# RTL Verification: Automated Statement Coverage and Mutation Testing

ECE5534: Term Project

Sonal Pinto
Department of Electrical and Computer Engineering,
Virginia Tech,
spinto64@vt.edu

May 7, 2016

***Abstract* — *This report presents techniques for Automated Statement Coverage for RTL verification. Verilator, an HDL simulator is used to translate Verilog models into C++, is not only fast, but enables easy verification. Three algorithms,* rand1*,* rand2 *and* saga*, inspired by standard software verification techniques are developed for Statement Coverage. The first is fully random, while the second is guided-random. Lastly,* saga *uses a heuristic optimization, Genetic Algorithms to search the input space. Finally, the generated testbenches are combined and validated using Mutation Testing to show the viability of Automated Statement Coverage for functional testing.***

## 1 Introduction

Design verification is not only an important step in the chip-design cycle, but often requires as much resources (in labor and hardware) as the IP design stage [1]. In current times, it might even be considered imperative to perform extensive verification of design modules at every stage, because the cost of re-tape-out, in case of a fatal design flaw, is greater with decreasing technology nodes. It would be advantageous to automate verification at the RTL (Register-Transfer Level) phase itself. Rapid testing and verification earlier in the design cycle would enable catching and fixing of bugs at a lesser cost, and therefore, faster overall time-to-market.

RTL testing and verification emphasizes that techniques will be deployed at a higher abstraction level, than the standard gate-netlist level (which would be closer to the final physical design). This would allow RTL designers to detect and amend IP design choices before the time costly process of synthesis. For this purpose, we shall considering translating the model into C++ for faster verification and Testbench generation, instead of slower HDL simulators [2]. Verilator is a free open-source Verilog HDL simulator. It compiles synthesizable Verilog into C++ or SystemC models. It claims on-par, if not better simulation performance than commercial Verilog simulators [3].

Once the Verilog models are converted to C++, aside from boosted simulation performance, we are able to employ standard software verification techniques to build testbenches. Statement Coverage is a basic software test metric where the goal of the test at hand is to ensure an execution of the program through every line of code, in every branch. In terms of digital design testing, we can translate the same goal as effectively engaging every part of the design. This could possibly help identify unreachable parts of the design, or provide testbenches that help identify working parts or regions in minimal time during the fabrication process.

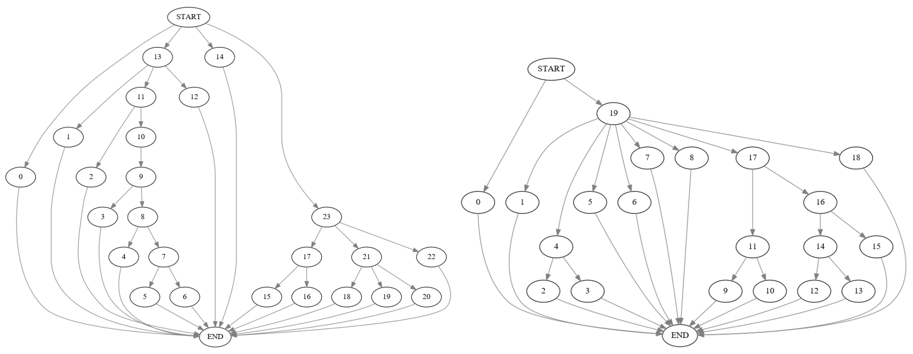
In further sections, three different methods for automated generation of testbenches for Statement Coverage of translated RTL models are described and contrasted. Two of these techniques rely on random test input vector generation, while the third technique utilizes a heuristic search (Genetic Algorithm), in an attempt to find optimal tests. The input search space (primary inputs to the model) is explored in order to find the solution. In order to contrast between solutions, and to ascertain the healthier solution, the Coverage Flow Graph is extracted (a subset of the Control Flow Graph restricted to coverage points) and the paths that each solution engage are compared. The final result is a Testbench, which is a set of individual tests that attempt to achieve the goal of full Statement Coverage. Finally, the testbenches generated from each of the techniques is held as a single Testbench, and evaluated for its functional testing viability by Mutation Testing. Functionally mutated models are tested with this single testbench, and an attempt is made to discover these functional changes. A good Testbench should be able to detect as many mutants as possible.

## 2 Background

In this section, the Coverage Flow Graph and its utility is described, followed by an overview of Genetic Algorithms, and an introduction of its application to Automated Statement Coverage. The Section is closed with a description of Mutation Testing and its evaluation metrics.

### 2.1 Coverage Flow Graph

The Coverage Flow Graph (CovFG) is a subset graph of the Control Flow Graph, indicating only information pertaining to coverage points. In the graph, the vertices represent the coverage points, and the edges represent the depth of the coverage point. A path in this graph indicates the order in which the coverage points would be engaged in any given execution of the model. Every operation sequence (function) of the Verilator model begins at single node, `START` and terminates at `END`. In general, vertices or nodes towards to the root would have most number of enumerated paths, while those closer to the terminal node would have lesser number of paths passing through them. A child node requires that the parent node be visited. Therefore, the coverage points that lie deeper in the CovFG, will require repeatable execution of the parent coverage points. The CovFG, does not hold any predicate information denoting the requirement



(a) uart (b) b07

Figure 1: Coverage Flow Graphs

for an execution to trace along a particular path. Fig.1 shows some examples of CovFG for simple models.

## 2.2 Testbench Structure

A `testbench` is a set of `tests`, and a typical execution requires that the model be reset (synchronous or asynchronous) before each test is applied. The primary inputs to a model, are coalesced to a single data structure called `inputvector`. During each time step or cycle, the bits of the `inputvector` are separated as per the input bit requirement of the model, and applied to the input variables of the Verilator model object. Fig.2 shows the inputvector specification for uart, where the 6 Primary Inputs are represented as a simple array of 13 bits. During inputvector application on the model, these bits are separated as per their boundaries specified in `INPUT_SPEC`.

**inputvector** : an array of bits
**test** : a list of inputvectors
**testbench** : a list of tests
**coveragestate** : an array of the coverage point bins

The `coveragestate` array mirrors the internal coverage state bins inside the Verilator model, and is updated each time an execution traces through a block that the

```
/**
 * The input vector consists of 13 bits
 * ld_tx_data : 1
 * tx_data : 8
 * tx_enable: 1
 * uld_rx_data: 1
 * rx_enable: 1
 * rx_in: 1
 */
const unsigned int INPUTSIZE = 13;
const unsigned int INPUT_PARTS = 6;
const unsigned int INPUT_SPEC[] = {1,9,10,11,12,13};
```

Figure 2: `inputvector` specification for uart

coverage point resides in.

## 2.3 Genetic Algorithms

A *genetic algorithm* (GA) is a robust heuristic optimization technique that helps search for non-deterministic solutions in vast search spaces, often converging in short amounts of time [4]. It mimics the natural process of evolution with the basic principle of *"Survival of the Fittest"*.

A standard GA begins with a population of random individuals or chromosomes. In each generation, individuals are selected with a higher disposition towards their health or fitness. These are then mated and combined to form offspring, thereby inheriting their characteristics. There is also a chance of mutation when an individual is born, allowing for a chance of introducing new traits. The population is kept constant and the algorithm is allowed to run for a few generation. In each generation, the average fitness of the population increases, and the end, the individual with the maximum fitness is selected as the solution. The genetic operators for `selection`, `crossover` and `mutation` are probabilistic and need to be set through experimentation. The viability of the solution greatly depends on the evaluation of each individual, i.e. the *fitness function*. A function that can distinguish between good and bad solutions with reasonable margin helps converge onto optimal solutions sooner.

## 3 Automated Statement Coverage

The goal for automated Statement Coverage is to engage each coverage point in the model. To this effort, techniques *rand1*, *rand2* and *saga* were developed.

### 3.1 CovFG Extraction

The CovFG is extracted from the final Abstract Syntax Tree (AST) dump during the Verilator model compilation, using Perl. Once the tree is obtained, for each node, every path is enumerated and stored aside. This collection of paths is called, `CFGPaths` and is an input to methods *rand2* and *saga*. Parsing the AST dump to generate the tree is done in `O(N)` time in a single pass, where the AST key indicates the level of each node, with respect to previous nodes. The path enumeration is done with a simple exhaustive Depth-First-Search for the terminal node.

### 3.2 Method-I: rand1

Random test generation is a common method used for initial software coverage testing. The algorithm, *rand1* does not rely on any input and requires no evaluation metrics.

First, any coverage point that hasn't been covered is set as a target. The algorithm begins to build a test with random inputvectors until the target coverage point is accessed. The test is initially empty, and after each addition of the random inputvector, the test is evaluated on the model. The search for that target is terminated if the test grows beyond a certain length. If indeed, the target was met, Then, the every other coverage point that was also inadvertently accessed by this test will also be marked as done. The algorithm continues to build tests for the next un-accessed coverage points.

## 3.3 Method-II: rand2

*rand2* forms the bridging algorithm between *rand1* and *saga*. It only procedurally generates random tests per coverage point, but also evaluates each step that needs to be taken.

Same, as *rand1*, the algorithm begins with an empty test, and the first un-accessed coverage point is termed as the target. Now a batch of random inputvectors are created, and are treated as prospective growths for the test. Each of these will be appended to a copy of the test on hand, and evaluated on the model. The inputvector with the highest *fitness* is selected as the candidate to the grow the test. Just as with *rand1*, the test is terminated if it does not meet the target within a certain length. Tests which succeed are recorded to the testbench and the search reiterates for other targets (un-covered coverage points).

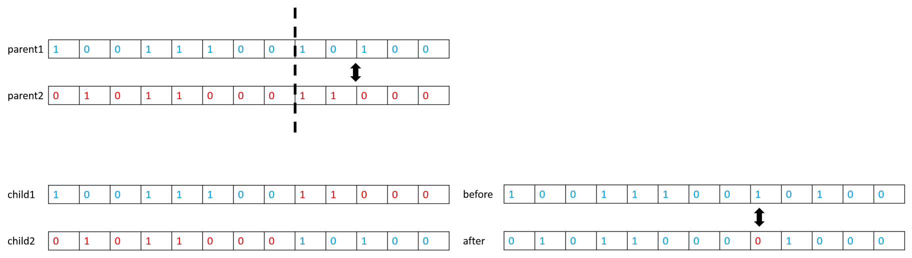
The *fitness* of test denoted as the best commonality score it shares with the paths for the target. We compare the `coveragestate` upon application of this test, against the enumerated paths for that target from the CovFG. For example, say the target was node-5, and one of its path is {`13-11-10-9-8-7-5`}. Suppose the test manages to engage points, 13, 11, 8 and 7, then the commonality score against this path is going to be 4, because the test manages to access 4 nodes which are in common with this path. Similar evaluation is committed for each path pertaining that target node, and the best score is assigned as its fitness. The same function to evaluate fitness for a test, is used in *saga* for fitness evaluation of individuals in the genetic algorithm. This indicates how close this particular test was to the accessing the target. It is presumed that continuous application of inputvectors that have high commonality score will converge upon the target sooner.

## 3.4 Method-III: saga

*saga* is an acronym for **S**tatement cover**A**ge **G**enetic **A**lgorithm. It builds upon *rand2* where it replaces the random test growth with a heuristic one.

At each growth step for a test, a GA is launched to find the most suitable inputvector to append to the test for the target. For this GA, the inputvector array is treated as an individual. A population of random individuals is initialized. A copy of the test is made, and each individual is appended to it's own copy. This is then evaluated on the model to obtain the *fitness* of each individual. To produce the next generation of individuals, *roulette* selection is used. This selection method gives us a higher uniform chance to select individuals with better fitness more often. Using the two selected parent individuals, two offspring are produced using *1-point crossover*. This technique first, pick a random point along the length of the chromosome, at which the genetic information of the parent individuals is exchanged (Fig.3). Mutation may now be performed on the offspring before adding it to the pool for the next generation. The mutation operation swaps a single bit in the chromosome. The process of selection, crossover and mutation is continued until the population size for the next generation is obtained. The probabilities for crossover ($P_c$) and mutation ($P_m$) are set to the default values often found in literature, 0.75 and 0.05 respectively.

At each generation (until a maximum predefined limit), the best individual is checked for convergence. If the target is met, then the GA is terminated and the best individual is appended to the test. Auxiliary targets (other targets met by this test) are also marked as done, at this stage. Finally, the test is recorded to the testbench, before beginning anew the growth of a test for remaining targets.



(a) 1-point Crossover (b) Mutation

Figure 3: Genetic Operations on `inputvector` for a test for uart

## 4 Testbench Evaluation using Mutation Testing

The resultant testbench from each of the previously described Automated Statement Coverage testbench generation techniques is combined to form a single testbench. This final testbench will then be validated using Mutation Testing to observe the viability of the end-product of an automated testbench generation scenario. Each test of the statement coverage goal, stops upon encountering the target coverage point. Therefore, these tests are relatively short and were created without any functional analysis in mind. Intuitively, it can be understood that the resultant testbench may not be suited for functional testing. Nonetheless, this posses an interesting starting point for automated RTL verification.

Mutation testing involves the introduction of small changes (*mutants*) in the DUT. In the case for RTL verification, we introduce four classes of mutations directly into the Verilog model. The individual mutations are stored in *diff* format for application and reference. An example mutation is shown in Fig. 4.

Four classes of traditional mutation:

**Boolean Negation Mutaion** : negation operator is removed
**Boolean Logical Mutation** : `&&` and $||$ are swapped
**Boolean Relational Mutation** : $\{$`==, !=, >, <`$\}$ are replaced with every other possibility in this set, resulting in a set of mutants
**Boolean Arithmetic Mutation** : $+$ and $-$ arithmetic operators are swapped

```
79c79
<     if (!rx_busy && !rx_d2) begin
---
>     if (rx_busy && !rx_d2) begin
```

Figure 4: Boolean Negation Mutant on line 79 in uart.v

Upon compilation using Verilator, a mutated C++ model is obtained. The final testbench is executed upon this model, and the circuit output is observed. If it differs from a clean run, then the mutant is marked as *killed*. The overall mutation testing score is defined as:

$$MutationTestingScore = \frac{mutants\_killed}{total\_mutants}$$

## 5 Results

The three techniques for Automated Statement Coverage are used to generate testbenches for standard circuits from the ITC'99 benchmark (`ckt`). Each testbench is compared for number of tests (`tcnt`), test length (`tmin/tmax/tavg`), overall inputvector count (`ivc`), and coverage (`cov` - normalized) for the number of coverage points in the circuit (`ncov`). Generation time (`time`, in seconds) is also presented to show the complexity of search involvement that each algorithm takes. Note that for increasing dimensions of the test, the longer each evaluation for each new inputvector takes. The results are displayed in Table 1, 2 and 3. The following constraints are applied across the board:

`MAXTESTLENGTH` : *rand1, rand2*: 512; *saga*: 256
`MAXPOPULATION` : *rand2*: 64; *saga*: 100(default), 64(b12)
`MAXGENERATION` : *saga*: 16

It can be noted that the coverage for each method is almost the same. While the test generation time increases from *rand1* being the fastest, to *saga* being the slowest

| ckt | rand1 | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | ncov | time | cov | tcnt | tmin | tmax | tavg | ivc |
| uart | 24 | 0.997 | 0.75 | 4 | 1 | 779 | 205.5 | 822 |
| b07 | 20 | 0.146 | 0.90 | 11 | 1 | 171 | 32.45 | 357 |
| b10 | 32 | 0.332 | 0.96875 | 8 | 1 | 179 | 56.5 | 452 |
| b11 | 33 | 0.248 | 0.909091 | 7 | 1 | 197 | 70.43 | 493 |
| b12 | 105 | 15.280 | 0.285714 | 4 | 1 | 329 | 127 | 508 |
| b14 | 211 | 2.150 | 0.900474 | 16 | 1 | 503 | 148.38 | 2374 |

Table 1: Automated Statement Coverage : `rand1`

| ckt | rand2 | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | ncov | time | cov | tcnt | tmin | tmax | tavg | ivc |
| uart | 24 | 62.57 | 0.833333 | 5 | 1 | 927 | 352.6 | 1763 |
| b07 | 20 | 1.199 | 0.90 | 12 | 1 | 87 | 22.83 | 274 |
| b10 | 32 | 4.139 | 0.96875 | 10 | 1 | 387 | 101 | 1010 |
| b11 | 33 | 5.432 | 0.878788 | 5 | 1 | 485 | 99.80 | 499 |
| b12 | 105 | 991.74 | 0.285714 | 4 | 1 | 585 | 187 | 748 |
| b14 | 211 | 97.111 | 0.890995 | 19 | 1 | 431 | 109.63 | 2083 |

Table 2: Automated Statement Coverage : `rand2`

| ckt | rand2 | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | ncov | time | cov | tcnt | tmin | tmax | tavg | ivc |
| uart | 24 | 79.125 | 0.791667 | 4 | 1 | 143 | 60 | 240 |
| b07 | 20 | 9.198 | 0.90 | 12 | 1 | 85 | 22.5 | 270 |
| b10 | 32 | 36.212 | 0.96875 | 8 | 1 | 183 | 48.25 | 386 |
| b11 | 33 | 36.485 | 0.878788 | 7 | 1 | 139 | 36.71 | 257 |
| b12 | 105 | 356.515 | 0.276190 | 3 | 1 | 195 | 66.33 | 199 |
| b14 | 211 | 837.518 | 0.857820 | 13 | 1 | 247 | 95.46 | 1241 |

Table 3: Automated Statement Coverage : `saga`

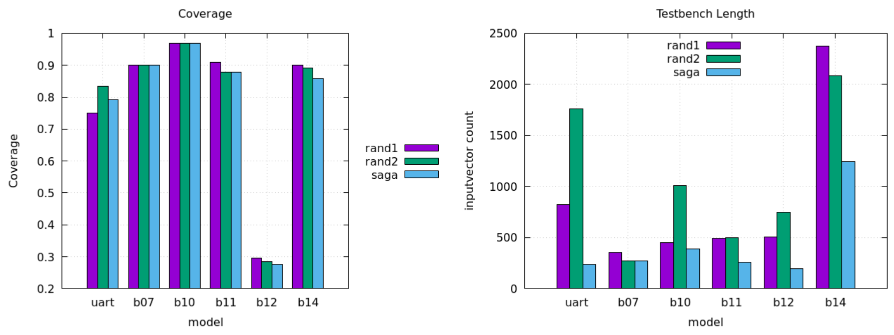


Figure 5: Testbench metrics from each Algorithm

| ckt | total_mutants | mutants_killed | Mutation Score |
|---|---|---|---|
| uart | 45 | 19 | 0.422 |
| b07 | 27 | 24 | 0.888 |
| b10 | 52 | 41 | 0.788 |
| b11 | 38 | 34 | 0.895 |
| b12 | 136 | 18 | 0.132 |
| b14 | 196 | 173 | 0.883 |

Table 4: Mutation Testing

(as expected, as it is the most computationally complex). *rand2* has the worst test length, and *saga* has the smallest testbench per circuit (Fig.5). It shows that the GA is able to efficiently search for the next best inputvector to append to the test, thereby requiring fewer inputvectors per coverage point. Circuit b12 has a complex decision tree, and therefore, none of the technique are able to attain satisfactory coverage for it. Considering that the techniques only explore the input search space, without any knowledge of predicate weight or data flow, the algorithms are not scalable for complex designs.

Mutation Testing is performed on the coalesced testbench (combination of all tests from the three testbenches) to show the viability of automated Statement Coverage. The result is displayed in Table. 4. A correlation is seen between very high statement coverage and high mutation score.

## 6 Conclusion

Automated RTL verification can indeed reduce resource usage (especially labour) during fast design cycles. Moreover, considering it is performed at an earlier stage in the design flow, the bugs that are detected and fixed are worth more *bang per buck*. Though, Statement Coverage is a minimal test, it forms a good starting point and may even be used for rapid verification scenarios like incremental design build comparison.

Three techniques for Automated Statement Coverage are implemented, *rand1*, *rand2* and *saga*. These methods are able to quickly generate testbenches that cover the design. Among all, *saga* generated the smallest testbenches, showing most efficiency for coverage in the least amount of applied inputvectors. Finally, The effectiveness of automated statement coverage is validated with functional Mutation Testing, where having high Statement coverage shows a good chance for high mutation score. For circuits with shallow decision branches, these methods perform quite well. It is revealed that these methods which search the input space are non-optimal for complex designs with deep branches like b12, and result in poor coverage. Future work would require harnessing guidance from analysis of branch predicates and data flow, to allow the search to access these hard-to-reach branches.

## References

[1] M. Abrahams, J. Barkley, *RTL verification strategies*. Wescon'98

[2] P.P. Jain, *Cost-effective co-verification using RTL-accurate C models*, ISCAS'99

[3] Veripool, *www.veripool.org/wiki/veripool/Verilog_Simulator_Benchmarks*

[4] R. L. Haupt, S. E. Haupt, *Practical Genetic Algorithms*, Wiley, 2004