Systematic Constraint Relaxation (SCR): Hunting for Over-Constrained Stimulus

Debarshi Chatterjee, Spandan Kachhadiya, Ismet Bayraktaroglu, Siddhanth Dhodhi
Nvidia Corporation
2788 San Thomas Expy
Santa Clara, CA - 95051

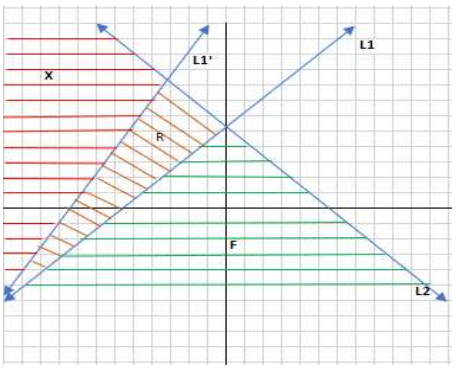
***Abstract*- Modern Verification Environments rely heavily on SystemVerilog (SV) constraint solver to generate legal stimulus [1]. Verification engineers write constraints based on design specifications to carve out a feasible region for stimulus that a Design Under Test (DUT) can support. The quality of such constraints often decides the quality of testing that is being done. The definition of legality of stimulus changes during the course of a project as new features get added to the design. It also changes across projects when certain old features are selectively enabled or disabled for a particular chip. Constraints typically get added on top of one another over projects and there is a significant burden of legacy code. In unit and integration level TestBenches (TBs), typically the number of such constraints could be anywhere from 1K-20K (or sometimes even larger). Constraints are also added temporarily to prevent tests from hitting known checker issues or known design bugs. These constraints are meant to be removed once the checker issue or RTL bug is fixed. What if these constraints were accidentally not removed? What happens if constraints were added with an incorrect understanding of the specifications? It is also possible that constraints were coded up with a correct understanding of the specification, but later the specification changed, and the constraints were not updated. Under these situations, a TB can contain over-constraints. Unlike under-constraints, over-constraints do not cause test failures and can silently degrade coverage and possibly impact simulation performance. The question then arises: How do we know if there are constraints which are over-constraining the Feasible Space and degrading quality of the stimulus? How do we identify redundant constraints which do not affect the stimulus, but cause performance degradation? In this paper, we propose Systematic Constraint Relaxation (SCR) – a technique that can automatically identify such over-constraints with minimal engineering effort. Some of these over-constraints can even escape functional coverage analysis.**

I. Introduction

Identifying over-constraints can improve testing and find potential bugs in the design. An automated solution to this problem can help various verification owners identify over-constraints without incurring a lot of human effort and intervention. Previous DVCON papers have highlighted several best practices for avoiding common mistakes in writing constraints [2][3]. However, to the best of our knowledge there has been no prior published work on automatically identifying over-constraints. So far, the standard approach in industry is to look at the coverage report and identify missing bins. When a bin for a cross-coverage is not hit in large number of random simulations, the first step is to check whether the coverage owners have identified accurately which bins are legal and which are illegal and marked them appropriately. In many cases, marking a bin illegal is done by the same DV engineer who coded up the constraints. If there is a common mode misunderstanding of the specification, the DV engineer might incorrectly mark the bin as illegal. In cases like this, an over-constraint might not be identified by coverage analysis. However, this does not happen all the time. Moreover, when a bin for a specific cross is not being covered, it does not immediately narrow the problem down to a constraint issue. Not being able to hit cross-coverage could be very well due to other testbench code unrelated to constraints. While coverage analysis is useful, it would be good to have a low-cost alternative to address this problem.

II. Technical Solution

To analyze this problem, let us suppose that the TB implementation has a Feasible Solution Space (F) constrained by a set of constraints C_1, C_2, … C_N. For visualization purposes, and without loss of generality, we assume 2 linear constraints $L_1 \leq 0$, $L_2 \leq 0$ constraining the feasible region to F in TB (Fig. 1a) Let us suppose that one of the constraints $L_1 \leq 0$ is over-constraining the stimulus. Let us also suppose that $L_1' \leq 0$ is the correct constraint and the correct feasible region is $F\ U\ R$, where U denotes Union. Let P denote the output of the randomizer obtained by relaxing the constraint $L_1 \leq 0$. P can fall in any of the three regions - F, R, X (Fig.1b). We can determine whether $P \in F$ or $P \notin F$. This can be done by checking if P satisfies the original set of constraints or not. Now, if $P \notin F$, how do we determine if $P \in R$ *or* $P \in X$? There is no definitive answer to this, since the region R is not well-defined, and we do not know if it even exists or not. However, we can use a heuristic to infer whether $P \in R$ *or* $P \in X$. To that end, let us note, if $P \notin F$ the TB expects the test to FAIL. Let S denote the status of a test by running a $P \notin F$. There could be 2 outcomes:



Solution Outcomes	TB Expects	Actual	Comments
Feasible(F)	Pass	Pass	Relaxation, but solution still in feasible region – nothing interesting!!
Feasible but over-constrained (R)	Fail	Pass	Case of interest – Do we have this on L1 relaxation? If so, L1 is over-constraining the feasible region
Outside (X)	Fail	Fail	Relaxation causes solution to be in infeasible region – nothing interesting

Figure 1. a) Feasible Region (F), Over-Constrained Region (R) and Infeasible Region (X) b) Solution Outcomes by Relaxing L_1 (Note: Actual column outcomes are not exhaustive)

1) $P \notin F$ and S==FAIL: In this case we cannot conclusively say if $P \in X$ or $P \in R$. This is because a failure can occur when P belongs to either region. For example, a) If $P \in X$, i.e., output of the randomizer falls in the region not supported by the design, we can expect the test to fail b) If $P \in R$, i.e., output of the randomizer falls in a region which is over-constrained by TB but supported by RTL. In that case also the test could fail – in the event there is an RTL bug in the over-constrained region or TB checker is unable to handle stimulus in this region. 2) $P \notin F$ and S==PASS: This could happen only if L_1 is over-constrained, and we can hypothesize $P \in R$. **Summary of the method: Relax L_i and generate P subject to all other constraints: If $P \notin F$ and S==PASS => $P \in$ R => L_i is over-constrained**

The basic idea of SCR is to automate the above process. A script searches for all files containing constraints in a TB. It then modifies each of those files by splitting up individual constraints into separate constraint blocks (Fig. 3), so that a single constraint C_i can be relaxed during a test run (by setting C_i.*constraint_mode(0)*). For every randomization of the object that contains C_i, we need to re-randomize the object with all variables of the object set to *rand_mode(0)* and C_i enabled. If the re-randomization fails, we conclude $P \notin F$, else we conclude $P \in F$. The trick here is to have the script insert custom code into *pre_randomize* and *post_randomize* sections in a way that the original code inside *pre_randomize* gets executed prior to the first randomization and the original code inside *post_randomize* gets executed after the second randomization, effectively rendering the second randomization process transparent to the TB code. If the second randomization finds $P \notin F$ we stop the process of constraint relaxation, so that there is at most one $P \notin F$ in a test. If the test is able to generate a single $P \notin F$ and the test eventually passes, then the script flags C_i as over-constrained. Finally, a post-processing script extracts the flagged constraints for review by the verification owner. The entire algorithm/process is explained in the Fig. 2 below. There are some subtleties w.r.t disabling constraints in presence of inheritance hierarchies and regarding save-restore rand_mode of variables before and after second randomization.

III. Common Questions

SCR is inconclusive when $P \notin F$ and we have a test failure. As mentioned earlier, this can happen when a test generating $P \notin F$ hits RTL bug in the over-constrained region R. Does that mean that SCR cannot really find over-constraints which were hiding RTL bugs? Typically, RTL bugs are sparse. Which means it can lie in the over-constrained region R, but the chances that all points in R will have RTL bug is very low. So, if we hit a $P \in R$ for which the test passes, the SCR flow would identify the over-constraint. Once the over-constraint is fixed, the RTL bug should be hit because now the feasible region would include the region R which had the RTL bug.

Why does the SCR flow generate a single $P \notin F$ and then stop constraint relaxation? A SCR test is only conclusive if the test passes. If we generate too many points outside the feasible region, likelihood of the point landing in illegal region X increases, thereby increasing the chances of test failure, which would render the test inconclusive.

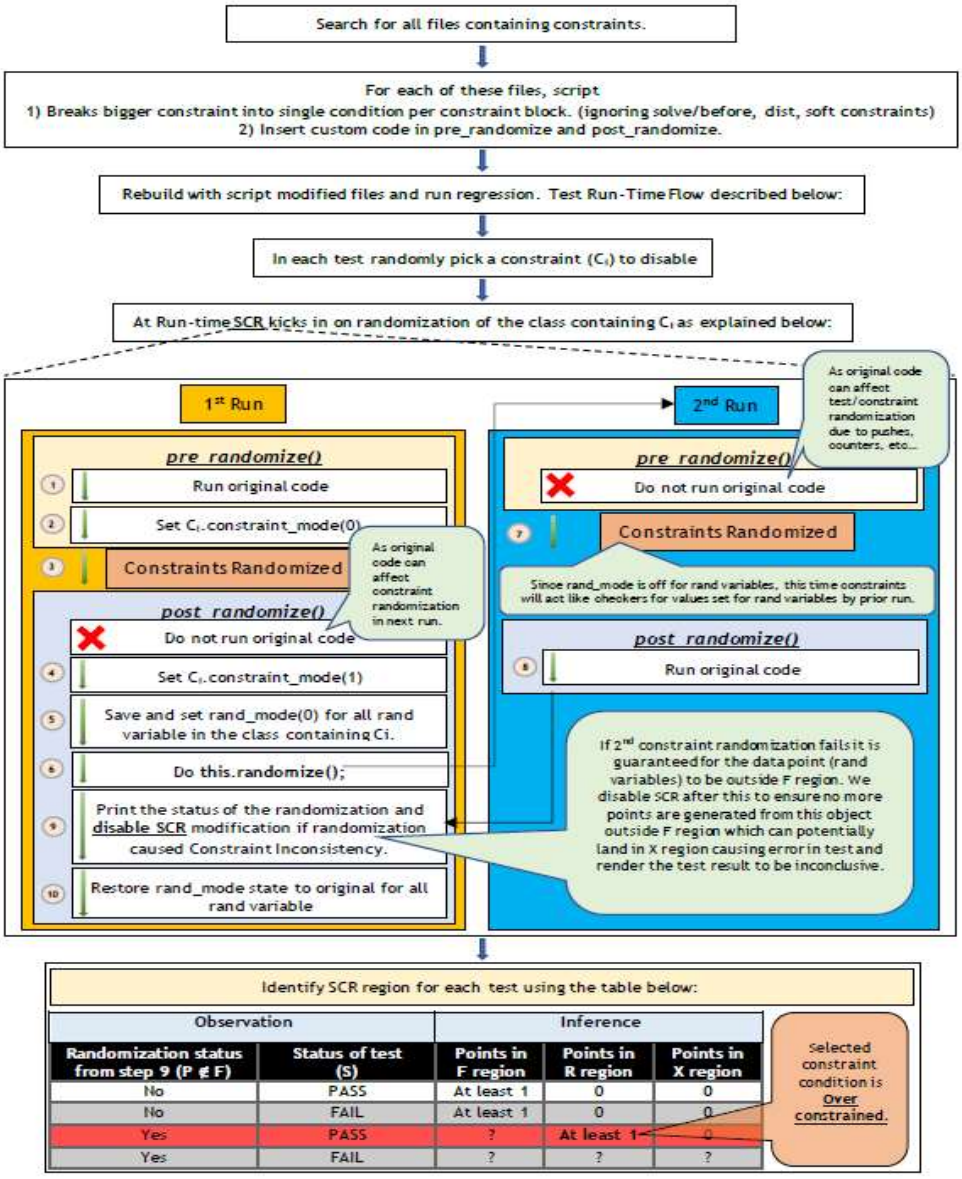


Figure 2. Flow chart explaining SCR Algorithm

Instead of relaxing the constraint and checking whether $P \in F$ or $P \notin F$, why does the SCR flow not invert the constraint to generate $P \notin F$? Theoretically, both solutions should work. In fact, inverting the constraint is guaranteed to generate $P \notin F$ – which is what we want. However, implementing the inversion procedure in the script becomes hard because it needs to account for all SV constraint syntax. To illustrate this better, refer to constraint block *var_randomization_3* in Fig. 3. The SCR script cannot statically unroll the foreach and split constraints inside *foreach* into separate constraint blocks because SV queue size is not known until runtime. So, if we follow the inversion methodology, the inverted code would look like "*foreach (payload[i]) {((inc_payload==1) && (i!=0)) ->payload[i]<=payload[i-1*]}". This means that all elements in payload[] need to violate the greater than constraint for the original constraint to be violated. We would not test the case to see if we have a passing test case when a single element in payload[] violate the constraint. Thus, although theoretically more appealing, the inversion procedure causes some loss of granularity in which we could cause constraints to be relaxed.

```
1  class SCR_demo;
2    rand bit cmd_type;
3    rand bit [7:0] length;
4    rand bit [31:0] payload[$];
5    rand bit inc_payload;
6
7  //  constraint var_randomization {
8  //    if (cmd_type == 0) {
9  //      length < 64;
10 //    } else {
11 //      length < 32;
12 //    }
13 //    payload.size == length;
14 //    foreach (payload[i]) {
15 //      (inc_payload == 1 && i!=0) -> (payload[i] > payload[i-1]);
16 //    }
17 //                                };
18
19   constraint var_randomization_0 {
20     if (cmd_type == 0) {
21       length < 64;
22     }
23   }
24   constraint var_randomization_1 {
25     if ( !( (cmd_type == 0)) ) {
26       length < 32;
27     }
28   }
29   constraint var_randomization_2 {
30     payload.size == length;
31   }
32   constraint var_randomization_3 {
33     foreach (payload[i]) {
34       (inc_payload == 1 && i!=0) -> (payload[i] > payload[i-1]);
35     }
36   }
37 endclass
```

Original constraints commented by SCR script

Modified split constraints inserted by SCR script

Figure 3. Sample Output showing original constraints commented out and split constraints added by SCR Script

ID	Fields					
ID1	F1	F2		F3	F4	F5
ID2	F1	F6	F7	F8	F9	F5

```
Constraint c1{
  F3 inside [x, y, z];
}
```

→

```
Constraint c1_fixed{
  ID==ID1 -> F3 inside [x, y, z];
}
```

Figure 4. A practical example of over-constraint not leading to coverage loss

IV. Results

We ran SCR to search for over-constrained stimulus in an integration level TB. The SCR script modified the classes containing constraints by adding custom code, split the constraints into separate blocks, and launched regression. The script was written in Python, and it took about 2 weeks of engineering effort to build it. From the data collected from 300 SCR tests, 22 cases of over constraints were identified. Running 300 tests took very minimal farm resource and was completed in an overnight run. These 22 cases were analyzed by verification engineers. For most cases this analysis was done by reviewing the code and specifications. For a few cases, it required re-producing the test-cases to

analyze which stimulus outside the feasible region caused the test to pass. Out of the 22 cases identified by SCR, 19 cases of over-constraints were not causing coverage loss.

Fig. 4 provides a practical example of an over-constraint identified by SCR which does not lead to coverage loss. Let us suppose, a hypothetical Design Under Test (DUT) with an interface that supports 2 packet types (or IDs) - ID1 and ID2. Let's also assume that a fixed number of most significant bits on the interface encodes the packet types in the ID field, and the encoding of the remaining fields on the interface depends on the ID field. This is shown in Fig. 4. Some fields like F1, F5 are shared across IDs while other fields like F2, F3, F6 etc. are unique to an ID. Let's suppose, the specification mandates that for packets of ID==ID1, the field F3 should be constrained within one of the fixed values {x, y , z}. Now, if a verification engineer chooses to code up this constraint as *c1* (shown in Fig. 4), then, SCR would detect *c1* as an over-constraint. This will happen because SCR will relax the constraint *c1* and will find a ID2 packet with F3 not in {x, y, z}. Since that will be a legal packet as per specification, the test will pass, and the SCR script will confirm a P outside TB defined feasible region which caused the test to pass. If the DV engineer coded the same constraint as *c1_fixed* (in Fig. 4), then SCR would not identify this as an over-constraint. As is clear from this example, SCR can also identify over-constraints that do not lead to coverage loss. Please note, if F3 was a shared field in ID1 and ID2, and F3 value was don't care for ID2 (as per spec), then the constraint *c1* would be an over-constraint leading to coverage loss. Most of the 19/22 cases of over-constraints not leading to coverage loss fell into this category. Once these were fixed, they were no longer picked by SCR.

We found 3/22 cases which were over-constraints that resulted in reduced coverage. We cannot share the details of the over-constraints without discussing micro-architectural details. However, we will broadly explain the nature of these over-constraints. One of the over-constraints which led to coverage loss was a known checker-issue and was tracked separately. The existence of two of the other over-constraints that led to coverage loss were not known to the DV engineers. They involved complex crosses of multiple variables and configuration scenarios. One of them was due to incorrect understanding of the specification and the other one was because of coding error in handling complicated crosses.

IV. Conclusion

In conclusion, SCR provides a novel methodology for solving a tough problem that silently plagues many testbenches. However, SCR is a heuristic and not a formal method. It does not guarantee that all over-constraints will be found. This is because even if an over-constraint region R exists, we might not be able to hit it by running a limited number of tests. Nevertheless, we feel that it is very powerful technique and the verification community in general can greatly benefit from adopting this technique to weed out over-constraints from their TBs.

References

[1] Constrained Random Simulation. In: Constraint-Based Verification. Springer, Boston, MA. 2006
[2] SystemVerilog Constraints: Appreciating What You Forgot in School to Get Better Results, Dave Rich, DVCon US 2020.
[3] The Top Most Common SystemVerilog Constrained Random Gotchas, Ahmed Yehia, DVCon-Europe 2014.