Optimizing Random Test Constraints Using Machine Learning Algorithms

Stan Sokorac
Sr. Principal Design Engineer at ARM
stan.sokorac@arm.com
512-423-4517
ARM Inc., 5707 Southwest Pkwy, Building 1 Suite 100, Austin, TX 78735

Abstract- **A staple of modern verification is constrained random simulation, which involves generation of random transaction streams controlled through a set of adjustable constraints. One of the major challenges of verification is finding the right combinations of constraints to produce the most stressful tests with the widest variety of random stimulus. With typical nightly and weekly regressions generating billions of simulation cycles in tens of thousands of tests, it is impossible for a human to process all available data. Verification engineers therefore use various aggregation and approximation methodologies, such as code and functional coverage, to gain insight into regression results. However, the fields of machine learning and data mining excel at exactly these kinds of problems by finding patterns in this vast repository of data and extrapolating insights to guide us in the best direction. This paper presents a methodology in which coverage results and machine learning algorithms are used to generate tests most likely to find new bugs in a design.**

I. Introduction

Modern designs are extremely complex. It is impossible to manually come up with all the stimulus necessary to completely validate the design. To solve this problem, verification engineers rely heavily on constrained random simulations. By writing a set of constraints and generating random streams of transactions, one could hit both common and uncommon design corners.

However, random testing is also very inefficient and, therefore, expensive. The common experience involves running millions or even billions of random simulation cycles and finding corner-case bugs that should have been hit in a fraction of those cycles. In my analysis, I have seen that the most frequently hit coverage points tend to be hit orders of magnitude more often than the rare ones. Figure 1 shows data from one recent functional coverage run, illustrating that 10% of coverage points collect over 99% of total coverage bin hits. The other 90% are hit very rarely, and bugs that are hidden behind those will require billions of simulation cycles to reach.

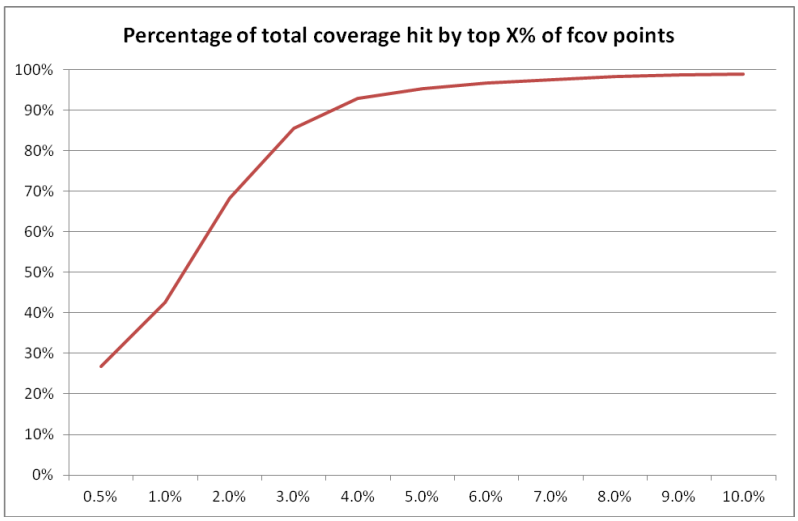

Figure 1. 10% of coverage points hit most often account for most of the coverage hits in a large regression. Data is collected by accumulating coverage hit totals starting with the most common point.

In this paper, two ideas are presented to make random simulation more effective at finding these hard-to-find bugs. The first section offers a new type of coverage designed to provide feedback on the kind of events that are likely to expose bugs. The second section describes an application of machine learning algorithms that uses coverage feedback to select and tune random tests, with the goal of increasing the coverage of rare states.

II. TOGGLE-PAIR COVERAGE

Non-trivial bugs require a combination of events and state changes to occur in close proximity in time. Most bugs are not particularly deep and do not require thousands of cycles to set up, but generally take a couple of events lining up that do not often happen together.

Verification engineers bias their stimulus toward areas that are likely to cause bugs, and write functional coverage to ensure the stimulus does what they expect it to do. This is a great way to use their experience and knowledge to find the most bugs as quickly as possible. However, once both code and functional coverage are 100% hit, it is hard to know where to go next. Code coverage is too easy to hit, and functional coverage is subjective, and by definition, already covers everything that is considered important.

Therefore, there is a need for a new kind of coverage that is as objective as code coverage, but much harder to complete (if not impossible), while simultaneously guiding us toward missed bugs.

"Design state space" is the ultimate coverage metric, often represented as all possible combinations of values in all flops in the design. By hitting every one of those points, it can be safely said that the design has been made to do everything that it can possibly do. However, that is far from a practical coverage measure – at 2^n, where n is the number of flops, it very quickly becomes impossible to cover even a tiny portion of it.

On the other hand, flops toggling in the design do represent state changes and should still be considered in some way. Straightforward flop toggle coverage is one of the common verification metrics. However, it is much too simple, like code coverage. A small number of well-crafted directed tests are generally able to close flop toggle coverage with very little effort, while still missing a number of more complicated bugs.

If flops toggling are considered as design *events*, going back to the concept that bugs occur when events line up together in time, then having two flops toggle in proximity to each other (in time) would represent a couple of events lining up together. By monitoring all flop toggles and extracting all toggles pairs, a new kind of coverage has been created, which can be referred to as 'toggle pair coverage'.

There are several good properties of toggle pair coverage. The total number of bins is n^2, where n is a number of flops in the design. This is, on one hand, much larger than code coverage, but still quite manageable. Figure 2 shows how the number of bins hits a sweet spot between plain toggle coverage and the full state space. It is large enough that it is highly unlikely that all bins can be hit and there is likely always room for further improvement.

The toggle pair coverage is also objective, removing the engineer's bias towards parts of the design or types of traffic that has already been covered extensively.

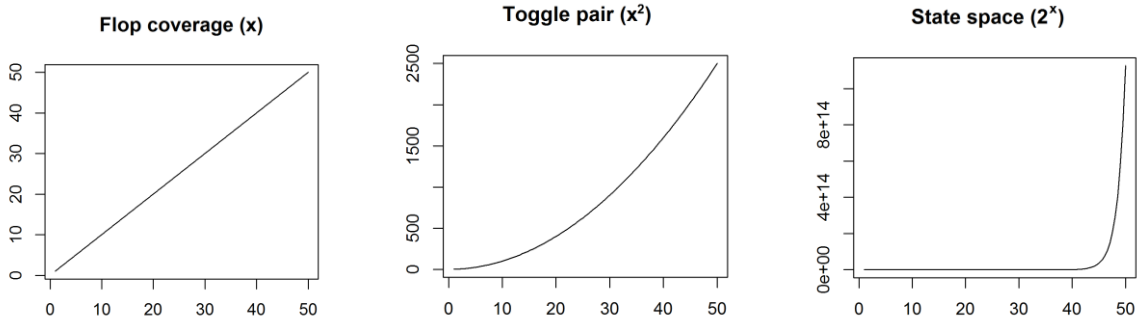

Figure 2. Toggle pair coverage provide a large, but manageable set of bins, compared to straight flop coverage and full state space.

A. Interpreting results

There are a few ways to interpret coverage results. First, one can count how many different toggle pair coverage bins[1] are hit and define that as the coverage *breadth.* The more unique bins hit by a particular test, the wider the coverage of the test is, which intuitively seems like a good thing. While that provides a good indication of how much of the design is covered, it does not help with the overall goal of figuring out which tests to run more and which to run less. In order to do that, a deeper analysis of the data is needed.

Looking at the total volume of hits is also important. Hitting many bins a few times each vs. hitting a few bins many times indicates a very different test profile. Let us define the total number of toggle pair bins hit, including multiple hits of the same bin, as the test activity *volume*.

Naturally, the most stressful tests will produce high toggle pair coverage in both volume and breadth – they do a lot of different things, and they do them frequently. However, many of these tests will repeatedly pound on the same toggle pair bins. This is an inefficient use or resources, and an important goal is to reduce or even eliminate such waste.

To avoid rewarding tests that do common things, let us define another metric. By examining how often each of the toggle pair bins is hit and by how many tests, the *rarity* of bins can be defined to be inversely proportional to that quantity. Bins that are often hit by many of the tests would therefore have a low rarity value, while the ones that are hit very infrequently would have a high rarity value. This metric is powerful, as it allows one to filter the toggle pair coverage results and focus the analysis of breadth and volume of tests to rare bins only.

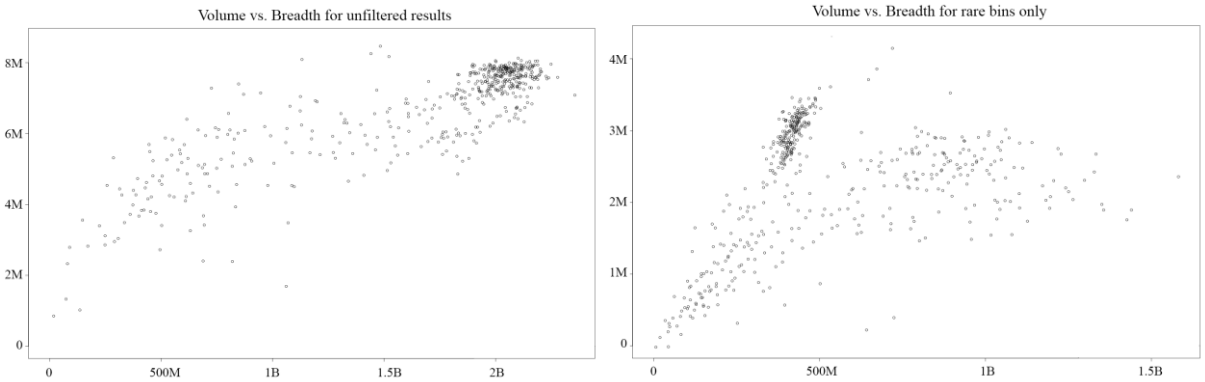

Figure 3. Volume (X-axis) vs. Breadth (Y-axis) plot of 1000 tests. Plot on the left shows unfiltered values of volume (total hits) and breadth (unique hits), while the plot on the right shows values after they have been filtered based on their rarity.

Figure 3 shows how the tests' locations on the breadth vs. volume plot change once the rarity filter is applied. While the initial chart showed a large grouping of stressful tests in the top right corner of the chart, many of them have migrated to the left once the easily hit bins have been removed. The new chart shows that only a small percentage of tests exercise rare bins effectively, and, ideally, these are the tests to be run more of because they provide more unique events with less simulation cycles.

The final step is to collapse the filtered breadth and volume metrics into one number, a *score* for each test. There are many common functions that could work here. F1 score [1] is commonly used in machine learning to reward results that score high in both dimensions and penalize those that are bad in either one. However, F1 would penalize non-stressful tests that hit many of the rare bins (high breadth, low volume) too strongly, and there is still significant value in selecting those tests. The following function was found to produce the best results, which is essentially the distance of the test point from origin, with an option to scale up the breadth axis, and an option to stretch out the values between good and bad tests with a power factor:

[1] A toggle pair coverage "bin" is one pair of flops in the design

$$Score = (Volume^2 + Rare\_Factor * Breadth^2)^{Power\_Factor}$$

The higher the *Rare Factor*, the more the score will favor tests that hit many different rare bins. The higher the *Power Factor*, the more spread out the scores will be – valuing high-scoring tests more than low-scoring ones. This can later be used as *learning rate* parameter in selection algorithms.

Table I shows an example of how the test scores can be modified using these optional parameters. The two factors give us the ability to tune the scores quite significantly, as the table shows. Test #1 is very stressfully, but narrowly focused, while test #2 hits a wide range of rare bins, with a much lower overall stress. While test #1 nominally scores higher, one could make test #2 much more valuable by bumping up the *Rare Factor*. Similarly, tests 13 and below are overall poor tests, but the nominal score has them scoring only slightly below the top tests. By setting the *Power Factor* to 4, the scores for those tests become significantly smaller than the top tests.

Varying these settings allows for different interpretation of results, and many combination yield interesting results. There is no one best way to determine these parameters as they are heavily dependent on the particulars of the testbench and the design under test. Experimentation, or even running machine learning algorithms on these meta-parameters, is needed to determine the optimal values for a particular setup.

This is now a way to find which tests exercise rare combinations of events in the design most commonly.

TABLE I
TEST SCORES FOR VARIOUS RARE AND POWER FACTOR VALUES

					Scores				
				Rare Factor	1	1	1	2	2
				Power Factor	0.5	4	10	1	5
Test	**Volume**	**Breadth**	**norm_vol**	**norm_brd**					
1	52866820	1635913	**1.00**	0.58	**1.16**	**3.22**	**18.64**	2.36	73.09
2	24083754	2806373	0.46	**1.00**	1.10	2.13	6.59	**4.21**	**1318.67**
3	44567961	1842487	0.84	0.66	1.07	1.70	3.76	2.43	85.58
4	43412138	1603645	0.82	0.57	1.00	1.00	1.01	1.98	30.46
5	34938147	1877157	0.66	0.67	0.94	0.61	0.29	2.23	54.70
6	37187583	1714573	0.70	0.61	0.93	0.57	0.24	1.99	31.04
7	31292009	1962939	0.59	0.70	0.92	0.50	0.17	2.31	65.39
8	14982993	2278261	0.28	0.81	0.86	0.30	0.05	2.72	147.93
9	26646991	1842955	0.50	0.66	0.83	0.22	0.02	1.98	30.36
10	34902126	1338068	0.66	0.48	0.81	0.19	0.02	1.35	4.40
11	29705209	1581578	0.56	0.56	0.80	0.16	0.01	1.59	10.04
12	12999333	1952907	0.25	0.70	0.74	0.09	0.00	2.00	31.80
13	15764516	1482763	0.30	0.53	0.61	0.02	0.00	1.21	2.55
14	8775679	1058515	0.17	0.38	0.41	0.00	0.00	0.60	0.08
15	13248153	466477	0.25	0.17	0.30	0.00	0.00	0.17	0.00

III. GENETIC ALGORITHM

My goal was to use the scoring methodology in part II to devise an algorithm that would generate a list of tests that would stress the rarely reached parts of the design as often as possible, thereby increasing the chance of finding new bugs. For this, machine learning algorithms were used, which excel at analyzing large amounts of data and recognizing the patterns.

In machine learning, it is often the case that the data and its interpretation make the difference between success and failure, and algorithm selection only gives incremental benefits. Therefore, while there are many more advanced and effective learning algorithms available, this paper uses one of the simplest, which is easiest to get up and running. The algorithm is known as a Genetic Algorithm [2].

In this particular application, the algorithm works in the following way:

1. The initial *population* is a random selection of tests, typically the normal list of tests used in nightly or weekly regressions.
2. For each test, toggle pair coverage is collected, and the score is calculated.
3. The next *generation* is determined through a mix of as-is tests from current generation, *mutated* tests, and brand new randomly generated tests. A common distribution used that has been found to work in most situations is to carry over 10% of the tests, generate 15% of new tests, and mutate the rest (75%). In testbenches with a particularly high test state space, a larger proportion of new tests could work better, giving more tests a chance to be picked.
4. Repeat steps 2 and 3

The mutation of tests is a key part of this algorithm. The test must be modified in such a way to produce different, yet generally similar results. Too similar, and all generations will look very much the same, and no progress towards a better selection of tests will be made. Too different, and results become random, with scoring having little impact on future tests.

In our testbench, each test is controlled through a set of about 150 command line options. By running a test with the same, or almost the same, set of command line options, and varying the random seed, tests that fit the above criteria can be generated.

A. Iterations and Analysis

Accurate algorithm progress evaluation is critically important. In each iteration, a number of charts and metrics are generated to provide information about the current status. Table II shows some of the information gathered from iteration to iteration.

TABLE II
CHARS AND METRICS GENERATED IN EACH ITERATION

Information	Description
Plot of breadth vs. volume, for all toggle pair coverage bins	Shows the overall distribution of tests, and gives a hint at the variety of tests (Figure 3).
Plot of breadth vs. volume, for rare toggle pair bins only	Comparing from iteration to iteration, shows whether common tests are being filtered out (Figure 3).
Number of new bins added and bins lost over iteration	If the algorithm is working well, common bins will still be hit even when selected against them, while new rare bins will be added as 'rare' tests are selected. This plot shows us how well that is working (Figure 5).
Number of fails per iteration	Any large spikes in failed tests could indicate that an iteration has hit some sensitive spot in the design, or that the test generator is producing invalid tests. This metric allows us to flag those iterations for review (Figure 4).
Total number of bins covered	The cumulative count of toggle pair bins covered by all iterations should grow if the algorithm is still optimizing the test selection. Once this curve flattens out, it is probably a good time to stop iterating (Figure 4).
Number of total bins considered rare	Another good indicator that the algorithm is working, and when the iterations should stop. By optimizing for test hitting rare bins, some of the rare bins will no longer be considered rare as many of the selected tests will be hitting them often. If this curve is flat, or has stopped decreasing, the iterations should likely end (Figure 6).

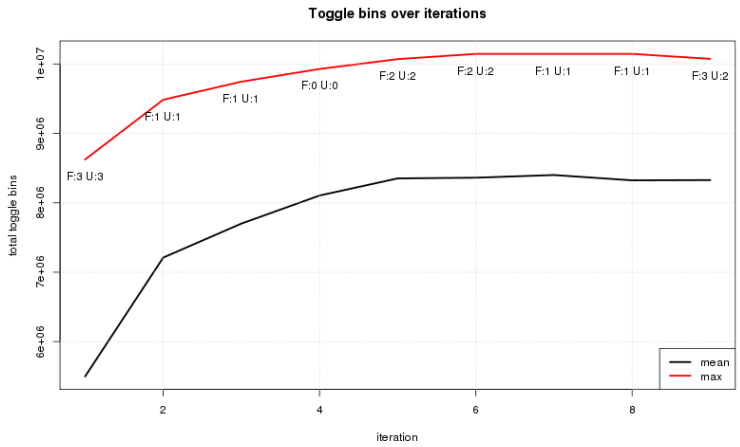

Figure 4. The average (in black) number of bins hit by a test per iteration, as well as the maximum (in red) number of bins hit by a test. 'F' and 'U' indicate fails, and unique fails.

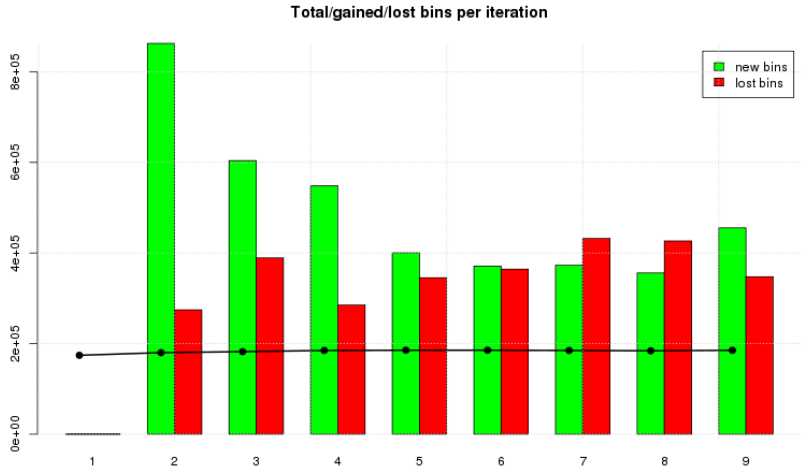

Figure 5. Number of bins hit in iteration that have not been hit previously (*new bins*), and the number of bins that have been previously hit but are no longer covered in current iteration.

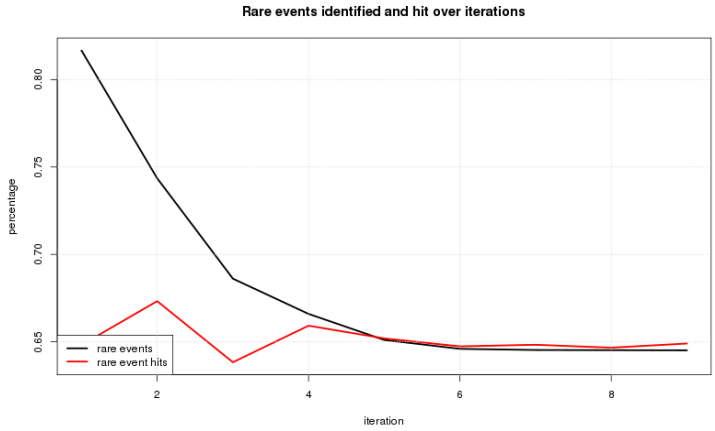

Figure 6. Percentage of bins considered *rare* in each iteration. As test selection improves and stimulus is better distributed, the number of overall rare events goes down.

IV. Clustering

In many cases, the above methodology produced excellent results by itself. However, on some of the larger designs, a problem frequently showed itself. A large design is likely to have several areas of toggle pair coverage where rare bins exist. The ideal goal is to find tests that cover all of those areas of the design. In practice, though, there will be a distinct set of tests that hit each of the desired areas, and some areas will be easier to hit frequently than others. In those situations, the tests that are hitting the "easier to hit frequently" areas will score higher than tests hitting hard-to-hit area, and will overwhelm them in a few iterations. One approach is to group tests by the coverage areas that they primarily hit, and run individual optimization and selection within each group.

For this purpose, another kind of machine algorithm was needed – a clustering algorithm. This is a type of *unsupervised learning* algorithm, in which the operator does not tell the machine what the desired answer is, but allows it to look through the data and find a pattern on its own. The most common, and also very effective, algorithm in this group is called *k-means clustering* [3] algorithm. Given a set of data points, and the number of clusters one would like to generate, it groups the similar data together.

The data points needed by the algorithm are simply the toggle pair coverage data, normalized so that tests that hit the same bins with any frequency and are considered similar. By comparing differences between normalized toggle-pair data, k-means clustering locates test groups exercising the same events combinations.

The data can be fed into k-means directly, but it is again important to be able to visualize the clustering to see if it makes sense. To visualize the data, it must converted into 2D or 3D, for which an algorithm called *multi-dimensional scaling (MDS)* [4] is used. MDS will take a matrix of differences between points and plot them in 2D/3D in such a way that similar points are near each other, and different points are far away. Figure 7 shows one such plot.

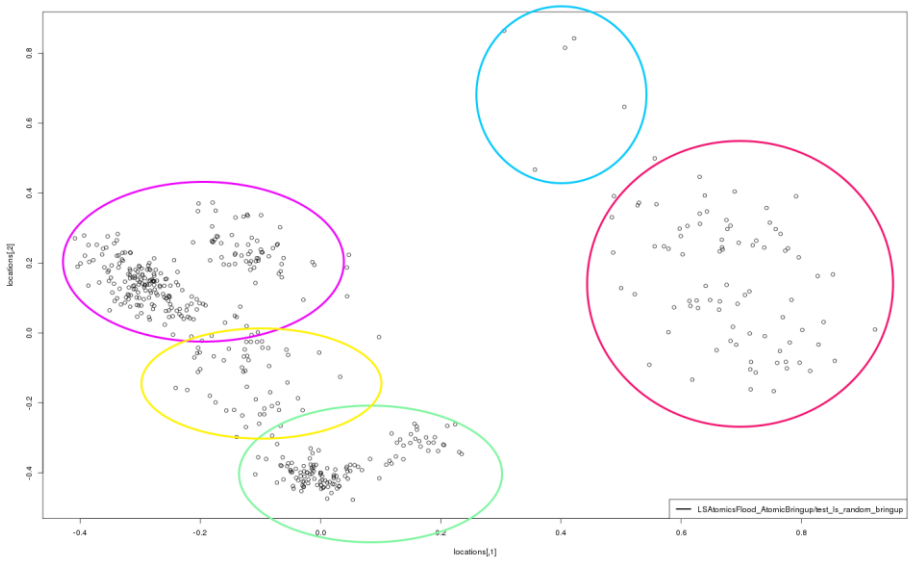

Figure 7. Plot of toggle-pair coverage similarity between tests. Hand-drawn clustering is one possible way to group similar tests.

Once the grouping is established, each cluster can be treated as a completely separate regression. The test scores are only compared within the same cluster, and distribution of new/carry-over/mutated test selection is carried over for each cluster.

V. Results

Early results are quite promising. Optimized regressions have consistently failed more often than non-optimized ones, and have uncovered a larger number of unique failures. Tables III and IV show two examples, one from an earlier milestone where it was still easy to find new bugs, and one from a more recent regression where bugs were starting to dry up.

TABLE III
FAIL RATES AND UNIQUE FAIL SIGNATURES IN AN EARLY-PROJECT EXPERIMENT

Regression	Test Count	Fail Count	Pass Rate	Cycles	Unique Signatures
Regular weekly run	30000	24	99.92	173.6 Million	4
6 iterations of 500 tests	2749	41	98.5	15.4 Million	5
Large run using 6th iteration test selection	30000	469	98.43	166.1 Million	8

TABLE IV
FAIL RATES AND UNIQUE FAIL SIGNATURES IN MID-PROJECT EXPERIMENT

Regression	Test Count	Fail Count	Pass Rate	Cycles	Unique Signatures
Large weekly run	10000	5	99.95	56.9 Million	5
8 iterations of 1000 tests	7327	16	99.78	41.7 Million	6

In table III, after the 6th iteration was reached, the population of 500 tests was used to generate a full 30,000 test regression through the same *mutation* methodology followed from iteration to iteration – i.e., each test's controls were slightly modified and new random seeds were generated. The resulting regression showed an order of magnitude higher number of failed tests, and double the number of unique fail signatures.

These results show that this methodology can be effective at producing higher rates of failures in smaller number of simulation cycles, while simultaneously exposing new bugs, through new unique fail signatures.

VI. FUTURE WORK

In a non-academic setting, it is often desirable to be able to show some results as quickly as possible. While this paper presents a complete solution to a problem, there are many other avenues to explore to further improve the results of this process.

For example, while toggle-pair coverage yields very good results in our experiments, it would be very interesting to run the same process through more traditional coverage data, like code and functional coverage.

In particular, though, there are many more machine learning algorithms that could be applied to this problem to further enhance and improve efficiency of test selection.

A. Neural Networks

The current learning algorithm relies on running each of the selected tests to learn about its properties. It has no way of inferring what a test that has never been run will do. However, other machine learning algorithms exist that could learn the effects of each of the test controls, and predict what a test will do without running it. Neural networks are particular powerful in these kinds of situations.

B. Anomaly Detection

Anomaly detection algorithms are used to detect credit card fraud, for example [5], as they search for unusual payment patterns. That same class of algorithms could be used to analyze coverage information to detect tests that are significantly different from others, flagging them for further analysis. These tests could be reaching novel parts of the design state space, or perhaps doing nothing interesting at all – in both cases, this would be important to know.

C. Meta-learning

Our genetic algorithm has many parameters – apart from the rare and power factors described in the paper, each run has several others. How many tests to run, how many iterations, what is considered a rare point, how many test should mutate and how many should stay the same? Through trial and error, a good set of parameters was reached, but they could be optimized further by a more formal analysis. Another genetic algorithm could, in fact, be used to iterate through many different options and find the best combinations.

REFERENCES

[1] "F1 score", Wikipedia, https://en.wikipedia.org/wiki/F1_score

[2] "Generic Algorithm", Wikipedia, https://en.wikipedia.org/wiki/Genetic_algorithm

[3] "k-means Clustering", Wikipedia, https://en.wikipedia.org/wiki/K-means_clustering

[4] "Multidimensional Scaling", Wikipedia, https://en.wikipedia.org/wiki/Multidimensional_scaling

[5] K. Chaudhary, et al, "A review of Fraud Detection Techniques: Credit Card" *International Journal of Computer Applications, Volume 45 – No.1, May 2012*