
        endfunction
        function void post_randomize();
            imp_cg.sample();
        endfunction
    endclass
```

In order to ensure that the packet length covered was actually transmitted to the DUT, the sample point would likely be outside the packet class by some DUT driver. For simplicity, we've indicated the sample at randomization. All extensions to the base class would inherit the cover group and its sample point (i.e., no missed sampling).

Random values, constraints, and coverage, coupled together, are the basis for constrained random verification. Implication #2, on page 2, identifies the minimum number stimulus generations required to cover *all* important bins. While planning improves the total number required, the probability that all important bins *will* be covered in any simulation is zero; random generation can only show coverage in the aggregate. As the total number of cover groups with points, bins and crosses grow, it is increasingly difficult to achieve complete coverage. Regression relies on the whim of the constraint solver and random seed.[1]

As verification engineers, we would like to randomize the packet length then block, or close, that bin from occurring again until all bins have been covered. For example, suppose the packet class was defined with not only the important bins in the `imp` constraint, but also additional constraints to block those bins.

[1] This is a simplification to describe the point. Ongoing research implies careful analysis of the cover space, such as with parameter domains [18] or coverage views [17], could statistically identify the set of tests required to achieve coverage goals.

```
class packet;
    constraint imp { len inside {0, [1:511], 512, [513:1023], 1024}; }
    constraint close_len_a { len != 0; }                // min length
    constraint close_len_b { ! len inside {[1:511]} };    // lower half
    constraint close_len_c { len != 512; }              // mid length
    constraint close_len_d { ! len inside {[513:1023]} }; // upper half
    constraint close_len_e { len != 1024; }             // max length
endclass
```

In this case, when the packet class is instantiated, all the `close_len` constraints must be disabled. Then, after each randomization, the correlating constraint must be enabled. However, SystemVerilog places some restrictions on changing the constraint mode [11]. The `constraint_mode()` built-in method is defined as both a task and a function. It is only when `constraint_mode()` is called as a task, and therefore called *from* a task, that the mode can actually be modified. Thus, because both instantiation and the post-randomize method are handled in functions, not tasks, the packet class cannot be self-contained.

```
class packet;
    function new() … endfunction
    function void post_randomize(); … endfunction
    task open_all_bins();
        close_len_a.constraint_mode(0);
        close_len_b.constraint_mode(0);
        close_len_c.constraint_mode(0);
        close_len_d.constraint_mode(0);
        close_len_e.constraint_mode(0);
    endtask
    task close_bin();
       if(len == 0)                     close_len_a.constraint_mode(1);
       elsif(len >= 1 && len <= 511)    close_len_b.constraint_mode(1);
       elsif(len == 512)                close_len_c.constraint_mode(1);
       elsif(len >= 513 && len <= 1023) close_len_d.constraint_mode(1);
       else                             close_len_e.constraint_mode(1);
    endtask
endclass
```

An external actor must execute the `open_all_bins()` task, above, after class construction and from within a task. Then, after randomization, an external actor must also call `close_bin()`, again from within a task. Of course, setting the constraints in the example above for all random variables in the verification environment is untenable. Instead, we would prefer an automatic method to achieve the same result.

In the following sections we describe inner workings of a satisfiability (SAT) solver as the basis of our approach.

A. Generalized SystemVerilog Constraint Formula

SystemVerilog specifies that all constraints are considered simultaneously in a conjunctive fashion [11]. In this discussion, we will ignore **soft** constraints. The formula for the constraint solver to solve on an instance of the `packet` class, considering both the `valid` and `imp` constraint constructs, is:

$$\begin{gathered}(len >= 0)\ \&\&\ (len <= 1024)\ \&\& \\ ((len == 0)\ \|\ (1 <= len <= 511)\ \|\ (len == 512)\ \|\ (513 <= len <= 1023)\ \|\ (len == 1024)).\end{gathered} \quad (2)$$

The structure of (2) follows the conjunctive-normal form (CNF) [13]:

$$\phi = \bigwedge_{i=1}^{L} \bigvee_{j_i=1}^{K_j} l_{j_i} \tag{3}$$

A *predicate*, p, is some Boolean term, a value. A *literal* represents a predicate in either its positive, p, or negative $!p$, form. A literal is said to be negative when representing $!p$, otherwise the literal is positive. In the equation above, l_{j_i}, is a single Boolean literal. A disjunction of literals (logical OR), $c_i = \bigvee_{j_i} l_{j_i}$, is a single Boolean *clause*. A propositional formula, ϕ, is the conjunction (logical AND) of all its clauses, $\bigwedge_i c_i$. Returning to the formula in (2), there are three clauses, as depicted in Table 1.

Table 1: Initial set of clauses in the example formula.

c_0	$\leftrightarrow$	(*len* >= 0)
c_1	$\leftrightarrow$	(*len* <= 1024)
c_2	$\leftrightarrow$	((*len* == 0) || (1 <= *len* <= 511) || (*len* == 512) || (513 <= *len* <= 1023) || (*len* == 1024))

In order to solve the formula, $\phi = c_0$ && c_1 && c_2, all three clauses must evaluate to **true**. However, the disjunction in c_2, that represents the `imp` constraint, is not in CNF. The two SystemVerilog **inside** constraints each contain a logical AND. Therefore c_2 must be distributed in order to propagate the logical ANDs to the clausal level. We show that, very cumbersomely, in Table 2.

Table 2: Derivation of clause c_2 to CNF.

c_2	$\leftrightarrow$	((*len* == 0) || (1 <= *len* <= 511) || (*len* == 512) || (513 <= *len* <= 1023) || (*len* == 1024))	
c_2	$\leftrightarrow$	((*len* == 0) || ((1 <= *len*) && (*len* <= 511)) || (*len* == 512) || ((513 <= *len*) && (*len* <= 1023)) || (*len* == 1024))	
c_2	$\leftrightarrow$	((*len* == 0) || (1 <= *len*) || (*len* == 512) || (513 <= *len*) || (*len* == 1024)) &&	(c_{2a})
		((*len* == 0) || (1 <= *len*) || (*len* == 512) || (*len* <= 1023) || (*len* == 1024)) &&	(c_{2b})
		((*len* == 0) || (*len* <= 511) || (*len* == 512) || (513 <= *len*) || (*len* == 1024)) &&	(c_{2c})
		((*len* == 0) || (*len* <= 511) || (*len* == 512) || (*len* <= 1023) || (*len* == 1024))	(c_{2d})

In Table 2, each sub-clause of c_2 has been individually labeled such that:

$$c_2 \quad \leftrightarrow \quad c_{2a} \,\&\&\, c_{2b} \,\&\&\, c_{2c} \,\&\&\, c_{2d}.$$

Therefore, clause c_2 in the formula may be substituted with its components to achieve ϕ in CNF.

$$\begin{aligned} \phi = {} & c_0 \,\&\&\, c_1 \,\&\&\, c_2 \\ = {} & c_0 \,\&\&\, c_1 \,\&\&\, (c_2) \\ = {} & c_0 \,\&\&\, c_1 \,\&\&\, (c_{2a} \,\&\&\, c_{2b} \,\&\&\, c_{2c} \,\&\&\, c_{2d}) \\ = {} & c_0 \,\&\&\, c_1 \,\&\&\, c_{2a} \,\&\&\, c_{2b} \,\&\&\, c_{2c} \,\&\&\, c_{2d}. \end{aligned}$$

Finally, Table 3 shows all the clauses in the example formula.

Table 3: Final set of clauses in the example formula.

c_0	$\leftrightarrow$	(*len* >= 0)
c_1	$\leftrightarrow$	(*len* <= 1024)
c_{2a}	$\leftrightarrow$	(*len* == 0) || (1 <= *len*) || (*len* == 512) || (513 <= *len*) || (*len* == 1024)
c_{2b}	$\leftrightarrow$	(*len* == 0) || (1 <= *len*) || (*len* == 512) || (*len* <= 1023) || (*len* == 1024)
c_{2c}	$\leftrightarrow$	(*len* == 0) || (*len* <= 511) || (*len* == 512) || (513 <= *len*) || (*len* == 1024)
c_{2d}	$\leftrightarrow$	(*len* == 0) || (*len* <= 511) || (*len* == 512) || (*len* <= 1023) || (*len* == 1024)

Next, we split out the predicates and their literals in each of the clauses to remove the less-than- and greater-than-or-equal constructs, and also enforce a standard ordering so that the variable, len, is on the left hand side of the operator: $(1 < len) \leftrightarrow (len > 1)$. While the resulting formula, in (4), appears to be getting unwieldy, in fact there are a large number of duplicate predicates. The constraint solver does not maintain each predicate separately.

$$\begin{gathered}\phi = ((len > 0) \,\|\, (len == 0))\ \&\&\ ((len < 1024) \,\|\, (len == 1024))\ \&\& \\ ((len == 0) \,\|\, (len > 1) \,\|\, (len == 1) \,\|\, (len == 512) \,\|\, (len > 513) \,\|\, (len == 513) \,\|\, (len == 1024))\ \&\& \\ ((len == 0) \,\|\, (len > 1) \,\|\, (len == 1) \,\|\, (len == 512) \,\|\, (len < 1023) \,\|\, (len == 1023) \,\|\, (len == 1024))\ \&\& \\ ((len == 0) \,\|\, (len < 511) \,\|\, (len == 511) \,\|\, (len == 512) \,\|\, (len > 513) \,\|\, (len == 513) \,\|\, (len == 1024))\ \&\& \\ ((len == 0) \,\|\, (len < 511) \,\|\, (len == 511) \,\|\, (len == 512) \,\|\, (len < 1023) \,\|\, (\mathrm{len} == 1023) \,\|\, (len == 1024))\end{gathered} \tag{4}$$

It is straightforward to see that if *len* == 0 then that predicate in clause c_0 and all c_2 sub-clauses are all true. Therefore, only unique instances of predicates are required in the formula; references are maintained from their respective unique literal within each unique clause. Table 4 shows all 13 unique literals and their predicates in ϕ.

Table 4: Unique predicates and literals in the example formula.

$l_0 \leftrightarrow p_0 \leftrightarrow (len > 0)$	$l_4 \leftrightarrow p_4 \leftrightarrow (len > 1)$	$l_9 \leftrightarrow p_9 \leftrightarrow (len > 513)$
$l_1 \leftrightarrow p_1 \leftrightarrow (len == 0)$	$l_5 \leftrightarrow p_5 \leftrightarrow (len == 1)$	$l_{10} \leftrightarrow p_{10} \leftrightarrow (len == 513)$
$l_2 \leftrightarrow p_2 \leftrightarrow (len < 1024)$	$l_6 \leftrightarrow p_6 \leftrightarrow (len < 511)$	$l_{11} \leftrightarrow p_{11} \leftrightarrow (len < 1023)$
$l_3 \leftrightarrow p_3 \leftrightarrow (len == 1024)$	$l_7 \leftrightarrow p_7 \leftrightarrow (len == 511)$	$l_{12} \leftrightarrow p_{12} \leftrightarrow (len == 1023)$
	$l_8 \leftrightarrow p_8 \leftrightarrow (len == 512)$	

All literals in ϕ are positive. It is possible to represent the negative of a predicate with a negative literal. For example, if (*len* != 511) were a predicate in the formula, then we could reuse predicate p_7 and introduce some new literal, l_m, as shown below:

$$!\, l_m \leftrightarrow p_7 \leftrightarrow (len == 511) \tag{5}$$

The advantage to the solver, with predicate reuse, is a reduction in memory use and overall complexity by reducing the number of predicate instances in a formula. However, there is a tradeoff on execution time required to potentially rewrite predicates to match a known similar predicate (for example, rewriting (*len* != 511) to become !(*len* == 511) in order to match p_7). However, this is not an issue facing the example formula ϕ.

B. Generalized SystemVerilog Constraint Solving

The SystemVerilog constraint solver may solve the formula by determining an assignment on all literals such that the formula is satisfied [13]. The set of all atoms in the formula, *Atoms*(ϕ), is the set of all unique Boolean literals, positive or negative. A *model* is an assignment to all literals that satisfy the Boolean formula ϕ (i.e., evaluates to **true**). The solver must successively try literal assignments to determine if a model can be found. If no model is found, then ϕ is unsatisfiable [13]. This means there is an inconsistency in the clauses provided that makes the formula evaluate to **false**. For example, consider the following formula:

$$\text{Eq. (4) \&\& } (len > 1024).$$

There is no solution for *len* that satisfies the clause (*len* > 1024) while at the same time restricting it to the range 0-1024 as required by the `valid` constraint in the `base_packet` class. In this case, the constraint solver will fail and the set of failing clauses is reported. Generally, the simulator vendors provide the inconsistency in a report generated during simulation [7, 8, 9].

Similar to the set of atoms, the set of variables in the formula, *Variables*(ϕ), is the set of all unique free variables. However, while the constraint solver directly attempts Boolean assignments on the set of atoms, it generally cannot associate those assignments with the underlying predicate. Therefore, solving the formula is usually attained via a two-step approach:

1. Find an assignment for all *Atoms* that satisfy ϕ,

2. Validate consistency on literal assignments within the context of the variable.

Step one, above, is handled by a satisfiability (SAT) solver. The SAT solver operates only in the Boolean domain without regard to the underlying predicate's domain. Step two is handled by a satisfiability modulo theory (SMT) solver that is context aware. For the example formula in (4), the theory of linear arithmetic solver ($\mathcal{LA}$) is employed to determine consistency of literal assignments in its domain. Multiple theories may be employed during solving depending on the requirements of the formula. For example, the theory of equality and uninterpreted functions ($\mathcal{EUF}$) may be employed for functional calls within ϕ.

Consider if the SAT solver indicates that l_0= **true**, l_1= **true**, and l_2= **true**. All clauses immediately evaluate to **true**. However, the theory solver would find an inconsistency:

$$\begin{aligned} l_0 &\leftrightarrow p_0 \leftrightarrow (len > 0) \leftrightarrow \textbf{true}, \\ l_1 &\leftrightarrow p_1 \leftrightarrow (len == 0) \leftrightarrow \textbf{true}, \\ l_2 &\leftrightarrow p_2 \leftrightarrow (len < 1024) \leftrightarrow \textbf{true}. \end{aligned}$$

The variable *len* cannot be both greater-than zero and equal to zero. Even though this is a valid assignment in the Boolean domain, it is not in the theory domain. The issue is both literals, l_0 and l_1, cannot be true at the same time. Therefore, the theory solver can introduce a new property:

$$l_0 \leftrightarrow !\, l_1 \implies (l_0 \rightarrow\ !\, l_1)\ \&\&\ (!\, l_1 \rightarrow\ l_0) \implies (!\, l_0 \parallel !\, l_1)\ \&\&\ (l_1 \parallel l_0).$$

Then, the theory solver communicates the new property back to the SAT solver by introducing two new clauses on ϕ using existing literals:

$$\begin{aligned} c_3 &\leftrightarrow (!l_0 \parallel !l_1), \text{ and} \\ c_4 &\leftrightarrow (l_1 \parallel l_0). \end{aligned}$$

The new clauses are simply appended to the formula, ϕ, and control passed back to the SAT solver. At this point, the SAT solver backtracks from the last assignment and, with the new property, attempts an assignment again[2]. In this manner, the SAT solver works in conjunction with SMT theory solvers to determine the final assignment to the sets of *Atoms* and *Variables*.

III. ENGINEERED CONSTRAINTS

In [12], we presented a type-parameterized random variable class that instantiated constraints in container classes on-the-fly. The crux of the technique took advantage of applying constraints across references and changing the reference at will. In Figure 1, the random variable is a class instance, as is the constraint. The dotted line represents a reference tying the two class instances together. Randomization occurs by applying the constraints to the random variable across the reference.

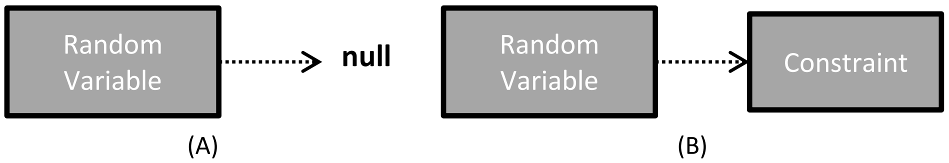


Figure 1: Random variable class with (A) undefined reference, and (B) reference to constraint class.

This approach differed from traditional constrained random techniques in that normally the constraints are applied either in the same class or through a class inheritance chain, as we presented in section II with the `base_packet` and `packet` classes. With engineered constraints, we restructure the constraint container classes to better align with general CNF form. In this manner, we can easily augment the constraint formula to solve.

[2] This is the Davis-Putnam-Logemann-Loveland (DPLL) procedure for incremental and learning SAT solvers [13].

In this section, we present engineered constraints as an implementation of the formula components: predicates, literals, clauses, and formulas.

A. *Engineered predicates*

The predicate typed base class forms the foundation for engineered constraints. In general, the predicate class models Boolean predicates in the constraint formula to be solved. However, it also provides a means to indicate a Boolean value on a literal, a clause, and the formula as a whole. Each of these sub-types is an extension to the predicate class.

The predicate class is typed to agree with the random variable type, refer to Figure 1. The class contains a single member, the randomized Boolean `result` flag, refer to Figure 2. When the result flag is constrained to **true**, then the underlying quantity must hold. Conversely, when the result is constrained to **false**, then the underlying quantity *must not* hold.

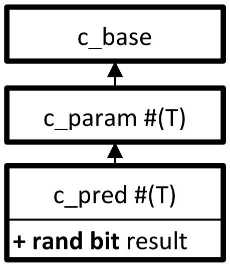


Figure 2: Predicate class hierarchy.

The `result` flag is never assigned a value; it is only determined through constraints in randomization. For example, consider the Boolean quantity for p_0 from NEED TO FIXME HERE!!!:

$$p_0 \leftrightarrow (len > 0).$$

This predicate indicates that the variable *len* must be greater-than zero. The predicate class, `c_pred`, models the predicate current Boolean value with the result flag, but not the underlying quantity. An extension to the predicate class ensures that when the result flag is true then *len* is greater-than zero. When the `result` flag is false then *len* is zero or less-than zero (dependent on the actual data type of *len*).

Referring again to Figure 2, two parent classes are defined for the predicate class. These exist as convenience classes to allow for easy passing of references. The type-parameterized base class, `c_param`, contains no members and no functions. The unparameterized base class, `c_base`, defines an interface as a set of functions. For example, the constraint can be converted to a string for debugging purposes, with or without current variable assignments. In this way, the user can access and cast at the base class without concern for the specific implementation type.

B. *Engineered literals*

The literal class implements Boolean literals from the formula. The class extends from the predicate class, inheriting the `result` flag, and adds two class members, refer to Figure 3. First, the literal contains a predicate class instance, declared with the **rand** attribute. Second, the literal class contains a flag to indicate a positive (`neg`=0) or negative (`neg`=1) literal.

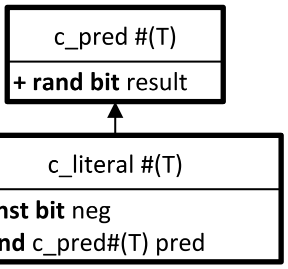


Figure 3: Literal class hierarchy.

For example, consider the following predicate and positive literal, from (4):

$$l_0 \leftrightarrow p_0 \leftrightarrow (len > 0).$$

The literal, l_0, is instantiated with a reference to the predicate class instance, p_0. When the literal is randomized, so too is the predicate. For l_0, the negative flag is zero, indicating this is a positive literal. One constraint is applied in the literal class, as in the code below.

```
class c_literal #(type T = int) extends c_pred #(T);
    constraint valid_result { result -> pred.result == !neg; }
endclass
```

The `valid_result` constraint ensures that when the literal is assigned the Boolean value **true**, then its predicate is assigned according to the nature of the literal, refer to Table 5.

Table 5: Literal to predicate assignment propagation.

	literal == true	**literal == false**
Positive literal (neg = 0)	pred = **true**	-
Negative literal (neg = 1)	pred = **false**	-

Notice in Table 5 that when the literal is **false**, then the predicate becomes a don't-care. Referring back to the formula in CNF form, refer to Eq. (4), on page 6, a single clause evaluates to **true** when any of its disjuncts (i.e., literals) evaluate to **true**. However, due to the nature of SAT/SMT solving, the other literals need not be assigned a value to have the clause evaluate to **true**. Basically, once the clause becomes **true**, then the solver no longer considers the remaining literals, if any. We model this in the literal class with the implication operator. If the literal is assigned a value **true**, then it is under consideration. Otherwise, the literal is not under consideration.

C. Engineered Clauses and Formulas

Clauses in the CNF structured formula are a disjunction of literals, while a formula is a conjunction of clauses. Their class hierarchies, in Figure 4, contain two members: a queue of literals or clauses, respectively, and a size. At randomization, the entire queue structure is randomized, including the size. As such, the size member ensures the queue size stays constant during randomization.

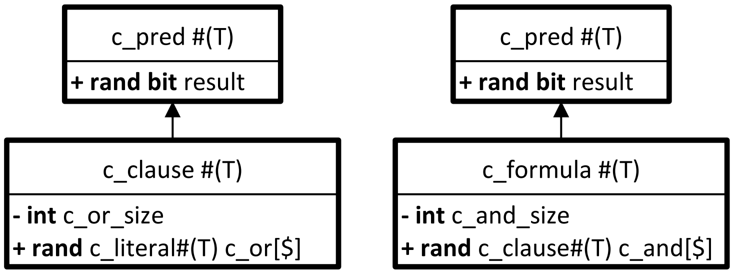


Figure 4: Clause and formula class hierarchy.

Consider a single clause in CNF. The disjunction (logical OR) of literals is modeled in SystemVerilog as a queue of `c_literal` class instances. At least one of those instances must evaluate to **true** (via SAT solver assignment). The essential action is to *reduce* a set of Boolean values to a single Boolean value. SystemVerilog has array reduction facilities [11], as in the code below.

```
class c_clause #(type T = int) extends c_pred#(T);
    constraint valid_size { c_or.size == c_or_size; }
    constraint valid_result {
        result -> c_or.or() with (member.result == 1);
    }
endclass
```

Each member of the `c_or` queue is, at base, a predicate and, therefore, implements the Boolean `result` flag. The `c_literal` member is activated when the `result` flag is asserted. The "or" array reduction method ensures that at least one member's `result` flag is asserted, thus making the literal resolve **true** and propagating that, according to the literal's type, to the underlying quantity (refer to Table 5).

The engineered formula is constructed in exactly the same manner, with the expectation that the SystemVerilog "and" array reduction method is employed. With the formula, all clauses in the `c_and` queue must have their Boolean `result` flag evaluate to **true**. This is propagated to each clause accordingly during randomization. In this way, each clause in the formula will evaluate to **true** during randomization, while at least one literal within each clause will evaluate to **true**.

The contents of the queues in both the clause and the formula may change between randomizations. Consider the case involving both the SAT and theory solver in section II-B. A new clause was created by the theory solver to indicate $l_0 \leftrightarrow\, !\, l_1$. Similarly, new clauses may be created and appended to the formula. For example, consider Eq. (4), representing the constraint formula $\phi = c_0 \;\&\&\; c_1 \;\&\&\; c_{2a} \;\&\&\; c_{2b} \;\&\&\; c_{2c} \;\&\&\; c_{2d}$. Suppose the first randomization on *len* generates a value assignment to zero. The following literal has therefore been assigned to true and its underlying quantity held:

$$l_1 \leftrightarrow p_1 \leftrightarrow (len == 0).$$

To ensure that this value is excluded from further randomizations, a new clause can be instantiated using the existing predicate instance in a new negated literal:

$$!\, l_{13} \leftrightarrow p_1 \leftrightarrow (len == 0).$$

Then, a new clause containing a single literal, a *unit clause*, is instantiated and appended to the formula:

$$c_3 \leftrightarrow\, !\, l_{13}.$$

The constraint formula now to be solved, referring to the clauses from Table 3 and new clause, above, is $\phi = c_0 \;\&\&\; c_1 \;\&\&\; c_{2a} \;\&\&\; c_{2b} \;\&\&\; c_{2c} \;\&\&\; c_{2d} \;\&\&\; c_3$.

Unit clauses force a specific action because the clause must evaluate to **true** in order to satisfy the formula. Here, we have taken an existing predicate, p_1, and instantiated as a new negative literal, $!\,l_{13}$. On the next randomization, the new unit clause will force the existing literal, l_1, to **false** and be taken out of consideration. Even as l_1 is no longer under consideration, the predicate still is because it exists in the formula in $!\,l_{13}$. Therefore, the quantity must hold in its negative form, !(*len* == 0), in order to satisfy the formula. In this way, the engineered constraint formula may be augmented during the course of simulation to guide randomization.

D. Engineered Constrained Values

The previous sub-sections have presented the CNF components of engineered constraints. We now present the randomized value classes. We have defined five types of randomized values: four according to SystemVerilog basic constraints[3] and one of our own:

- Constant: 0,
- Uniform contiguous range: **inside** { [1:511] },
- Uniform set: **inside** { 0, [1:511] },
- Weighted set: **dist** { 0 := 25, **inside** [1:511] := 75 },
- Non-rand sequence: **seq** [0, **inside** [1:511] , **dist** { 0 := 25, **inside** [1:511] := 75 }].

Refer to Figure 9 for class constrained values. Each class employs constraints and necessary support class members and fits within the engineered constraints CNF structure. Each constrained value class may be solved with a single randomize function call without the need for any additional setup. The constant valued class simply enforces the constant at random-time.

```
class c_val_const #(type T = int) extend c_val #(T);
    constraint force_const { const_value == value; }
endclass
```

The constant value is unique in the constrained value classes in that it is unaffected by the result flag. Essentially, the constant is modeling an actual number and, therefore, is always that number.

[3] The current implementation of engineered constraints does not support the **unique** constraint.

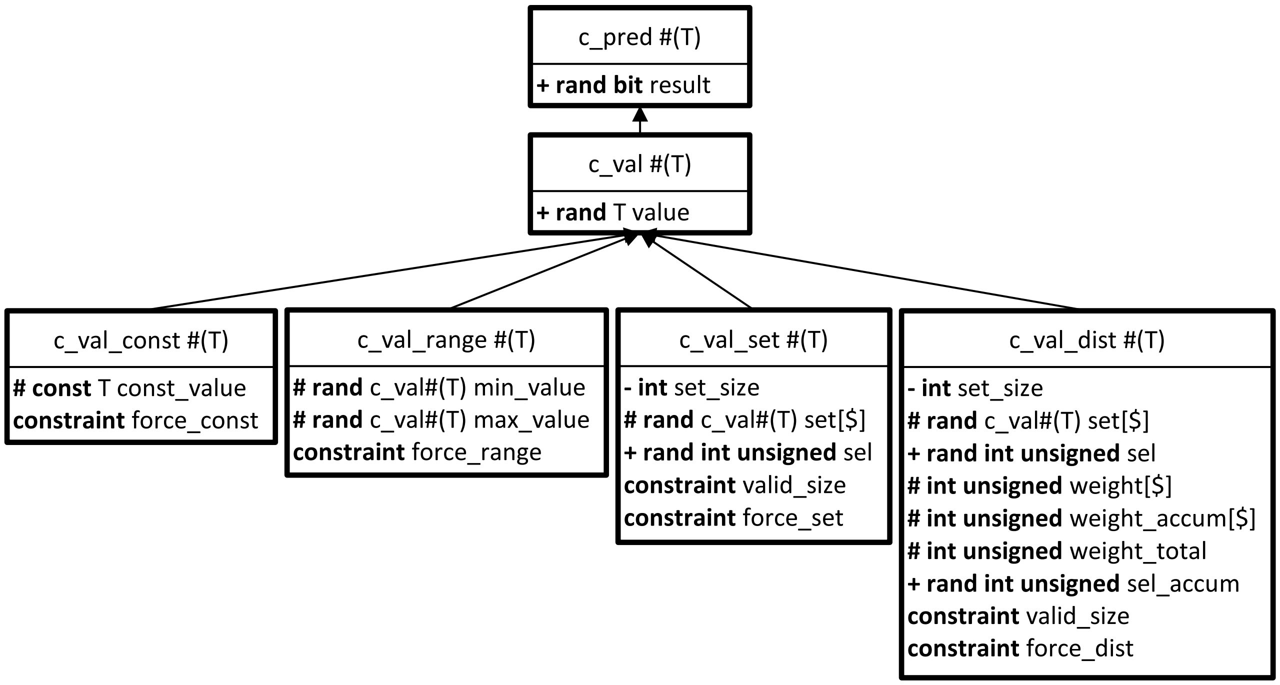


Figure 5: Constrained value container class hierarchies.

The contiguous range constraint enforces a uniform and contiguous range at random-time:

```
1   class c_val_range #(type T = int) extends c_val #(T);
2       constraint force_range {
3           solve result before value;
4           min_val.result == 1;
5           max_val.result == 1;
6           if(result) { value inside {[min_value:max_value]}; }
7           else       { !(value inside {[min_value:max_value]});
8       }
9   endclass
```

The ordering constraint on line 3, above, ensures the `result` flag is assigned (from the literal, clause, and formula above) before the local value is chosen. Notice, in Figure 5, that the bounds in the `c_val_range` constraint are themselves `c_val` instances. This allows for nested constraints on the bounds, lines 4-5 above. Therefore, the construct supports random bounds:

```
inside { [ inside {[0:3]}, inside {[7:9]} ] }.
```

SystemVerilog does support bounds that are inverted (i.e., max bound < min bound); however, this is most useful for non-negative data types (e.g., int unsigned, bit, or logic).

The uniform set constraint incorporates a selector and a queue. This differs slightly from using the native SystemVerilog set constraint. For example, the following `len` random variable will be assigned a value from the `sel` queue with uniform probability.

```
class a;
    rand int len;
    int sel[$] = '{0, 1, 2, 3};
    constraint valid { val inside { sel }; }
endclass
```

With engineered constraints, however, we want the option to block, or close, an entire literal rather than just a value. Therefore, the `c_val_set` constraint uses a selector to identify which set member is chosen at randomization.

```
1       class c_val_set #(type T = int) extends c_val #(T);
2           constraint valid_size { set.size == set_size; }
3           constraint force_set {
4               solve result before sel;
5               solve sel before value;
6               foreach(set[i]) set[i].result == 1;
7               sel inside { [0:(set_size - 1)] };
8               if(result)
9                   foreach(set[i]) (i == sel) -> value == sel[i].value;
10              else
11                  foreach(set[i]) value != sel[i].value;
12          }
13      endclass
```

Dissecting the code above, as in the constant constraint class, each of the queue's members is also `c_val` instances and, thus, nested constraints. Therefore, the foreach loop on line 6 ensures all members resolve positively. The ordering constraint on line 5 and the inside constraint on line 7 ensures the set selector is chosen before local value assignment. Lines 9 and 11 make it immediately apparent why the queue must be randomized and the selector chosen first. On line 9, an iteration through the set occurs, constraining the local value to the selected member's local value. Conversely, on line 11, the iteration constrains the local value to be *outside* the set. Recall that the `c_val_set` instance is a predicate and controlled by an external literal. A positive literal constrains the value to a range *inside* the set, as in: val **inside** { 1, 2, 3 }. A negative literal constrains the value to a range outside the set, as in: !(val **inside** {1, 2, 3}).

The weighted set constraint randomizes its selector based on user specified weights. In `c_val_set`, the selector is randomized over the length of the set; each member has equal weight. The `c_val_dist` class has an additional (non-random) set with weights attributed to each member.

```
1       class c_val_dist #(type T = int) extends c_val #(T);
2           constraint valid_size { set.size == set_size; }
3           constraint force_dist {
4               solve result before sel_accum;
5               solve sel_accum before sel;
6               solve sel before value;
7               foreach(set[i]) set[i].result == 1;
8               sel_accum inside { [0:weight_total] };
9               foreach(weight_accum[i]) {
10                  if((i+1) < set_size)
11                      (sel_accum < weight_accum[i+1]) -> (sel == i);
12                  else
13                       sel == i;
14              }
15              if(result)
16                  foreach(set[i]) (i == sel) -> value == sel[i].value;
17              else
18                  foreach(set[i]) value != sel[i].value;
19          }
20      endclass
```

Lines 8-14 in the code above constrain the selector in a weighted fashion. Note that the set of weights and the accumulated set of weights are maintained as the distribution constraint is formed. For example, consider the following weighted set constraint:

value **dist** { 1 := 10, 2 := 10, 3 := 80 }.

The `c_val_dist` class is instantiated and populated with each element individually. The `c_val_dist` instance member's content for the above constraint is shown in Figure 6. The randomized members are still not known.

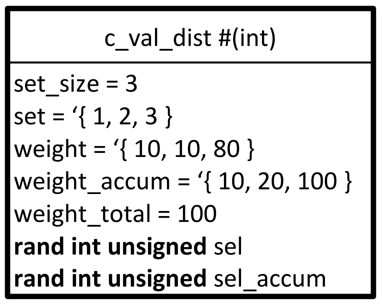


Figure 6: Example populated distribution instance.

Therefore, the `force_dist` constraint, referring back to the code for `c_val_dist`, randomly assigns `sel_accum` to some value in the weighted range first, line 8. Then, the foreach loop on lines 9-14 constrains `sel` based on the accumulated weights. Finally, lines 15-18 constrain the local value in the same fashion as for the `c_val_set` class.

The constraint sequence container class enables nesting constraints in a deterministic fashion. For example, consider if the first randomization should always be 0, followed by a uniform set, and finally a distribution:

val **seq** [0, **inside** [1:511] , **dist** { 0 := 25, **inside** [1:511] := 75 }].

The sequence ensures the above set of constraints is applied in a sequential fashion.

E. Engineered Boolean Binary Operations

The engineered constraint value container classes, in section III-D, enable engineered random variables. However, Boolean operations on those classes may be performed with engineered Boolean binary operator classes. Each binary operation, Figure 7, is a predicate in the formula. This allows the Boolean to operate positively or negatively, according to the literal assignment in the clause.

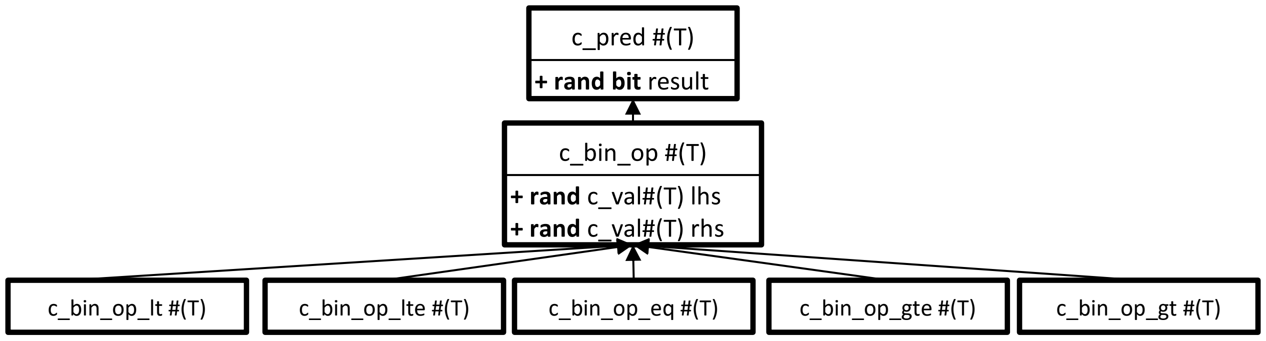


Figure 7: Boolean binary operation class hierarchies.

All binary operations contain a left-hand-side, `lhs`, and right-hand-side, `rhs`, as in the following:

`lhs` *OP* `rhs`.

Then, each operation-specific class extension implements the generalized constraint below, where *OP* is the actual operator implemented (e.g., `c_bin_op_lt` implements the less-than operator: `lhs < rhs`).

```
constraint valid_op
{
    if(result)
        (lhs OP rhs);
    else
        !(lhs OP rhs);
}
```

Combining binary Boolean operations with random value constraint container classes enables refinement of the constraint formula between randomizations. Furthermore, the Boolean operation is a predicate and, therefore, can implement both the positive and negative flavors. As such, it is not strictly necessary to define the entire set of operations in Figure 7. For example, the positive literal (*len* >= 0) is functionally equivalent to the negative literal !(*len* < 0). We have implemented each to ease debugging; the leaf class name directly implies the operation.

F. Example

Returning back to the original example in Eq. (2), we can implement the constraint as an engineered formula. In Eq. (4), we individually identified an example of each expanded predicate and literal that may be used by the constraint solver to find an assignment for *len*. With SystemVerilog we are able to take some liberties. Notably, instead of the CNF derivation found in (4) we can simply model the original equation from (2), representing the (min <= *len* <= max) predicates as instances of the `c_val_range` constraint class.

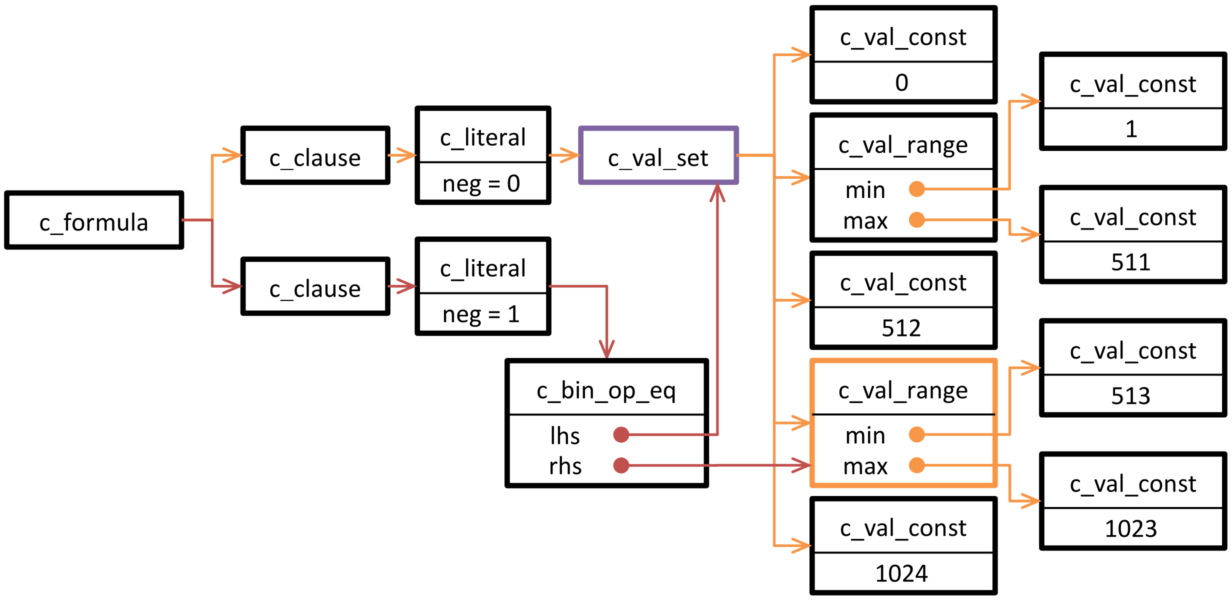


Figure 8: Equation (2) as an engineered formula: orange indicates the initial composition, red an additional clause.

The initial construction of Eq. (2) can be found by following the upper clause instance (orange arrows) in Figure 8. The *len* variable is represented by the `c_val_set` instance (in purple). In this formula, the value contained here is the randomized length:

$$len \leftrightarrow \texttt{c\_val\_set}.value.$$

Suppose that the first randomization on the formula yielded *len* = 768. This means that the predicate representing **inside** [513:1023] (orange c_val_range box) was selected, and its random value assigned. Recall that this bin was considered important for verification. Now that we have selected a value from the bin, we can close the bin by instantiating a negative literal in a new unit clause in the formula. Following the lower clause in Figure 8 (red arrows), we find that the literal re-uses the predicate instance for the negative literal. The formula to solve on the next randomization is:

(*len* **inside** { 0, **inside** {[1:511]}, 512, **inside** {[513:1023]}, 1024 }) && !(*len* **inside** {[513:1023]}).

If we continue in this manner, then on the sixth iteration randomization will fail because all the bins in the original formula would have been covered. We are then able to reset the formula by *popping* off the new clauses and return to the single initial clause. The engineered constraints formula can be manipulated in this way.

G. *Automatic Closure of Quantity or Value*

In a post-randomization method, we can employ automatic closure of a specific value or quantity. Referring back to the example in section III-F, the first randomization produced an assignment *len* = 768. The value constraint container that represents this selection is highlighted in orange in Figure 8, the `c_val_set` class instance in the formula. We can optionally close the entire range from further consideration by instantiating a new literal in a new unit clause in the formula. This action was taken in the example and is shown in Figure 8 by the red arrows. We can also optionally close the specific value by instantiating a new predicate, literal, and unit clause in the formula:

Eq. (2) && *len* != 768.

The random variable itself can handle this automatic instantiation. For c_val_set and c_val_dist class instances for the random variable under consideration, the last quantity selection is blocked, as in Figure 8. For other constraint class instances, the last value is blocked, as in the example above. Automatic closure is initiated by the random variable container class, in section IV. The action taken for automatic closure can be customized via class inheritance.

Adding minimum hit counts to random bins is one enhancement to automatic closure. For example, we could specify that **inside** {[513:1023]} must hit 3 times before it is closed. This is similar to specifying counts for functional coverage bins. Unless the count is achieved, the bin is considered uncovered. This feature is not yet supported in our implementation and is reserved for future work.

IV. Engineered Random Variables

The type-parameterized random variable container class brings together all the components of the engineered constraints approach. The class hierarchy, in Figure 9, has a current `value` as well as two predicate references. First, the `val` predicate is the value to be randomized in the formula. Recall from the example in section III-F that one constrained `c_val` (or class extension) instance represents the random variable. That `c_val` class instance is stored here as `val` for easy reference. The `formula` predicate is a randomized reference to the formula that randomizes and constrains `val`. Finally, the static constraint `factory`, one per type T in the simulation, is used to assist in instantiating and manipulating the `formula` (with both an API and a parser to process string constraints).

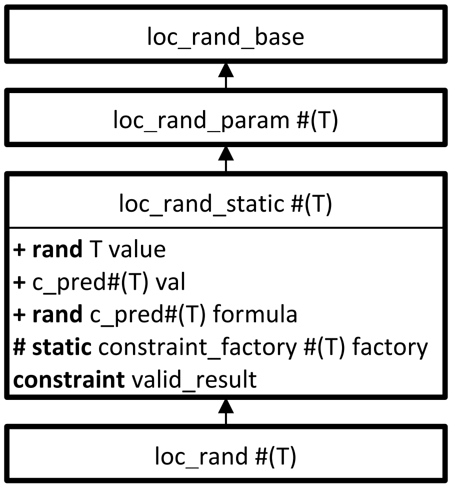

loc_rand_base
loc_rand_param #(T)
loc_rand_static #(T)
+ **rand** T value
+ c_pred#(T) val
+ **rand** c_pred#(T) formula
# **static** constraint_factory #(T) factory
constraint valid_result
loc_rand #(T)

Figure 9: Typed random variable container class hierarchy.

The random variable container class implements one constraint. The formula, clauses, and literals are all extensions of the predicate class. On line 4, below, the `formula` predicate's `result` flag is forced to **true**, thereby activating its constraints. On line 5, the random variable's `value` class member is assigned to the randomized value in the `val`.

```
1      class loc_rand_static #(type T = int) extends loc_rand_param #(T);
2          constraint valid_result {
3              solve formula before sel;
4              formula.result == 1;
5              value == val.value;
6          }
7      endclass
```

At `loc_rand_static` construction, `val` and `formula` class members are instantiated as the *default predicate*: `c_val`. This value constrainer has no constraint, as indicated in Figure 5. Therefore, the random variable instance may be randomized unconstrained.

The constraint factory may be used directly to parse a constraint specified as a string. Furthermore, it may be used to discover a constraint string on the command-line and/or in the UVM configuration database, refer to [12]. Finally, the factory has a public API that can be used directly to manipulate the formula by instantiating predicates, literals, and clauses. Alternatively, the formula itself is public and may be inspected and modified.

In Figure 9, there are `loc_rand_static` and `loc_rand` class definitions. The difference between the two is automatic dynamism in their interchangeable constraints. The `loc_rand` class can check, in its `pre_randomize()` function, for a new string constraint in the UVM configuration database and from the command-line. This action is not available in the `loc_rand_static` class. This is substantially the same as the random container classes in [12]. Note that we have found a severe performance degradation when this action is performed tens of thousands of times in a simulation. The root cause was found to be the structure and regular expression access to the UVM configuration database and not associated with the random container class [12]. Therefore, we suggest employing this action once, at the first randomization, to pick up any user changes on the constraint either from the UVM configuration database or command-line. Then this action should be disabled. Following this guideline we have found simulation performance not to be impacted at all, refer to [12] for performance statistics.

For example, consider Eq. (2). The following code implements its use in a random variable container class.

```
class my_comp extends uvm_component;
    loc_rand#(int) len;
    function new();
        len = new(“LENGTH”, this);
        len.push({“inside {”,
                  “0, inside {[1:511]}, 512,”,
                  “inside {[513:1023]}, 1024 }”);
    endfunction
    task run_phase(uvm_phase phase);
        len.randomize(); // pick up user constraints if any
        len.disable_dynamic();
        `uvm_info(“MY_COMP”, $sformatf(“len = %0d”, len.value),
                  UVM_MEDIUM)
    endtask
endclass
```

The random variable in the `new()` function above is instantiated with a name, `“LENGTH”`, and a scope. Then, a new constraint is pushed onto the random variable as specified in the string. This invokes the constraint factory in the random variable container class to parse and instantiate a new constraint on `formula`. Then `val` is assigned to the `c_val_set` instance that implements the outermost **inside** constraint. At randomization, `len.value` is constrained to `val.value` as determined by the constraint solver by solving `formula`.

H. Constraint Duality—Reverse Engineering

SystemVerilog constraints are formal equations to be solved by the constraint solver. However, a duality exists with constraints to allow both generation and checking [14]. The same constraint may be used to generate random stimulus as well as perform checks on observed values [11, 15]. Engineered constraints support both generation and checking. To check a value is applies to constraints, two steps are required:

1. Set the random variable `value` instance to the observed value,
2. Pop quantity and/or closure clauses from the `formula`.

First, the random variable instance's `value` must be assigned to the observed value. This is the same for constraint checking on any random variable. Second, if any automatic (or manual) closure clauses have been added to the `formula`, then they need to be popped before checking. This ensures the check on the `value` is consistent with the original, and presumably "valid," constraints. Then checking can occur. Referring to the `my_comp` example above:

```
len.value = observed_value; // save the observed to the random variable
if(len.randomize(0) == 0)
begin
    `uvm_error("INVALID_VALUE",
              $sformatf("Value (%0d) fails the constraint: %0s",
                        len.value, len.formula.constraint2string())
end
```

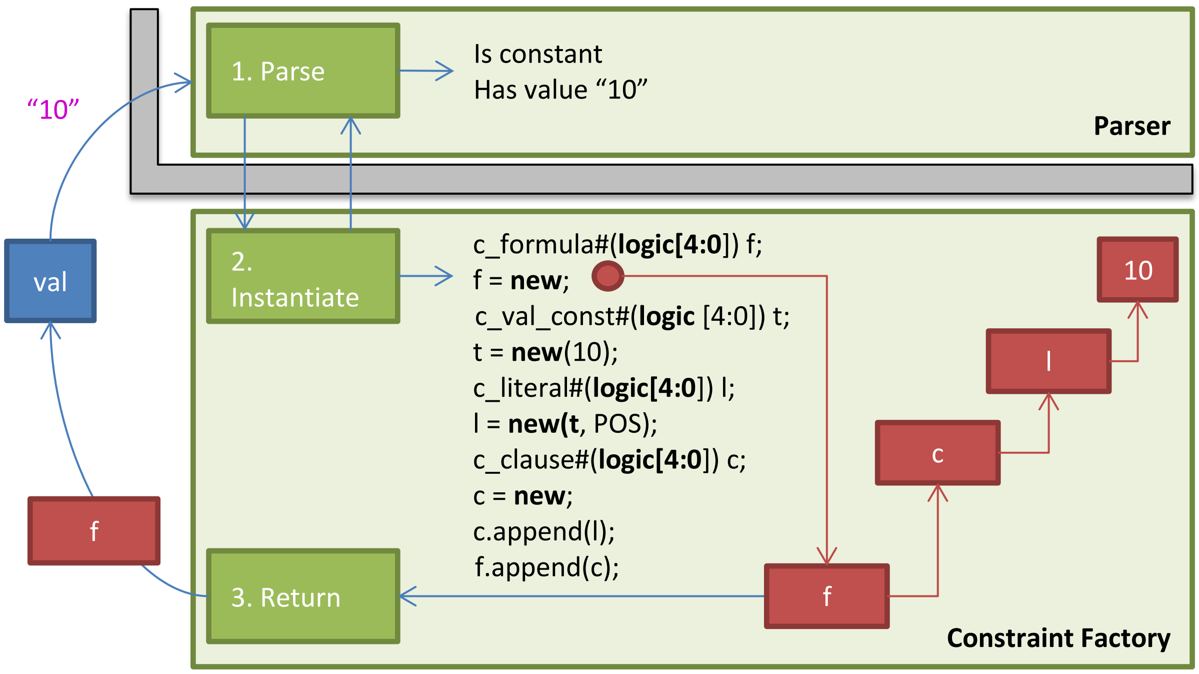
The argument to the `randomize()` call, 0, tells the constraint solver to check values according to the constraints. Essentially, the constraint solver, in the background, instantiates a new predicate and positive literal that constrains the value to the observed value:

Eq. (2) && (*len* == *observed_value*).

If the solver is unable to find the equation satisfiable then the observed value is invalid based on the supplied constraints (i.e., the `formula`).

I. Engineered Constraint Factory

The engineered constraint factory has two responsibilities. First, it must parse a constraint specified as a string and instantiate as a set of SystemVerilog engineered constraint container classes, as in Figure 10.



```
c_formula#(logic[4:0]) f;
f = new;
c_val_const#(logic [4:0]) t;
t = new(10);
c_literal#(logic[4:0]) l;
l = new(t, POS);
c_clause#(logic[4:0]) c;
c = new;
c.append(l);
f.append(c);
```

Figure 10: The typed constraint factory works with a Bison-generated C++ parser to instantiate the formula.

The constraint string `"10"` is passed to the random variable class, `val`. Then the constraint factory is invoked and the parser employed to dissect the string. The parser reads the input from the Left with a Right-most derivation (LR parser). This is the standard output of a Yacc/Bison generated parser [16]. The parser interacts with the SystemVerilog typed constraint factory to instantiate and compose the formula. When complete, the formula is returned to the random variable class.

V. Limitations, Workarounds, and Future Work

We have presented a flexible formula for engineered constraints. However there are some limitations to the current implementation. These can be grouped into items that are "difficult to support," and items that are simply "not yet supported."

- Not all SystemVerilog constraint constructs are supported,
- The **randc** construct is not supported, and
- Soft constraints are not supported.

Arguably, some of the above limitations may be worked around by instantiating new clauses in the formula. For example, the SystemVerilog **unique** constraint and the **randc** construct are not supported by the methodology. Both approaches ensure unique values are chosen over successive randomizations. This is accomplished in the methodology by instantiating a new negative literal unit clause onto the formula after each randomization that contains the last value chosen. Soft constraints are a tool to allow constraints to be declared but possibly violated, generally through class inheritance. We do not support this with our methodology because it seems to complicate the formula.

The following items are not yet supported in our implementation but can be achieved with the methodology.

- Multiple hits before automatic closure,
- Only a single variable may be defined in the string,
- Referring to other variables is restricted, and
- Not all modulo-theories are supported.

Automatic closure of a range is currently limited to a single count, as presented in III-G. Once the bin range is randomly generated, it can be closed. Future work will allow for multiple hits before the bin is closed, similar to functional coverage counts.

Currently, only a single variable may be specified in the constraint string. This need not be the case. Recall in section II that a unique set of free variables is maintained by the constraint solver. We could enhance the engineered constraints technique to also support multiple variables specified in the string constraint. Then the random variable class would maintain a queue of `c_val` instances that would be solved simultaneously. This is reserved for future work.

Currently, referring to other existing engineered random variables or class variables is restricted. This *is* possible using the **randomize with** construct, but somewhat tedious. As mentioned in section III-A, all predicate instances, including the formula itself, have a Boolean `result` flag that must be constrained externally. In section IV, the random variable class instance constrains the top (formula) level `result` flag. However, (a) the formula is simply a predicate instance, (b) its `result` flag is driven externally, and (c) the constraint formula need not be a strict-CNF structure. Therefore, we can simply instantiate the formula from one random variable class as a positive literal in a new unit clause in another. For example, consider the two independent formulas below:

$$\phi_1 = (len > 4) \qquad\qquad \phi_2 = (max\ \textbf{inside}\ \{\ 128, 256, 512, 1024\ \})$$

We can combine both by adding $\phi_2$as a new positive literal in a unit clause in ϕ_1. Then we can add an additional unit clause, enforcing a dependency between the two. The final constraint formula to solve would be:

$$\phi_1 = \underline{(len > 4)}\ \&\&\ \underline{(max\ \textbf{inside}\ \{\ 128, 256, 512, 1024\ \})}\ \&\&\ \underline{(len < max)}$$

Each clause in ϕ_1 is underlined. Both *len* and *max* would be solved simultaneously and considering the dependency.

Finally, our current library supports simple Boolean quantities between agreeing types. For example, a logic[31:0] type-parameterized random variable can compare greater-than, less-than, equal, etc., with a constant logic[31:0] typed value. In SystemVerilog, this is similar to:

```
rand logic [31:0] val;
constraint valid { val < 32; }
```

The numeric decimal constant, 32, is implicitly casted to a logic vector of 32-bits, then the comparison is made. Our methodology supports that and similar Boolean quantities. However, the following is also valid and reasonable:

```
rand logic [31:0] len;
constraint valid_length {
    len inside {[0:1024]};
    len % 4 == 0;
}
```

Our constraint library currently does not support the modulo operator nor any other (theory of) linear arithmetic operation ($\mathcal{LA}$). This is reserved for future work. Other theories may or may not be supported. For example, the theory of equality and uninterpreted functions ($\mathcal{EUF}$) cannot be fully supported—SystemVerilog does not support references to functions.

VI. CONCLUSIONS

SystemVerilog constraints, by their nature, are declarative. Run-time manipulations of elaborated constraints and random variables are possible. These, however, provide only limited control and must be called from SystemVerilog tasks. We have presented a methodology that composes the constraint formula as a set of engineered container classes following a conjunctive-normal form structure. Some limitations still exist in our approach, but the flexibility to augment the constraint formula on-the-fly purely within SystemVerilog functions is very appealing. We can guide randomization to achieve our goals while maintaining constrained random verification.

REFERENCES

[1] J. Bergeron, Writing Testbenches using SystemVerilog, Springer Science+Business Media, 2006.

[2] S. Iman, Step-by-Step Functional Verification with SystemVerilog and OVM, Hansen Brown Publishing Company, 2008.

[3] J. Bergeron, E. Cerny, A. Hunter and A. Nightingale, Verification Methodology Manual for SystemVerilog, Springer Science+Business Media, 2006.

[4] S. Rosenberg and K. A. Meade, A Practical Guide to Adopting the Universal Verification Methodology (UVM), Cadence Design Systems, Inc., 2010.

[5] A. Piziali, Functional Verification Coverage Measurement and Analysis, Springer Science+Business Media, 2008.

[6] P. James, Verification Plans: The Five-Day Verification Strategy for Modern Hardware Verification Languages, Kluwer Academic Publishers, 2004.

[7] Cadence Design Systems, Inc., "Incisive© Enterprise Manager Verification Planning," 2014.

[8] Mentor Graphics Corp., "Questa© SIM Verification Management User's Manual," 2014.

[9] Synopsys, Inc., "Verification Planner User's Guide," 2014.

[10] Accellera, "Universal Verification Methodology (UVM) 1.1 User's Guide," 2011.

[11] IEEE Computer Society, "SystemVerilog--Unified Hardware Design, Specification, and Verification Language (1800-2012)," New York, 2013.

[12] J. Ridgeway, "Interchangeable SystemVerilog Random Constraints," in *Synopsys Users Group*, San Jose, 2014.

[13] C. Barrett, R. Sebastiani, S. A. Seshia and C. Tinelli, "Satisfiability Modulo Theories," in *Handbook of Satisfiability*, vol. 185, A. Biere, M. Huele, H. van Maaren and T. Walsch, Eds., IOS Press, 2008, pp. 825-885.

[14] J. Yuan, C. Pixley and A. Aziz, Constraint-Based Verification, New York: Springer Science+Business Media, Inc., 2006.

[15] P. Marriott and J. Bromley, "Reverse Gear: Re-imagining Randomization using the VCS Constraint Solver," in *Synopsys Users Group*, San Jose, 2014.

[16] C. Donnelly and R. Stallman, "Bison, The Yacc-compatible Parser Generator," 2 April 2009. [Online]. Available: http://www.gnu.org/software/bison/bison.html.

[17] S. Asaf, E. Marcus and A. Ziv, "Defining Coverage Views to Improve Functional Coverage Analysis," in *Proceedings of the 41st Design Automation Conference*, San Diego, 2004.

[18] C. I. Castro, M. Strum and J. C. Wang, "Automatic Generation of a Parameter-Domain-Based Functional Input Coverage Model," in *LATW2010, 11th Latin-American Test Workshop*, 2010.

VII. Appendix A: Discussion on previous errors

In the original version of this paper we incorrectly translated Eq. (2) into CNF because we used the Boolean distribution law in an invalid manner.

We restate Eq. (2) below:

$$(\mathit{len} >= 0) \ \&\& \ (\mathit{len} <= 1024) \ \&\& \\ ((\mathit{len} == 0) \ || \ (1 <= \mathit{len} <= 511) \ || \ (\mathit{len} == 512) \ || \ (513 <= \mathit{len} <= 1023) \ || \ (\mathit{len} == 1024)).$$

In the original paper this was incorrectly translated into (with the offending literals highlighted):

$$((\mathit{len} > 0) \ || \ (\mathit{len} == 0)) \ \&\& \ ((\mathit{len} < 1024) \ || \ (\mathit{len} == 1024)) \ \&\& \\ ((\mathit{len} == 0) \ || \ !(1 > \mathit{len}) \ || \ !(\mathit{len} > 511) \ || \ (\mathit{len} == 512) \ || \ !(513 > \mathit{len}) \ || \ !(\mathit{len} > 1023) \ || \ (\mathit{len} == 1024)).$$

The two quantities (1 <= *len* <= 511) and (513 <= *len* <= 1023) were rewritten according to the following derivation:

$(\mathit{min} <= \mathit{len} <= \mathit{max})$	$\Longrightarrow$	$(\mathit{min} <= \mathit{len}) \ \&\& \ (\mathit{len} <= \mathit{max})$	
	$\Longrightarrow$	$!(!(\mathit{min} <= \mathit{len}) \ || \ !(\mathit{len} <= \mathit{max}))$	De Morgan's law
	$\Longrightarrow$	$!(\mathit{min} > \mathit{len}) \ || \ !(\mathit{len} > \mathit{max})$.	Distribution (wrong)

The issue here is that when employing distribution of the negation operator we did *not* affect the logical-OR. A correct derivation would be:

$(\mathit{min} <= \mathit{len} <= \mathit{max})$	$\Longrightarrow$	$(\mathit{min} <= \mathit{len}) \ \&\& \ (\mathit{len} <= \mathit{max})$	
	$\Longrightarrow$	$!(!(\mathit{min} <= \mathit{len}) \ || \ !(\mathit{len} <= \mathit{max}))$	De Morgan's law
	$\Longrightarrow$	$!(\mathit{min} > \mathit{len}) \ \&\& \ !(\mathit{len} > \mathit{max})$.	Distribution

Unfortunately, this provides our discussion no major benefit as the full formula is still not in CNF (although it could have demonstrated the use of negative literals). In this paper, we correctly translate the formula for Eq. (2), keeping only positive literals, into Eq. (4), but yielding a much larger Boolean formula for the satisifiability (SAT) solver.

Note, that this error in our theory section has no impact on the overall Engineered SystemVerilog Constraints classes as we need not consider the actual CNF formula (only the solver requires that information). The original, and simpler, Boolean formula in (2) suffices.