Discover Over-Constraints by Leveraging Formal Tool

Dongsheng Ouyang, Ray Zhang, Lucas Liu, Doris Yin, Wayne Ding
NVIDIA Semiconductor (Shanghai) Co. Ltd
Shanghai, China

***Abstract*-Over-constraints will cause RTL bug escape and project schedule slip. Traditionally, verification engineers rely on code review or simulation with coverage to find over-constraints, but this workflow is very time and resource-consuming and does not work very well. We propose to use formal to identify over-constraints. With formal, we can report random variables' reachable space, identify unreachable expressions in constraints, and report unreachable bins in stimulus coverage. With the help of these reports, we identify over-constraints in a short turnaround time during the constraint composition stage.**

I. Introduction

Constrained Random Testing (CRT) is a dominant and powerful stimulus generation methodology. Verification engineers write constraints as random specifications, and constraint solvers randomly generate legal stimuli based on these constraints. One of the most challenging problems of CRT is over-constraint. Over-constraints are constraints that are too narrow, ruling out certain legal input scenarios. Over-constraint is a common constraint quality problem that the below sources may introduce,

1) Low-quality constraints with dead code, unreachable conditions, and other problems.
2) Legacy constraints that do not match current design specs.
3) Temporarily added constraints as designs or checkers are not stable in the early stage of verification.
4) Scenario-specific or coverage-oriented constraints are wrongly added to the shared constraint database.

Over-constraints create two quality and productivity issues. The first is RTL bug escape. Over-constraints limit stimulus space that leads to low-quality stimulus consequently. This could hide RTL bugs. And the second is schedule delay. Over-constraints are usually exposed during the coverage closure stage, a relatively late stage of ASIC development flow. Fixing these constraints requires much engineering effort and test regressions. These two hazards put our schedule at risk.

Over-constraints degrade the quality of random stimuli, and verification engineers require a tool/method to identify over-constraints during the constraint composition stage. Traditionally, verification engineers rely on the below two methods to identify over-constraints,

1) Code review. Reviewers manually check if random constraints code matches requirements from design specs. Reviewers find it hard to find potential over-constraints from thousands of constraints.
2) Simulation with coverage. Detecting over-constraints by simulation is inefficient because it is hard to judge that interesting input is missing due to unreachability or low distribution.

Ref. [1] proposed a systematic constraint relaxation technique to identify over-constraints automatically. It is also a simulation solution. It assumes that the test always fails when over-constrained and requires a large set of tests. These limitations are too strong to apply this technique during the constraint composition stage.

II. Analysis

A random variable's random space is defined by all its reachable values. A reachable value is a legal value that satisfies all constraints on this variable. Figure 1 shows a code snippet with three random variables and their constraints, and we use a 3-dimension table to exhaust all these random variables' random space.

To identify over-constraints, the 1st step is to determine which variables are over-constrained. If we get a variable's random space, we can find all over-constrained variables by comparing the intended space against the reachable space.

But when a set of complex constraints limits random variables, it is impossible to precisely figure out its random space by the constraint writer or by the constraint reviewer. We should rely on a tool to exhaust a random variable's random space. The first tool that comes to mind is the random solver. The random solver randomly selects a solution value as the output of each randomize call. It will report constraint conflict if an illegal solution fixes any specific random variable. The random solver can validate a variable's random space, but it is very inefficient to exhaust all random possibilities. We need to find another tool that can explore random space efficiently and automatically.

```
rand bit [2:0] a;
rand bit [2:0] b;
rand bit [2:0] c;

constraint cst {
  a >=2;
  b > a;
  4'(a+b) == c;
}
```

Variable	Reachable space
a	{2,3}
b	{3,4,5}
c	{5,6,7}

Solution table

a/b	0	1	2	3	4	5	6	7
0	X	X	X	X	X	X	X	X
1	X	X	X	X	X	X	X	X
2	X	X	X	c=5	c=6	c=7	X	X
3	X	X	X	X	c=7	X	X	X
4	X	X	X	X	X	X	X	X
5	X	X	X	X	X	X	X	X
6	X	X	X	X	X	X	X	X
7	X	X	X	X	X	X	X	X

Figure 1. A random space example

III. TECHNICAL SOLUTION

As a powerful proof engine, formal is good at solving reachability problems exhaustively. But formal cannot understand random constraints semantics directly. This paper proposes a solution to convert our random constraints into formal readable constraints. With formal, a series of reports are generated, which help to identify over-constraints quickly. We propose a workflow to use formal to detect over-constrains depicted in Figure 2. Below are its key steps,

1) Constraint database generation. Rerun the random test with a UCLI (Unified Command-line Interface) [2] command file to explore and collect constraint information.
2) Convert random constraints into formal readable constraints by a script.
3) Run formal to prove the reachability of constraint expressions, variables' random spaces, and cover bins in stimulus coverage.
4) Generate constraint quality reports based on formal run results.

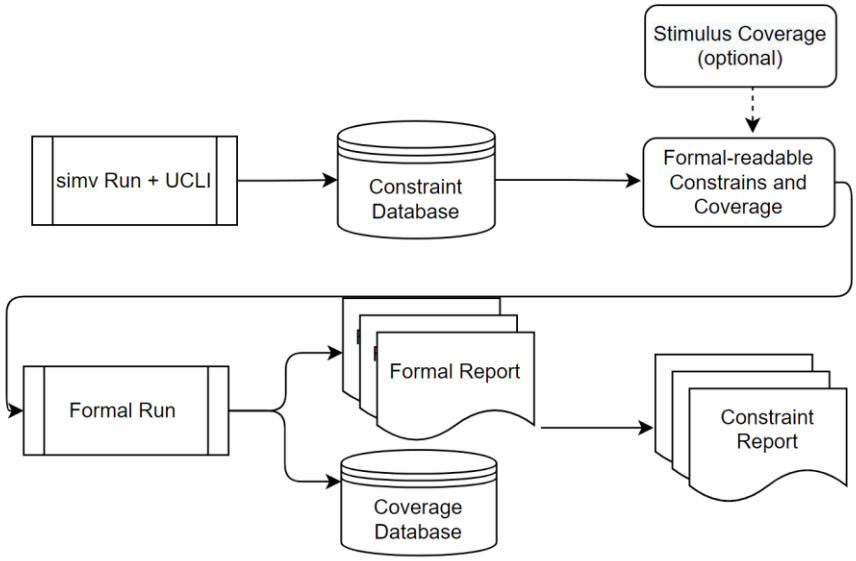

Figure 2. Formal workflow in random constraints qualification

A. Constraint database generation

For the Synopsys VCS simulator, we can rerun a test with a Synopsys UCLI command file to explore and collect all constraint-related information and dump it into a constraint database. The database stores all random partitions of all randomize calls. Each random partition is expressed as a SystemVerilog extracted test. In extracted tests, all complicated constraint structures like hierarchical, iterative, and inline constraints are flattened and replaced by simple constraint expressions. Table I shows a simple random constrained test and its generated extracted test. We can see that the extracted test's constraints are all constituted with boolean expressions, simplifying the conversion from random to formal readable constraints in the next step. Other miscellaneous information, like random variables' attributes and the original name in the source code, is also included in this database.

The database extracts all random constraint data that we are interested in from the random test. We can query its information to give verification engineers a comprehensive constraint report on their tests. This report includes items like randomize call number in test, TOP-N partition by random variable number, etc. With this report, verification engineers can identify the most critical and risky randomize calls or partitions in their tests. Generally, we only care about those randomize calls and partitions with large-sized random variables and complicated constraints.

Table I
EXTRACTED CONSTRAINT TEST OF A RANDOM TEST

Original Test with Random Constraints	Extracted Tests

class tile_owner_config;
 	
rand bit[1:0] headsettilemask[2];
// one Tile cannot be attached to multi Heads
constraint cons_tile_head_mapping
{
  foreach(headsettilemask[i]) {
  foreach(headsettilemask[j]) {
  (i!=j)->>
  (headsettilemask[i]&headsettilemask[j])==0;
  }
  }
}
 	
endclass
// …
tile_owner_config obj = new;
obj.randomize() with {
  // special case, fixed one-one mapping
  foreach(headsettilemask[h]) {
  headsettilemask[h] == 1<<h;
  }
};
// …
	class c_1_1;
	
	rand bit[3:0] headsettilemask_0_;
	rand bit[3:0] headsettilemask_1_;
	constraint cons_tile_head_mapping_this
	{
	  (headsettilemask_0_ & headsettilemask_1_) == 4'h0;
	  (headsettilemask_1_ & headsettilemask_0_) == 4'h0;
	}
	
	constraint WITH_CONSTRAINT_this
	{
	  headsettilemask_0_ == (1 << 0);
	  headsettilemask_1_ == (1 << 1);
	}
	
	endclass
	// …

B. Convert random constraints into formal readable constraints

In the previous step, we save each partition's constraints in the constraint database as a SystemVerilog test. However, formal tool still cannot comprehend these constraints directly. In formal verification terminology, a constraint is an assume property to limit the inputs of design. If we want to leverage formal, we should do a conversion from SystemVerilog random constraints into their equivalent SVA (SystemVerilog Assertion) assume properties. These properties are as simple as a single cycle boolean expression. With the format of SVA assume property, the formal tool can read those constraints as the input.

Table II lists typical random constraint operators and their equivalent SVA properties. Foreach iterative constraint, array reduction iterative constraint, and hierarchical constraint are excluded from this table as we have flattened them in the extracted tests. Based on this table, we can do this conversion automatically by a script. Currently, the below constraint declarations/operators are supported in our constraint conversion script,

- **`inside`** operator
- **`dist`** operator
- implication operator
- **`if-else`** style constraint

The output of this step is a formal readable DUT (Design Under Test): random variables are converted to its design signals, and random constraints are converted into SVA constraints on these signals. With the formal DUT, we can run formal to generate constraint qualify reports.

Table II
RANDOM CONSTRAINT TO SVA CONVERSION TABLE

random constraints	SVA properties
`v inside {5, 10};`	`v inside {5, 10};`
`x dist {100 := 1, 200 := 2, 300 := 5}; *`	`x inside {100, 200, 300};`
`mode == little -> len < 10;`	`mode == little |-> len < 10;`

`if (mode == little)`
`len < 10;`
`else if (mode == big)`
`len > 100;`
	`if (mode == little)`
	`len < 10;`
	`else if (mode == big)`
	`len > 100;`

`unique {a, b, c};`	Unsupported
`solve s before d;`	Ignored
`soft length inside {32,1024};`	Ignored

*Weight of zero is not supported.

C. Run formal to generate the unreachable code report

The 1st constraint qualify report we can generate based on formal results is unreachable constraint expression report. Formal can help to prove the reachability of all SVA assume properties, and we can trace back these SVA properties to the original constraint expressions and mark them as reachable or unreachable. In the unreachable code report, all unreachable constraint expressions are listed as a table, and the constraint writer can identify and fix unexpected unreachable expressions. These unreachable expressions are a source of over-constrained and under-constrained stimuli.

Figure 3 shows a simple example of using unreachable code reports to find an over-constraint introduced by an overflow in constraint expression. a and b are all 4-bit random variables. The report tells the constraint writer that "a+b" cannot be greater than 4'hf, it seems not reasonable, but we will find the expression gotcha here: "a+b" is evaluated as a 4-bit expression, and it cannot be greater than 4'hf.

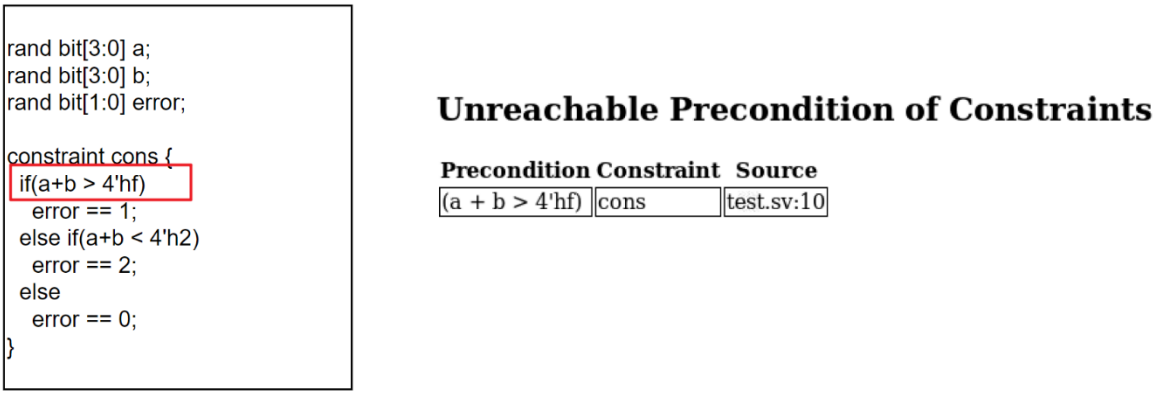

Figure 3. Unreachable code report

D. Iteratively run formal to generate the random space report

To explore a random variable's random space, we can divide each variable's full range (the space defined by its bit-width) into serval sub-ranges. And each sub-range can be represented by an SVA cover property. With the constraints in formal DUT from step B and each random variable's cover properties, we can run formal to prove the reachability of each sub-range. For the unreachable sub-ranges, we can rule them out from the variable's reachable random space. For the reachable sub-ranges, with a divide-and-conquer strategy, we can finally get the accurate reachable space for each random variable.

Besides, we use a simulation regression to offload the workload on formal. We run a simulation regression of the extracted test before formal runs. The regression results will tell us which part of the random space was hit by the simulation, and then formal can focus on the reachability of those sub-ranges that the simulation has not hit. Results show that with this hybrid solution, we can speed up the random space exploration's overall turn-around time by 2X.

The output of this step is a table of all random variables' random space. In this report, unreachable spaces will be marked red, and reachable spaces will be marked green. We developed a smart algorithm to divide random variables

into two categories: discrete variables with all possible values and continuous variables with a min~max range (Figure 4). Reviewers can quickly check variables' random space with design spec requirements to find over-constrained variables.

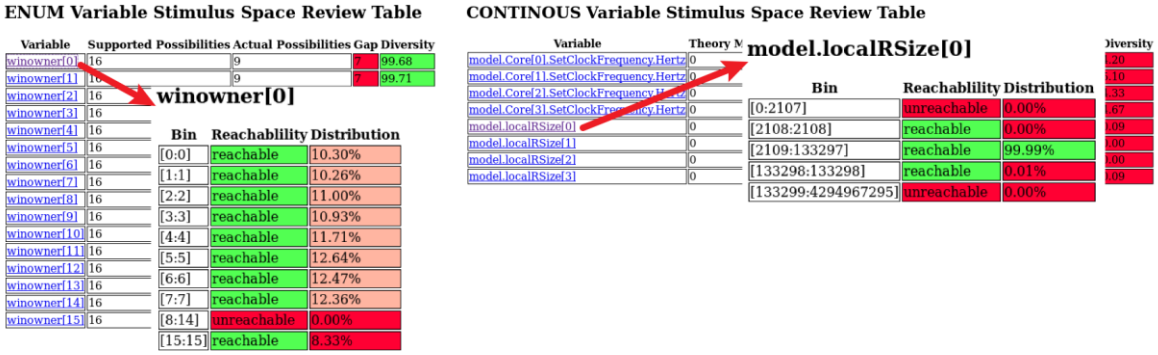

Figure 4. Random space report

E. Run formal to generate the unreachable stimulus coverage report

The test writer adds stimulus coverage to answer whether interested random states are covered or not in simulation. Stimulus coverage setup random goals explicitly for random stimuli. Any unreachable bin in stimulus coverage means a target's random state is unreachable. In this case, we also rely on formal to tell us any bins are unreachable, and adding more tests does not help.

Figure 5 demonstrates a simple example of formal running with stimulus coverage. The unproved failure target in formal results exposed a coverage hole that cannot be hit based on existing constraints. Users can check unreachable bins in coverage reports and find out the unwanted constraint expressions that lead to this failure. The entire process does not need to involve any design units and simulations. It only contains a single formal run.

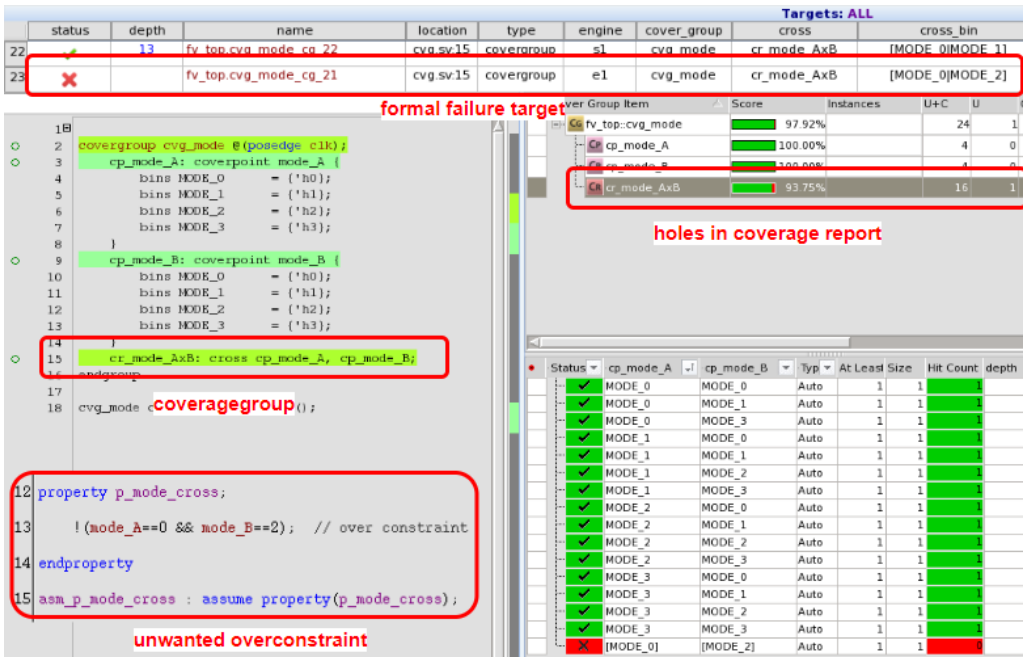

Figure 5. Stimulus coverage report from formal running

III. RESULTS

With the below three constraint qualify reports, verification engineers can identify over-constraints with a short turnaround time.

1) **The unreachable code report** lists all invalid constraint expressions. The constraint writer should investigate whether these unreachable expressions are intended or not. And then constraint writer should root cause each unexpected unreachable expression. Typical causes of their unreachability could be hard coding, typos, incorrect bit-width of expression, and legacy constraints for obsoleted features. The unreachable code report focuses on the quality of constraint expression. It can help constraint writers do a good QA for their code before tests and testbenches are fully ready. To identify over-constraints and other problematic constraints, the constraint writer should glance at this report to pick low-hanging fruits.
2) **The random space report** lists all random variables' random space. The designer, the architecture, and the verification engineer should all review this report and compare each random variable's intended space with reachable space. Any mismatch means that the random variable is potentially over-constrained or under-constrained. After the report review, the verification engineer should dig further to determine the constraints that lead to this mismatch. The random space report is the most crucial constraint qualify report and we recommend that all stakeholders review and sign-off this report at the RTL final stage.
3) **The unreachable stimulus coverage report** lists all unreachable cover bins in stimulus coverage. Reviewers can judge whether those unreachable bins are over-constrained cases or can be excluded by a waiver. One important thing we should mention here is the random space report can only report the reachability of single variable states. As we define cross bins in stimulus coverage, this report can report the reachability of a complicated random scenario that evolves multiple random variables.

We have integrated the whole constraint qualify report generation workflow in our internal constraint qualify tool NVRO (Nvidia Random Optimizer). With the help of Jenkins [3], we can run NVRO on a random codebase periodically and automatically to monitor the quality of constraints (Figure 6). With an additional Diff APP in NVRO, any constraint quality degradation between 2 versions will be reported in time.

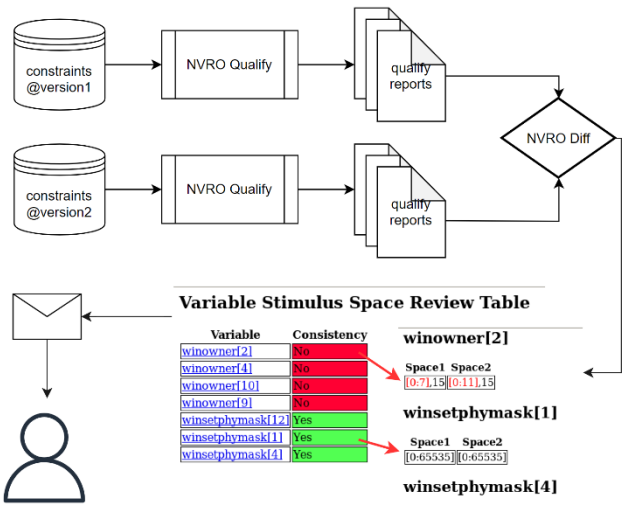

Figure 6. Monitor the quality of constraints by NVRO

Experimental results show that this workflow has good scalability and performance. We have tried this workflow on NVIDIA's Display IP. It has a central random generator with one randomize call. Table III lists a comprehensive report on this random generator with over 15K random variables and 6762 random partitions. We only generated constraint qualify reports for its top-5 partitions as other partitions are too small and less likely to have over-constraint problems. The Jenkins log shows that the whole workflow (from constraint database generation to qualify reports generation) spends less than 3 hours on average. Another thing that we need to clarify here is that the run-time of the

whole workflow is not related to the design's size or the test's simulation time. It is related to the size of the constraints network, as we do not run formal with DUT.

Table III
THE OVERALL REPORT OF A RANDOM GENERATOR

Item	value
#Random partition	15670
#Random variable	6762
TOP-N partitions	[1964, 33, 33, 33, 33]

VI. CONCLUSION

We proposed a new solution to over-constraints detection by leveraging the formal tool in the simulation world. It creates a working model with a speed-of-light turnaround from constraint directly to report. It is one decade more advanced than the original turnaround from constraint writing to test creating, to coverage regression, and finally to coverage report analysis to fix possible constraint issues.

We have integrated the whole workflow into one internal tool. We applied this tool to an NVIDIA IP's random generator with over 15K random variables. The tool can generate random space reports and unreachable code reports in one night's turn-around time. It helps us find several over-constraints during the constraint composition and coverage sign-off stage.

REFERENCES

[1] Systematic Constraint Relaxation (SCR): Hunting for Over-Constrained Stimulus, Debarshi Chatterjee, DVCon US 2022.
[2] Synopsys, VCS® Unified Command Line Interface User Guide R-2020.12, December 2020, https://www.synopsys.com/
[2] https://www.jenkins.io/