Automated Seed Selection Algorithm for an Arbitrary Test Suite

David Crutchfield
Sr Principal CAD Engr
Cypress Semiconductor
859 977 7555
Lexington, KY 40507
David.Crutchfield@cypress.com

Brian Craw
Sr Staff CAD Engr
Cypress Semiconductor
859 977 7572
Lexington, KY 40507
Brian.Craw@cypress.com

Jim Sharpe
Principal CAD Engr
Cypress Semiconductor
859 977 7554
Lexington, KY 40507
Jim.Sharpe@cypress.com

Brandon Skaggs
Sr Staff CAD Engr
Cypress Semiconductor
859 977 7598
Lexington, KY 40507
Brandon.Skaggs@cypress.com

***Abstract*- Modern digital simulators provide mechanisms for identifying which tests within a regression provide the best functional coverage--and which tests consume resources without adding coverage; however, the establishment of an optimal set of seeds is a manual process. This step is often ignored during the test implementation phase--where changes in the design and/or test bench can invalidate a set of seeds quickly after they are identified. This leads to wasted resources and longer regression times. This paper presents an algorithm for the automatic maintenance of optimal seed sets by iteratively increasing or decreasing the number of seeds used per test based on the incremental coverage collected at each stage.**

I. Introduction

Constrained randomization of values for test bench variables is a common practice of modern verification through SystemVerilog. The randomization solver of a simulator is responsible for setting such values and is controlled through a seed argument. By changing the input seed for a given test, a user can alter the randomization solver output, and thus, the values of test bench variables that are designed to be randomized based on type.

Choosing these seeds can be an art within itself. It could be that certain seeds do not provide appreciable differences in variable values, thereby not providing any additional functional coverage or code coverage from one seed to another. Choosing blindly could lead to redundant tests, wasting hardware and license resources unnecessarily. Tool vendors have provided help in this area with tools that rank tests according to the test bench or design coverage they have provided for a given regression. The ranking process not only identifies the tests with the most coverage, it also identifies tests that don't contribute to coverage at all. Knowing which tests to eliminate in a regression is essential in improving efficiency of a regression.

Unfortunately, having a ranking tool is only part of the solution. Assume that a verification engineer has been building coverage by intelligently selecting seeds through test ranking over several regressions. If the design being tested or test bench happens to change due to a feature enhancement or bug fix, the selected seeds are no longer valid and the process of building coverage must start again. This can be a tedious process. What has been observed within Cypress is the total disregard for ranking tools, as the results do not carry when a change is made. Instead, larger regressions are launched where an engineer will ask for many random seeds for each test. This causes an unwanted multiplication factor on resources, regardless of the actual coverage achieved. Furthermore, engineers tend to hard code seeds and always select ones they feel provide proper stimulus without checking to see if they do. Automation is needed in properly building a test and seed list combination without user interaction.

Within Cypress we have developed a tool for verification management. The Verification Management System (VMS) is responsible for building compilation and simulation tasks based on a generated or user provided test list. VMS includes automated seed generation, as well as coverage ranking of regressions. Given that VMS can control seed values, ranking of regressions, generation of a test list, and launching a regression based on the generated test list, it is ideal for automating the process of building a regression that provides optimal coverage based on the initial test list provided.

II. Definition of Terms

Constraints	Refers to descriptions of the relationships between simulation parameters; used to select appropriate random stimulus during Constrained Random Verification
CPU	Central Processing Unit – the main compute resource of a workstation
CRV	Constrained Random Verification – methodology for applying pseudo-random stimulus in an effort to verify design intent
DUT	Design Under Test – the portion of a circuit being compared to expected behavior
Functional Coverage	A metric for evaluating the completeness of verification, based on the concept of identifying important use-cases and tracking that all valid cases are exercised
Licenses	Refers to the limiting of the number of simultaneous executions of software, per agreement with the vendor, which is typically tied to the payment for support.
LSF	Platform Load Sharing Facility – Workload management platform, job scheduler, for distributed high-performance computing environments
Pseudo-random	Having the nature of producing numbers that satisfy one or more statistical tests for randomness but produced by a definite mathematical procedure
Questa VM	Questa Verification Management – Functionality within Questa SIM that offers a wide variety of features for managing the regression. These features are built upon the Unified Coverage Database (UCDB).
Questa VRM	Questa Verification Run Management – Mentor Graphics' tool for launching verification tasks within a regression
Seed	The starting input for pseudo-random calculations. Given that the randomization algorithm is maintained constant, the specification of a seed is all that is required to reproduce a pseudo-random sequence.
SystemVerilog (SV)	A class-based extension of Verilog in common use for digital verification
TB	Test Bench – the code that instantiates the DUT and is used to verify design intent.
UCDB	Unified Coverage Database – Mentor's format for consolidating coverage information across a regression set.
VMS	Verification Management System – Internally developed tool for gathering design and test bench file information, compilation arguments, simulation arguments and test information, launching each task, and collecting regression information and status

III. SystemVerilog Constrained Random Verification

Constrained Random Verification (CRV) is a methodology whereby pseudo-random stimulus is applied to the DUT in an effort to verify its design intent. To gauge verification completeness, SystemVerilog also provides a coverage construct. A high-level summary of CRV is that this pseudo-random stimulus is iteratively applied until 100% of the user defined coverage metrics are met. This pseudo-random stimulus is guided by constraints and the built-in constraint solver in SystemVerilog.

As with any solver, there must be a starting point. In SystemVerilog, this start value is called a seed. It can be randomly generated by a simulator, or specified by the verification engineer on a per test basis. In an unchanging environment, a given seed will generate the same pseudo-random stimulus every time the test is run. This can be

referred to as randomization stability. This randomization stability allows a verification engineer to rank their seeds in such a way as to run the minimal set of tests needed to achieve coverage goals, thus ensuring efficient usage of resources for regression runs.

However, a verification environment is typically an evolving set of code, at least early on, and this begs the question "what will cause randomization instability?" Unfortunately, there are many things that can lead to instability, and here are a few of the main culprits:

1. The first and most obvious way to change the randomization results is to change the constraints. The solver works sequentially, thus even a minor constraint change can affect the results of the next stage and so on. A good rule of thumb is to assume any change to the constraints will invalidate any prior seed ranking.
2. Use of $randomize based on events throughout the simulation cycle. The best way to ensure random stability is to randomize only once before the simulation begins. If randomization of stimulus occurs based on temporal events, then even non-constraint related changes to the test bench code can cause random instability by changing the timing and / or order of the $randomize function calls.
3. For a similar reason, it is also possible for changes in the DUT to affect the randomization stability. If the test bench uses DUT output to generate further random stimulus via the $randomize function, and the DUT changes in such a way as to alter timing or otherwise affect the signals involved in the $randomize function, downstream stimulus can change.
4. Lastly, different simulators and even different versions of the same simulator, can alter the solve order due to bug fixes, optimization differences, or algorithm updates.

For large regression suites, a change in the randomization can be extremely costly in that large numbers of seeds need to be run again to find the optimal seed set. While good test bench practices can be used to avoid items 1-3 above, it can be difficult to avoid randomization stability changes based on tool updates. In fact, it can be that an update is needed to address a bug or implement a new feature, and then the verification engineer must go through the pain of establishing a new, optimal seed list. This makes an automated approach highly desirable.

IV. Verification Management System

Years ago, Cypress developed VMS (Verification Management System) to manage logic verification regressions across the entire company. Along with the VMS tool, the project created a standard for design and test bench organization, specification of tool arguments, test list creation, regression status reports, and coverage information. VMS leverages Mentor Graphics' Questa VRM (Verification Run Management) tool to launch compilation and simulation jobs through a load sharing facility (LSF), and to generate and merge functional coverage information. Having a standardized verification environment is important in cases where new techniques are desired. These techniques or methodologies can be developed and propagated easily throughout the company without burdening verification engineers who are focused on design and product delivery. In the context of this paper the intent was to provide automation for narrowing down SystemVerilog randomization seeds in an efficient way, saving valuable CPU and license resources.

A. Seed Generation in VMS

VMS offers SystemVerilog seed generation to aid in SV seed development and book keeping. For an individual test within the test list, SV seed intent can be specified through three options as defined in Table 1.

TABLE 1
VMS SEED GENERATION OPTIONS

Option	Description
-num_seeds	Specifies the number of SV seeds to generate
-rand_seed	Use this argument to seed the random number generator
-sv_seed	Specifies explicitly which SV seed to use (no generation)

Using these options, a user can initially generate pseudo-random seeds through "-rand_seed" and "-num_seeds". If specific seeds have been found to provide better coverage than others they can be targeted directly through "-sv_seed". Over multiple regression runs and through coverage analysis a targeted test list can be created to provide optimal coverage.

B. Ranking Tests with Questa Simulation VM

Questa Simulation Verification Management (Questa VM) is a tool provided by Mentor Graphics that enables coverage and verification management of the Unified Coverage DataBase (UCDB). The UCDB contains all code coverage, functional coverage and assertion data from a simulation run, or multiple simulation runs through coverage merging. VMS creates unique tests based on test name (UVM_TESTNAME) and a given SV seed. Therefore, there can be multiple named tests, but replicated based on the number of SV seeds given for each within a merged UCDB. Questa VM contains a feature to rank tests within a merged UCDB based on the coverage contributed by each test. This can be used to prune out unwanted tests and focus only on the test-seed pairs that will contribute to closing coverage.

C. Shotgun Approach for Seeds

While it is understood that not all verification engineers generate test-seed pairs the same way, it has been observed that a shotgun approach is typical. Through VMS, an engineer might ask for some number of seeds per test regardless of that test's ability to actually improve coverage. For instance, assume there are 100 randomized tests in a test list. An engineer may request 10 random SV seeds for each test, bringing the regression total to 1000. This becomes the default regression size and results are merged over time from one regression to the next. After 10 regressions 10000 tests would have been executed with little thought to actual coverage contribution of these 10000 test-seed pairs. In almost all cases it can be shown through coverage ranking tools that 10-30% of the tests accumulated within a large regression contribute to coverage. Taking advantage of ranking tools between regression runs will allow for intelligent selection of seeds to dynamically shift resources to tests that are providing coverage contribution.

IV. SEED SELECTION ALGORITHM

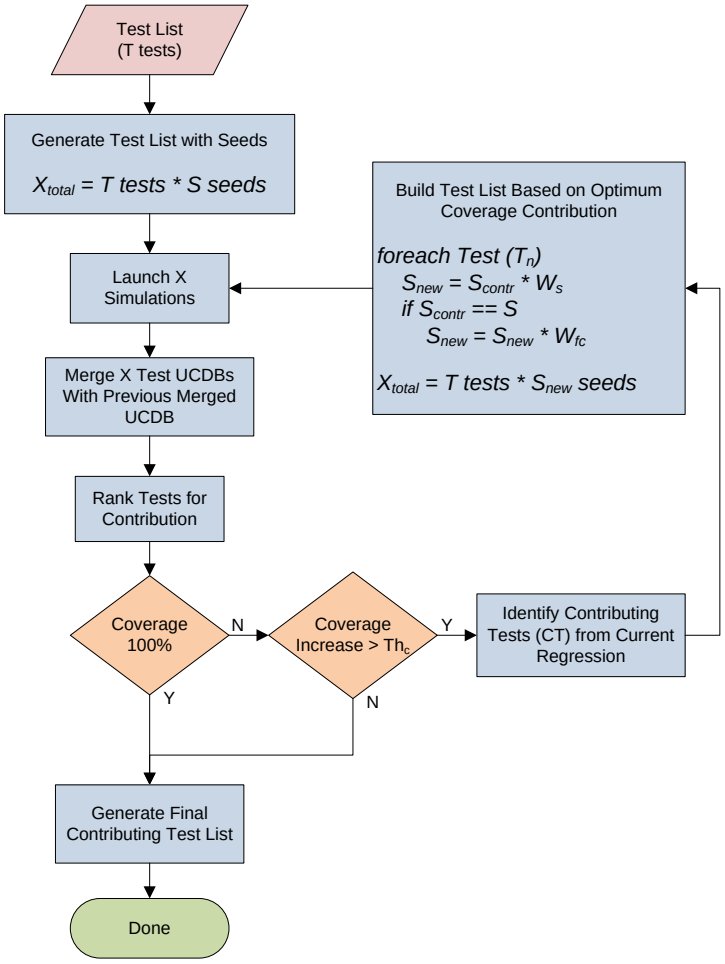

Figure 1. Seed Selection Algorithm

Figure 1 presents an algorithm to quickly eliminate non-contributing tests and highlight seeds for contributing tests in an efficient manner. The basis of this algorithm is to use the tools provided and intelligently select test and seed combinations repetitively until all choices appear to be exhausted. Each step in the algorithm will be discussed here in detail.

Test List – The test list is a required input to the algorithm and will consist of ***T*** tests. These tests will be named according to VMS specifications, where the test name is given along with any options to make the given test unique. One such argument could be the SV seed. A test may have an option specified to generate some number of SV seeds for the test name given.

Generate Test List with Seeds – Process the input Test List to identify and generate test name and seed pairs. The output of this step will be a fully populated test list where each line contains a test name and seed pair. Test names may be repeated with unique SV seeds. The generated list will be the exact number of tests to execute for the initial regression. The number of tests will be $X_{total} = T \text{ tests} * S \text{ seeds}$.

Launch X Simulations – Launch the tests determined in the previous step. There are two paths into this step. Either the initial full test list or a pared down test list identifying only new test and seed combinations will be executed.

Merge X Test UCDBs With Previous Merged UCDB – Merge test level UCDBs from the current regression into the UCDB that contains all regressions executed to this point. This merged UCDB will be used for test ranking and will contain all tests from the initial regression and subsequent incremental runs. For the initial regression it will simply be a merge of all tests executed without previous results.

Rank Tests for Contribution – Use ranking tool to rank the merged UCDB. This will create two lists of UCDBs; one for contributing tests and one for non-contributing tests.

Coverage 100% – Decide if to continue based on coverage met. If 100% coverage is met, there is no need to continue refining test and seed pairs. However, if more coverage could possibly be gained then continue.

Coverage Increase > Th_c – Decide if to continue based on incremental coverage increase. If the increase in coverage is above a predetermined threshold, $\mathbf{Th_c}$, then continue. Further investigation should halt if the threshold is not met.

Identify Contributing Tests (CT) from Current Regression – From ranking results, identify contributing tests from the current regression. Contributing tests would have at least one seed paired with them that produced an incremental coverage improvement. At this point unique tests from the initial test list may be pruned out as they have not produced incremental coverage in the current regression based on the seeds chosen.

Build Test List Based on Optimum Coverage Contribution – Create a test list from contributing tests that will reward contribution by providing seeds accordingly. The reward can be controlled through seed weightings, $\mathbf{W_s}$ and $\mathbf{W_{fc}}$. $\mathbf{W_s}$ will represent a general seed weighting for all tests. $\mathbf{W_{fc}}$ will represent an additional weighting for tests where seeds from the current run fully contributed to the incremental coverage.

Pseudo code for this step is provided in the figure and described here. Looping through each test, create new seeds in quantity based on contributing seeds from the current regression, $\mathbf{S_{contr}}$, and $\mathbf{W_s}$. The number of seeds for the next regression is represented as $\mathbf{S_{new}}$. Furthermore, if the number of seeds contributing is the same as the number of seeds initially provided, **S**, for a given test of the current regression then add more seeds based on $\mathbf{W_{fc}}$. The intent is to place more focus on the maximum contributing tests.

As an example, assume that during the current regression $\mathbf{T_1}$ had five seeds contributing incremental coverage ($\mathbf{S_{contr} = 5}$). For the regression, **10** seeds had been provided for $\mathbf{T_1}$ initially ($\mathbf{S = 10}$). If $\mathbf{W_s}$ was configured to be **2** then $\mathbf{S_{new}}$ for $\mathbf{T_1}$ would be given **10** seeds for the next regression. In this case, $\mathbf{T_1}$ would be rewarded with **10** more seeds. As another example, consider $\mathbf{T_2}$ provided **10** contributing seeds out of **10** with $\mathbf{W_s}$ and $\mathbf{W_{fc}}$ set to 2. $\mathbf{S_{new}}$ would be set to **20** seeds for a $\mathbf{W_s}$ of **2** and increased to **40** given $\mathbf{W_{fc}}$ is also **2**. $\mathbf{T_2}$ would be highly rewarded for maximum seed contribution. Each test, $\mathbf{T_n}$, would be considered in this fashion to build a new test list with **X** tests.

Generate Final Contributing Test List – Once the optimal coverage has been identified from for the initial test list given, generate a new test list based on ranking results. The new test list will only contain test and seed pairs that contribute to coverage. This will potentially be an accumulation of tests over several regressions as tests are merged in over subsequent cycles of the algorithm.

V. Results Comparison

Several iterations on this algorithm have been generated for comparison here for one type of test bench. One pass will be discussed in detail for an understanding of what is being highlighted. Figure 3 contains results from three total iterations for the first test bench. Results from a second test bench will be discussed later in this section through Table 9 and Table 10. The first test bench was chosen as it allows for quicker results while the second test

bench contains much longer lead time for results. Only one iteration for the shotgun approach and algorithm will be presented for the second test bench. Tables with all data collected for each iteration of the first test bench are contained in the appendix. With each method, regressions are executed until coverage no longer increases (i.e. $\mathbf{Th_c = 0}$).

A. Test Bench 1 Results

The initial comparison data point for each utilized the shotgun approach to creating test-seed pairs. The number of unique randomized tests for this comparison is 10 total**.** Figure 2 provides the test list in terms of VMS test structure. Each line indicates a new test and each test indicates a test directory structure where VMS can find test specific configuration information. The VMS seed generation options are also provided.

```
parms_tree/RUN_CAPTURE_MODE/mxtcpwm_full_random_test                -num_seeds 10 -rand_seed random  #T1
parms_tree/RUN_PWMDT_MODE/mxtcpwm_full_random_test                  -num_seeds 10 -rand_seed random  #T2
parms_tree/RUN_PWMPR_MODE/mxtcpwm_full_random_test                  -num_seeds 10 -rand_seed random  #T3
parms_tree/RUN_PWM_MODE/mxtcpwm_full_random_test                    -num_seeds 10 -rand_seed random  #T4
parms_tree/RUN_QUAD_RANGE0_CMP_MODE/mxtcpwm_full_random_test        -num_seeds 10 -rand_seed random  #T5
parms_tree/RUN_QUAD_RANGE0_MODE/mxtcpwm_full_random_test            -num_seeds 10 -rand_seed random  #T6
parms_tree/RUN_QUAD_RANGE1_CAPT_MODE/mxtcpwm_full_random_test       -num_seeds 10 -rand_seed random  #T7
parms_tree/RUN_QUAD_RANGE1_CMP_MODE/mxtcpwm_full_random_test        -num_seeds 10 -rand_seed random  #T8
parms_tree/RUN_SR_MODE/mxtcpwm_full_random_test                     -num_seeds 10 -rand_seed random  #T9
parms_tree/RUN_TIMER_MODE/mxtcpwm_full_random_test                  -num_seeds 10 -rand_seed random  #T10
```

Figure 2. TB1: Example Test List

As defined here, there are 10 tests all indicating that 10 random seeds should be generated for each. This would create 100 tests for every regression run targeting this test list. Data from nine regressions targeting this test list was captured in Table 2 and Table 3. Table 2 shows the number of seeds executed for each test per regression run and highlights the number of seeds kept per test after a final ranking step.

TABLE 2
TB1: SHOTGUN APPROACH: TEN TEST REGRESSION WITH TEN SEEDS EACH

Test #	Reg 1	Reg 2	Reg 3	Reg 4	Reg 5	Reg 6	Reg 7	Reg 8	Reg 9	Totals	Seeds Kept Per Test
T_1	10	10	10	10	10	10	10	10	10	90	13
T_2	10	10	10	10	10	10	10	10	10	90	41
T_3	10	10	10	10	10	10	10	10	10	90	8
T_4	10	10	10	10	10	10	10	10	10	90	38
T_5	10	10	10	10	10	10	10	10	10	90	20
T_6	10	10	10	10	10	10	10	10	10	90	6
T_7	10	10	10	10	10	10	10	10	10	90	8
T_8	10	10	10	10	10	10	10	10	10	90	9
T_9	10	10	10	10	10	10	10	10	10	90	6
T_{10}	10	10	10	10	10	10	10	10	10	90	11
Totals	100	100	100	100	100	100	100	100	100	900	160

As shown, a total of 900 tests were executed and only 160 of those tests contribute to coverage. This indicates 18% of the tests should be kept to attain the same coverage level achieved. For the same regressions, Table 3 provides coverage details, number of new contributing tests, final rank contribution, and total wall clock time for each regression.

TABLE 3
TB1: SHOTGUN APPROACH: COVERAGE DETAILS

Regr #	Coverage %	Tests Run	Contributing Tests per Iteration	Final Rank Contribution	Total Wall Clock Time (s)	Total Kept Seeds
1	76.05	100	69	19	22612.77	69
2	76.26	100	52	16	44114.21	101
3	76.74	100	33	17	65462.10	120
4	76.98	100	30	16	86872.50	135
5	76.99	100	30	19	107173.24	146

6	77.04	100	28	22	128018.57	158
7	77.24	100	27	24	147417.34	159
8	77.47	100	14	12	166907.69	158
9	**77.47**	100	15	15	**186266.24**	**160**

The total coverage attained using this method was 77.47%. The total wall clock time of all tests executed was 186266.24 sec or 51.74 hrs. As stated before, while 900 tests were executed only 160 contributed to the final coverage number for a contribution rate of 18%. It should be noted that new test-seed pairs may replace previous pairs that contributed to coverage. In this example 69 tests from regression 1 were considered contributing, but this number reduced to 19 after all regressions had completed.

Table 4 and Table 5 provide data while running the same initial regression as before, but explores results of the algorithm where $\mathbf{W_s}$ = 2 and $\mathbf{W_{fc}}$ = 1. The intent is to double the number of contributing seeds for a test from a previous regression and stop regressions once no tests contribute to incremental coverage.

TABLE 4
TB1: ALGORITHM WITH W_s = 2, W_{fc} = 1: TEN TEST REGRESSION WITH TEN INITIAL SEEDS EACH

Test #	Reg 1	Reg 2	Reg 3	Reg 4	Reg 5	Reg 6	Reg 7	Reg 8	Reg 9	Totals	Seeds Kept Per Test
T_1	10	16	14	2	2	2	N/A	N/A	N/A	46	13
T_2	10	20	34	38	30	16	N/A	N/A	N/A	148	41
T_3	10	10	6	2	0	0	N/A	N/A	N/A	28	8
T_4	10	20	30	36	32	20	N/A	N/A	N/A	148	38
T_5	10	16	18	14	6	4	N/A	N/A	N/A	68	20
T_6	10	6	2	0	0	0	N/A	N/A	N/A	18	6
T_7	10	10	6	2	0	0	N/A	N/A	N/A	28	8
T_8	10	14	4	2	0	0	N/A	N/A	N/A	30	9
T_9	10	14	8	4	4	0	N/A	N/A	N/A	40	6
T_{10}	10	12	12	8	2	2	N/A	N/A	N/A	46	11
Totals	100	138	134	108	76	44	0	0	0	600	160

In this example only 600 total tests were required to achieve comparable coverage results versus the shotgun approach for a test count savings of 33%. Additionally, three less regressions were required as no tests contributed after Regression 6. For the same regressions, Table 5 provides coverage details, number of new contributing tests, final rank contribution, and total wall clock time for each regression.

TABLE 5
TB1: ALGORITHM WITH W_s = 2, W_{fc} = 1: COVERAGE DETAILS

Regr #	Coverage %	Tests Run	Contributing Tests per Iteration	Final Rank Contribution	Total Wall Clock Time (s)	Total Kept Seeds
1	76.05	100	69	27	22612.77	69
2	76.68	138	67	35	54187.42	114
3	77.55	134	54	37	87251.37	134
4	77.78	108	38	25	113711.20	148
5	78.00	76	22	23	133868.65	152
6	**78.00**	44	13	13	**144371.49**	**160**
7	N/A	N/A	N/A	N/A	N/A	N/A
8	N/A	N/A	N/A	N/A	N/A	N/A
9	N/A	N/A	N/A	N/A	N/A	N/A

The total coverage attained using this method was 78.00%, which was slightly better than 77.47% as provided by the shotgun approach. This is likely due to the seeds chosen, not the algorithm. The total wall clock time of all tests executed was 144371.49 sec or 40.10 hrs for a reduction of 22%. The contribution rate increased to 27% as only 600 tests were executed and 160 of those contributed to coverage.

As a final example Table 6 and Table 7 provide data while running the same initial regression as before, but explores results of the algorithm where $\mathbf{W_s}$ = 2 and $\mathbf{W_{fc}}$ = 2. The intent is to double the number of contributing seeds for a test from a previous regression, double seeds for a test that had fully contributing seeds from a previous run, and stop regressions once no tests contribute to incremental coverage.

TABLE 6
TB1: ALGORITHM WITH $W_s = 2$, $W_{fc} = 2$: TEN TEST REGRESSION WITH TEN INITIAL SEEDS EACH

Test #	Reg 1	Reg 2	Reg 3	Reg 4	Reg 5	Reg 6	Reg 7	Reg 8	Reg 9	Totals	Seeds Kept Per Test
T_1	10	16	12	8	4	6	N/A	N/A	N/A	56	12
T_2	10	40	48	40	34	20	N/A	N/A	N/A	192	51
T_3	10	10	4	2	0	0	N/A	N/A	N/A	28	6
T_4	10	40	58	44	24	10	N/A	N/A	N/A	186	45
T_5	10	16	10	6	2	0	N/A	N/A	N/A	44	14
T_6	10	6	6	2	0	0	N/A	N/A	N/A	24	5
T_7	10	10	2	0	0	0	N/A	N/A	N/A	22	6
T_8	10	14	6	6	2	0	N/A	N/A	N/A	38	7
T_9	10	14	6	2	0	0	N/A	N/A	N/A	32	7
T_{10}	10	12	12	6	4	2	N/A	N/A	N/A	46	10
Totals	100	178	164	116	70	38	0	0	0	666	163

In this example 666 tests were executed to achieve better coverage results versus the shotgun approach for a test count savings of 26%. As before, three less regressions were required as no tests contributed after regression 6. Based on results, T_2 and T_4 were doubly rewarded in Regression 2 for having seeds that fully contributed to coverage during Regression 1. Note that 3 more seeds were kept compared to both previous attempts emphasizing better coverage attained. For the same regressions, Table 7 provides coverage details, number of new contributing tests, final rank contribution, and total wall clock time for each regression.

TABLE 7
TB1: ALGORITHM WITH $W_s = 2$, $W_{fc} = 2$: COVERAGE DETAILS

Regr #	Coverage %	Tests Run	Contributing Tests per Iteration	Final Rank Contribution	Total Wall Clock Time (s)	Total Kept Seeds
1	76.05	100	69	28	22612.77	69
2	77.78	178	82	41	67125.55	126
3	78.03	164	58	35	111433.47	142
4	78.63	116	35	29	141276.32	151
5	79.02	70	20	19	159551.93	156
6	**79.02**	38	11	11	**169013.31**	**163**
7	N/A	N/A	N/A	N/A	N/A	N/A
8	N/A	N/A	N/A	N/A	N/A	N/A
9	N/A	N/A	N/A	N/A	N/A	N/A

The total coverage attained using this method was 79.02%, which was 1.55% better than 77.47% as provided by the shotgun approach. As the number of contributing tests increased, the coverage increase could be due to the algorithm properly focusing earlier on the fully contributing tests. Even if not the case, it should be noted that the coverage attained by the shotgun approach was achieved by Regression 2 using this algorithm. At this point only 278 tests had been executed with a total wall clock time of 67125.55 sec or 18.65 hrs. This is a savings of 69% in total tests executed and 64% in total wall clock time of all tests executed. The contribution rate at the point of Regression 2 was 45%.

Table 8 includes a summary of key points between all three methods tried across three separate run comparisons of the same test bench. Highlighted is that coverage found by the shotgun approach was attained much faster by both algorithmic methods in all three cases. This can also be seen in Figure 3 through graphs of coverage versus wall clock time for each method.

TABLE 8
TB1: SUMMARY OF METHODS TRIED

Method	Total Coverage %	Total Tests	Seeds Kept	Versus Shotgun Method			
				Test Reduction	Regression When Coverage Met	Test Count When Coverage Met	Wall Clock Time When Met (s)
R1							
Shotgun: 10 seeds each reg	77.47	900	160	N/A	9	900	186266.24

Algorithm: Ws = 2; Wfc = 1	78.00	600	160	300	3	372	87251.37
Algorithm: Ws = 2; Wfc = 2	79.02	666	163	234	2	278	67125.55
R2							
Shotgun: 10 seeds each reg	77.60	700	143	N/A	7	700	149216.24
Algorithm: Ws = 2; Wfc = 1	78.82	686	170	14	2	246	56666.6
Algorithm: Ws = 2; Wfc = 2	79.23	726	172	(26)	3	440	98925.80
R3							
Shotgun: 10 seeds each reg	77.82	900	164	N/A	9	900	217487.71
Algorithm: Ws = 2; Wfc = 1	77.99	668	158	232	6	632	154877.51
Algorithm: Ws = 2; Wfc = 2	78.45	660	167	240	3	510	126202.14

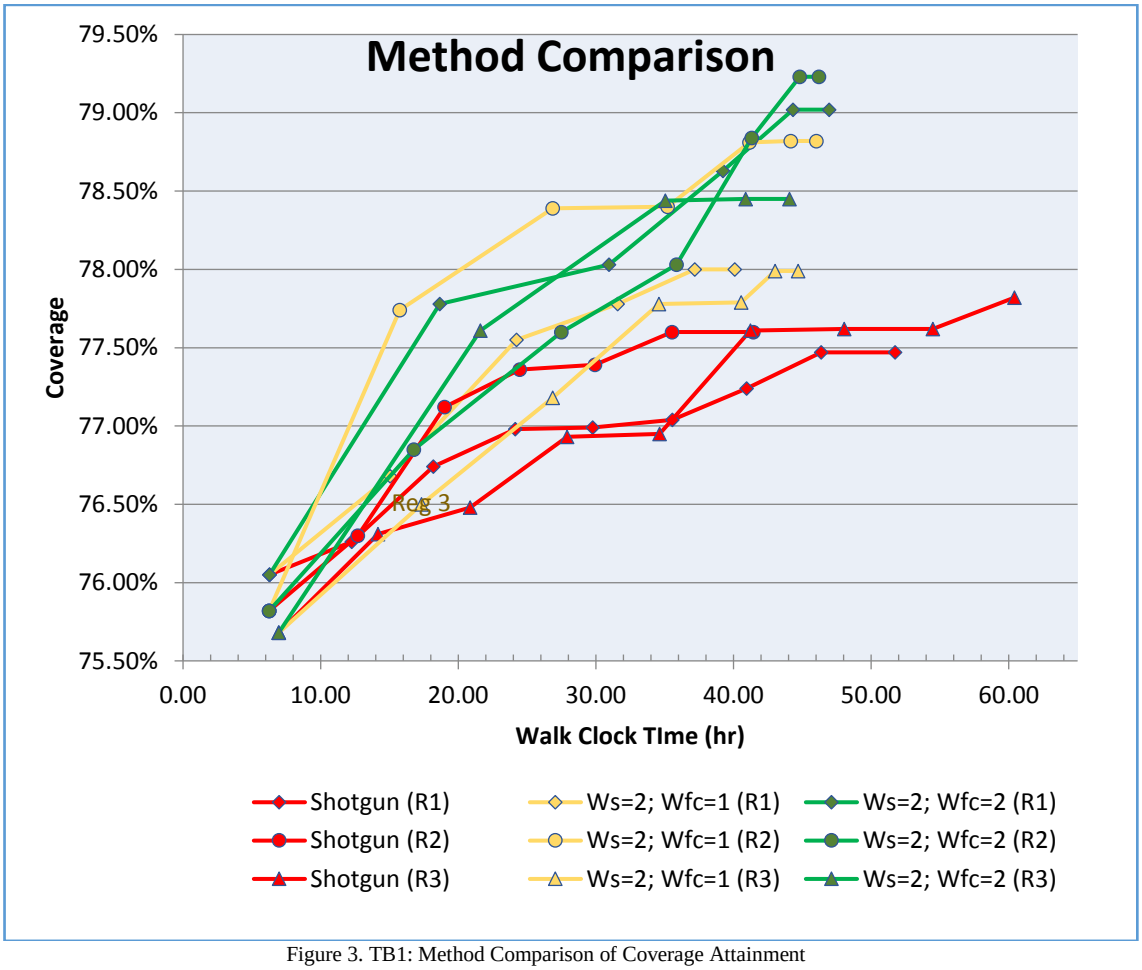

Figure 3. TB1: Method Comparison of Coverage Attainment

B. Test Bench 2 Results

As with the first test bench, there are 10 tests for the second test bench all indicating that 10 random seeds should be generated for each. This is shown in Figure 4.

```
m4cpuss_dw_random_transfer_test                -num_seeds 10 -rand_seed random  #T1
m4cpuss_dw_basic_transfer_test                 -num_seeds 10 -rand_seed random  #T2
cpuss_fault_random_test                        -num_seeds 10 -rand_seed random  #T3
cpuss_irq_test                                 -num_seeds 10 -rand_seed random  #T4
cpuss_rom_access_test                          -num_seeds 10 -rand_seed random  #T5
cpuss_ram_access_test                          -num_seeds 10 -rand_seed random  #T6
cpuss_flash_access_test                        -num_seeds 10 -rand_seed random  #T7
m4cpuss_random_mem_bus_infra_access_test       -num_seeds 10 -rand_seed random  #T8
cpuss_protection_reg_test                      -num_seeds 10 -rand_seed random  #T9
cpuss_misc_reg_test                            -num_seeds 10 -rand_seed random  #T10
```

Figure 4. TB2: Example Test List

This would create 100 tests for every regression run targeting this test list. Data from nine regressions targeting this test list was captured in Table 9 and Table 10. Table 9 shows the number of seeds executed for each test per regression run and highlights the number of seeds kept per test after a final ranking step.

TABLE 9
TB2: SHOTGUN APPROACH: TEN TEST REGRESSION WITH TEN SEEDS EACH

Test #	Reg 1	Reg 2	Reg 3	Reg 4	Reg 5	Reg 6	Reg 7	Reg 8	Reg 9	Totals	Seeds Kept Per Test
T_1	10	10	10	10	10	10	10	10	10	90	41
T_2	10	10	10	10	10	10	10	10	10	90	16
T_3	10	10	10	10	10	10	10	10	10	90	6
T_4	10	10	10	10	10	10	10	10	10	90	2
T_5	10	10	10	10	10	10	10	10	10	90	1
T_6	10	10	10	10	10	10	10	10	10	90	2
T_7	10	10	10	10	10	10	10	10	10	90	23
T_8	10	10	10	10	10	10	10	10	10	90	66
T_9	10	10	10	10	10	10	10	10	10	90	1
T_{10}	10	10	10	10	10	10	10	10	10	90	1
Totals	100	100	100	100	100	100	100	100	100	900	159

As shown, a total of 900 tests were executed and only 159 of those tests contribute to coverage. This indicates 18% of the tests should be kept to attain the same coverage level achieved. For the same regressions, Table 10 provides coverage details, number of new contributing tests, final rank contribution, and total wall clock time for each regression.

TABLE 10
TB2: SHOTGUN APPROACH: COVERAGE DETAILS

Regr #	Coverage %	Tests Run	Contributing Tests per Iteration	Final Rank Contribution	Total Wall Clock Time (s)	Total Kept Seeds
1	72.88%	100	50	20	112214.64	50
2	73.20%	100	32	17	245781.12	65
3	73.35%	100	29	19	407193.75	91
4	73.43%	100	29	15	554897.93	105
5	73.89%	100	25	19	665679.11	123
6	73.90%	100	24	13	780300.98	134
7	73.93%	100	24	21	895211.77	147
8	73.94%	100	20	19	1005816.78	154
9	**73.94%**	100	16	16	**1154565.67**	**159**

The total coverage attained using this method was 73.94%. The total wall clock time of all tests executed was 1154565.67 sec or 320.71 hrs. As stated before, while 900 tests were executed only 159 contributed to the final coverage number for a contribution rate of 18%. It should be noted that new test-seed pairs may replace previous pairs that contributed to coverage. In this example 50 tests from regression one were considered contributing, but this number reduced to 20 after all regressions had completed.

Table 11 and Table 12 provide data while running the same initial regression for test bench two, but explores results of the algorithm where $\mathbf{W_s} = 2$ and $\mathbf{W_{fc}} = 1$.

TABLE 11
TB2: ALGORITHM WITH $W_s = 2$, $W_{fc} = 1$: TEN TEST REGRESSION WITH TEN INITIAL SEEDS EACH

Test #	Reg 1	Reg 2	Reg 3	Reg 4	Reg 5	Reg 6	Reg 7	Reg 8	Reg 9	Totals	Seeds Kept Per Test
T_1	10	20	30	34	28	18	8	2	2	152	44
T_2	10	20	24	12	6	0	0	0	0	72	16
T_3	10	14	10	2	0	0	0	0	0	36	6
T_4	10	6	2	0	0	0	0	0	0	18	3
T_5	10	4	0	0	0	0	0	0	0	14	0
T_6	10	2	0	0	0	0	0	0	0	12	0
T_7	10	10	8	2	2	0	0	0	0	32	5
T_8	10	20	36	62	90	82	66	54	38	458	131
T_9	10	2	0	0	0	0	0	0	0	12	1
T_{10}	10	2	0	0	0	0	0	0	0	12	1
Totals	100	100	110	112	126	100	74	56	40	818	207

In this example the algorithm narrowed down to two tests that provided the most contribution with regards to seeds. For T_8, many more seeds were identified. T_5 and T_6 were eliminated as contributors pointing out the coverage redundancy of these tests with regards to others. Unlike test bench one, this method continued through Regression 9 trying to gain more coverage from T_1 and T_9. For the same regressions, Table 12 provides coverage details, number of new contributing tests, final rank contribution, and total wall clock time for each regression.

TABLE 12
TB2: ALGORITHM WITH $W_s = 2$, $W_{fc} = 1$: COVERAGE DETAILS

Regr #	Coverage %	Tests Run	Contributing Tests per Iteration	Final Rank Contribution	Total Wall Clock Time (s)	Total Kept Seeds
1	72.88%	100	50	17	112214.64	50
2	73.56%	100	55	25	327634.83	88
3	73.68%	110	56	37	773717.09	124
4	74.12%	112	63	30	1471147.61	166
5	74.16%	126	50	35	2587410.89	184
6	74.19%	100	37	24	3378029.35	202
7	74.38%	74	28	24	3954061.21	203
8	74.39%	56	20	18	4614303.64	202
9	**74.39%**	40	7	7	**5009957.93**	**207**

The total coverage attained using this method was 74.39%, which was slightly better than 73.94% as provided by the shotgun approach. While many more seeds were chosen and coverage did improve, it was at a very high cost with regards to wall clock time consumed. The total wall clock time of all tests executed was 5009957.93 sec or 1397.65 hrs. The two tests that the algorithm pushed ahead with happen to be the longest running tests causing run time to balloon. Ideally these longer running tests need to be separated into shorter sequences if possible. Due to the length of these tests, the algorithm with $\mathbf{W_s} = 2$ and $\mathbf{W_{fc}} = 2$ was not investigated for this paper.

CONCLUSION

The focus of this paper has been on an efficient approach to choosing SystemVerilog seeds for Constrained Random Verification. The algorithm discussed utilizes a combination of industry provided tools and a verification management system developed by Cypress. Vendor provided tools can rank regression results identifying contributing and non-contributing tests for both functional and code coverage. VMS takes advantage of ranking features and combines them with automatic seed generation to create an intelligent and automated seed selection algorithm. Shown in this paper is the algorithm and results from three different methods. One method does not use intelligent selection and blindly applies the same number of random seeds for each test in each regression regardless of prior test-seed contribution. The other two methods investigate the algorithm of this paper with different weightings for seed generation based on previous contribution results.

There are three key components of this algorithm. First, if a test from a previous regression does not contribute to coverage, then there is no need to execute this test further. Second, if a test did contribute to coverage, then it should be rewarded with more seeds in quantity proportional to the number of contributing seeds from the previous run. In other words, focus on seed generation is shifted to tests that matter, and away from those that do not.

Finally, by the nature of the algorithm, as a test contributes fewer seeds it will provide diminishing returns over time. Incremental test contribution will tend to zero eliminating the test from consideration.

From results shown here, this algorithm can find contributing seeds and achieve maximum coverage much quicker than blindly applying more seeds from one regression to the next. In some cases, with 69% fewer tests and 64% faster in total cumulative wall clock time of all tests executed. Additionally, it is possible to increase coverage if more seeds are provided to tests that contribute. Overall this algorithm shows great promise in improving seed selection efficiency freeing up valuable hardware and license resources for other tasks.

As shown through the second test bench, consideration must be given for test run time. If a test that runs much longer than the others is contributing small incremental coverage, it likely should not be given more seeds, as resources are not being efficiently used. Future work on this paper will focus on test or sequence efficiency to make more intelligent decisions about seed application and provide proper feedback to the test bench owner.

APPENDIX

TABLE 13
TB1: SHOTGUN APPROACH: TEN TEST REGRESSION WITH TEN SEEDS EACH (2ND ITERATION)

Test #	Reg 1	Reg 2	Reg 3	Reg 4	Reg 5	Reg 6	Reg 7	Reg 8	Reg 9	Totals	Seeds Kept Per Test
T_1	10	10	10	10	10	10	10	N/A	N/A	70	13
T_2	10	10	10	10	10	10	10	N/A	N/A	70	35
T_3	10	10	10	10	10	10	10	N/A	N/A	70	8
T_4	10	10	10	10	10	10	10	N/A	N/A	70	33
T_5	10	10	10	10	10	10	10	N/A	N/A	70	15
T_6	10	10	10	10	10	10	10	N/A	N/A	70	7
T_7	10	10	10	10	10	10	10	N/A	N/A	70	7
T_8	10	10	10	10	10	10	10	N/A	N/A	70	9
T_9	10	10	10	10	10	10	10	N/A	N/A	70	4
T_{10}	10	10	10	10	10	10	10	N/A	N/A	70	12
Totals	100	100	100	100	100	100	100	0	0	700	143

TABLE 14
TB1: SHOTGUN APPROACH: COVERAGE DETAILS (2ND ITERATION)

Regr #	Coverage %	Tests Run	Contributing Tests per Iteration	Final Rank Contribution	Total Wall Clock Time (s)	Total Kept Seeds
1	75.82%	100	78	17	22550.59	73
2	76.30%	100	54	26	45756.24	106
3	77.12%	100	35	21	68434.57	117
4	77.36%	100	35	20	88095.54	126
5	77.39%	100	28	25	107759.71	134
6	77.60%	100	22	15	127904.95	142
7	**77.60%**	100	19	19	**149216.24**	**143**
8	N/A	N/A	N/A	N/A	N/A	N/A
9	N/A	N/A	N/A	N/A	N/A	N/A

TABLE 15
TB1: ALGORITHM WITH $W_s = 2$, $W_{fc} = 1$: TEN TEST REGRESSION WITH TEN INITIAL SEEDS EACH (2ND ITERATION)

Test #	Reg 1	Reg 2	Reg 3	Reg 4	Reg 5	Reg 6	Reg 7	Reg 8	Reg 9	Totals	Seeds Kept Per Test
T_1	10	18	18	10	6	4	2	N/A	N/A	68	14
T_2	10	20	36	44	38	22	14	N/A	N/A	184	52
T_3	10	14	12	4	2	0	0	N/A	N/A	42	9
T_4	10	18	32	34	26	14	10	N/A	N/A	144	43
T_5	10	16	18	8	8	6	2	N/A	N/A	68	16
T_6	10	10	6	4	4	2	0	N/A	N/A	36	6
T_7	10	10	8	4	0	0	0	N/A	N/A	32	5
T_8	10	12	10	2	2	0	0	N/A	N/A	36	7
T_9	10	14	8	4	0	0	0	N/A	N/A	36	6
T_{10}	10	14	12	2	2	0	0	N/A	N/A	40	12
Totals	100	146	160	116	88	48	28	0	0	686	170

TABLE 16
TB1: ALGORITHM WITH $W_s = 2$, $W_{fc} = 1$: COVERAGE DETAILS (2ND ITERATION)

Regr #	Coverage %	Tests Run	Contributing Tests per Iteration	Final Rank Contribution	Total Wall Clock Time (s)	Total Kept Seeds
1	75.82%	100	78	22	22550.59	73
2	77.74%	146	80	38	56666.6	125
3	78.39%	160	58	36	96710.2	150
4	78.40%	116	44	34	126714.79	162
5	78.81%	88	24	22	148147.57	166
6	78.82%	48	14	13	158999.09	167
7	**78.82%**	28	5	5	**165658.06**	**170**
8	N/A	N/A	N/A	N/A	N/A	N/A
9	N/A	N/A	N/A	N/A	N/A	N/A

TABLE 17
TB1: ALGORITHM WITH $W_s = 2$, $W_{fc} = 2$: TEN TEST REGRESSION WITH TEN INITIAL SEEDS EACH (2ND ITERATION)

Test #	Reg 1	Reg 2	Reg 3	Reg 4	Reg 5	Reg 6	Reg 7	Reg 8	Reg 9	Totals	Seeds Kept Per Test
T_1	10	18	16	12	12	4	0	N/A	N/A	72	14
T_2	10	40	56	46	24	14	6	N/A	N/A	196	52
T_3	10	14	6	0	0	0	0	N/A	N/A	30	7
T_4	10	18	34	44	36	32	12	N/A	N/A	186	44
T_5	10	16	16	16	6	0	0	N/A	N/A	64	16
$14T_6$	10	10	8	4	2	0	0	N/A	N/A	34	7
T_7	10	10	10	2	0	0	0	N/A	N/A	32	6
T_8	10	12	10	2	2	2	0	N/A	N/A	38	7
T_9	10	14	8	6	0	0	0	N/A	N/A	38	9
T_{10}	10	14	10	2	0	0	0	N/A	N/A	36	10
Totals	100	166	174	134	82	52	18	0	0	726	172

TABLE 18
TB1: ALGORITHM WITH $W_s = 2$, $W_{fc} = 2$: COVERAGE DETAILS (2ND ITERATION)

Regr #	Coverage %	Tests Run	Contributing Tests per Iteration	Final Rank Contribution	Total Wall Clock Time (s)	Total Kept Seeds
1	75.82%	100	73	22	22550.59	73
2	76.85%	166	87	43	60403.35	130
3	77.60%	174	66	44	98925.88	160
4	78.03%	134	41	27	129080.89	166
5	78.84%	82	26	22	148795.64	171
6	79.23%	52	6	9	161360.5	172
7	**79.23%**	18	5	5	**166379.37**	**172**
8	N/A	N/A	N/A	N/A	N/A	N/A
9	N/A	N/A	N/A	N/A	N/A	N/A

TABLE 19
TB1: SHOTGUN APPROACH: TEN TEST REGRESSION WITH TEN SEEDS EACH (3RD ITERATION)

Test #	Reg 1	Reg 2	Reg 3	Reg 4	Reg 5	Reg 6	Reg 7	Reg 8	Reg 9	Totals	Seeds Kept Per Test
T_1	10	10	10	10	10	10	10	10	10	90	14
T_2	10	10	10	10	10	10	10	10	10	90	41
T_3	10	10	10	10	10	10	10	10	10	90	8
T_4	10	10	10	10	10	10	10	10	10	90	46
T_5	10	10	10	10	10	10	10	10	10	90	19
T_6	10	10	10	10	10	10	10	10	10	90	4
T_7	10	10	10	10	10	10	10	10	10	90	9
T_8	10	10	10	10	10	10	10	10	10	90	7
T_9	10	10	10	10	10	10	10	10	10	90	8
T_{10}	10	10	10	10	10	10	10	10	10	90	8
Totals	100	100	100	100	100	100	100	100	100	900	164

TABLE 20
TB1: SHOTGUN APPROACH: COVERAGE DETAILS (3RD ITERATION)

Regr #	Coverage %	Tests Run	Contributing Tests per Iteration	Final Rank Contribution	Total Wall Clock Time (s)	Total Kept Seeds
1	75.68%	100	78	19	25031.87	78
2	76.31%	100	53	20	50981.97	103
3	76.48%	100	40	17	75086.42	122
4	76.93%	100	36	23	100493.18	137
5	76.95%	100	32	16	124673.84	151
6	77.61%	100	22	19	148438.57	153
7	77.62%	100	22	17	172955.22	157
8	77.62%	100	21	18	196164.5	160
9	**77.82%**	100	15	15	**217487.71**	**164**

TABLE 21

TB: ALGORITHM WITH $W_s = 2$, $W_{fc} = 1$: TEN TEST REGRESSION WITH TEN INITIAL SEEDS EACH (3RD ITERATION)

Test #	Reg 1	Reg 2	Reg 3	Reg 4	Reg 5	Reg 6	Reg 7	Reg 8	Reg 9	Totals	Seeds Kept Per Test
T_1	10	20	12	6	4	2	0	N/A	N/A	54	13
T_2	10	20	28	34	34	18	14	N/A	N/A	158	46
T_3	10	12	8	2	2	0	0	N/A	N/A	34	6
T_4	10	20	38	40	28	10	6	N/A	N/A	152	39
T_5	10	14	16	12	4	2	2	N/A	N/A	60	15
T_6	10	14	2	0	0	0	0	N/A	N/A	26	5
T_7	10	16	10	4	0	0	0	N/A	N/A	40	6
T_8	10	12	8	6	2	2	2	N/A	N/A	42	8
T_9	10	14	6	4	4	0	0	N/A	N/A	38	6
T_{10}	10	14	10	8	6	4	2	N/A	N/A	54	14
Totals	100	156	138	116	84	38	26	0	0	658	158

TABLE 22

TB1: ALGORITHM WITH $W_s = 2$, $W_{fc} = 1$: COVERAGE DETAILS (3RD ITERATION)

Regr #	Coverage %	Tests Run	Contributing Tests per Iteration	Final Rank Contribution	Total Wall Clock Time (s)	Total Kept Seeds
1	75.68%	100	78	23	25031.87	78
2	76.50%	156	69	25	62415.58	120
3	77.18%	138	58	40	96712.67	146
4	77.78%	116	42	33	124453.35	150
5	77.79%	84	19	18	146013.16	151
6	77.99%	38	13	12	154877.51	158
7	**77.99%**	26	7	7	**160927.54**	**158**
8	N/A	N/A	N/A	N/A	N/A	N/A
9	N/A	N/A	N/A	N/A	N/A	N/A

TABLE 23

TB1: ALGORITHM WITH $W_s = 2$, $W_{fc} = 2$: TEN TEST REGRESSION WITH TEN INITIAL SEEDS EACH (3RD ITERATION)

Test #	Reg 1	Reg 2	Reg 3	Reg 4	Reg 5	Reg 6	Reg 7	Reg 8	Reg 9	Totals	Seeds Kept Per Test
T_1	10	40	16	4	2	N/A	N/A	N/A	N/A	72	11
T_2	10	40	50	38	22	N/A	N/A	N/A	N/A	160	51
T_3	10	12	10	2	2	N/A	N/A	N/A	N/A	36	8
T_4	10	40	52	30	12	N/A	N/A	N/A	N/A	144	42
T_5	10	14	16	8	4	N/A	N/A	N/A	N/A	52	16
T_6	10	14	8	0	0	N/A	N/A	N/A	N/A	32	6
T_7	10	16	12	4	2	N/A	N/A	N/A	N/A	44	9
T_8	10	12	12	8	0	N/A	N/A	N/A	N/A	42	7
T_9	10	14	6	4	4	N/A	N/A	N/A	N/A	38	7
T_{10}	10	14	12	4	0	N/A	N/A	N/A	N/A	40	11
Totals	100	216	194	102	48	0	0	0	0	660	167

TABLE 24

TB1: ALGORITHM WITH $W_s = 2$, $W_{fc} = 2$: COVERAGE DETAILS (3RD ITERATION)

Regr #	Coverage %	Tests Run	Contributing Tests per Iteration	Final Rank Contribution	Total Wall Clock Time (s)	Total Kept Seeds
1	75.68%	100	78	27	25031.87	78
2	77.61%	216	97	59	77810.25	135
3	78.44%	194	51	44	126202.14	159
4	78.45%	102	24	22	147154.32	164
5	78.45%	48	15	15	158629.32	167
6	N/A	N/A	N/A	N/A	N/A	N/A
7	N/A	N/A	N/A	N/A	N/A	N/A
8	N/A	N/A	N/A	N/A	N/A	N/A
9	N/A	N/A	N/A	N/A	N/A	N/A