Dynamic Regression Suite Generation Using Coverage-Based Clustering

Shahid Ikram
5086838970
Shahid.ikram@cavium.com

Jim Ellis
5086838826
Jim.Ellis@cavium.com

Cavium Networks, 600 Nickerson Road, Marlborough, MA 01752

***Abstract*- Regressions are an important and expensive activity in the complex VLSI designs. Whenever a part of a design is changed, verification needs to find if the change is working and the change has not broken other working pieces. We present here a methodology based on the coverage-based clustering of modules and the ranking of the associated tests. The regressions' lists are generated dynamically based upon the changes made in RTL using RTL and clustering information. Empirical studies are performed with code modifications through fault mutations. The results show measurable enhancements over expert-knowledge-based regressions.**

I. Introduction

A VLSI simulation consists of RTL modules and a verification environment. At an abstract level, a verification environment [5] can be viewed as a composition of a bunch of tests which run during simulations in addition to monitoring and detection mechanisms. The RTL and the verification environment are living entities during the life of a project and evolve with the project. The RTL usually consists of many modules. Each module probably has multiple instantiations. An RTL design is holistic in the sense that sum is greater than parts. This implies mutual dependencies of the RTL instances to achieve the goals of the design implementation. Therefore as RTL changes, verifiers needs to ensure:

1. The new functionality is working.
2. It has not broken anything else in the design.

The first part is tested using "resolution" tests and the second part is taken care through regressions suites [11]. Both of these sets of tests are a subset of a total number of tests available and are created by the experts of the design. Generally speaking, regressions incur the largest costs to the VLSI projects in terms of time, simulation, compute and human resources. The research question we are posing is:

Is there a way to create these lists (resolutions and regressions) dynamically depending upon the changes in the design using RTL information and the available coverage data?

Stating more formally, when RTL is modified what subset of available test-suite should be simulated that will:

1. Ascertain coverage of the changes made.
2. Ensure another related functionality is not affected/broken by these changes.
3. Take minimal resources in terms of time, computation licenses etc.

A static analysis of the RTL can yield the dependencies among RTL modules but verification tests are dynamic entities with constraints and random seeds. Therefore an approach that can analyze the dynamics of the system have the potential to provide better insight.

II. Motivation

To measure the quality of the verification environments, different coverage metrics [5] are collected. Figure 1 shows a three-dimensional perspective of two different designs. The design on the left has 28 tests and 500+ modules, while the design on the right has 94 tests and 365 modules. The z-axis represents the accumulative coverage scores of the designs.

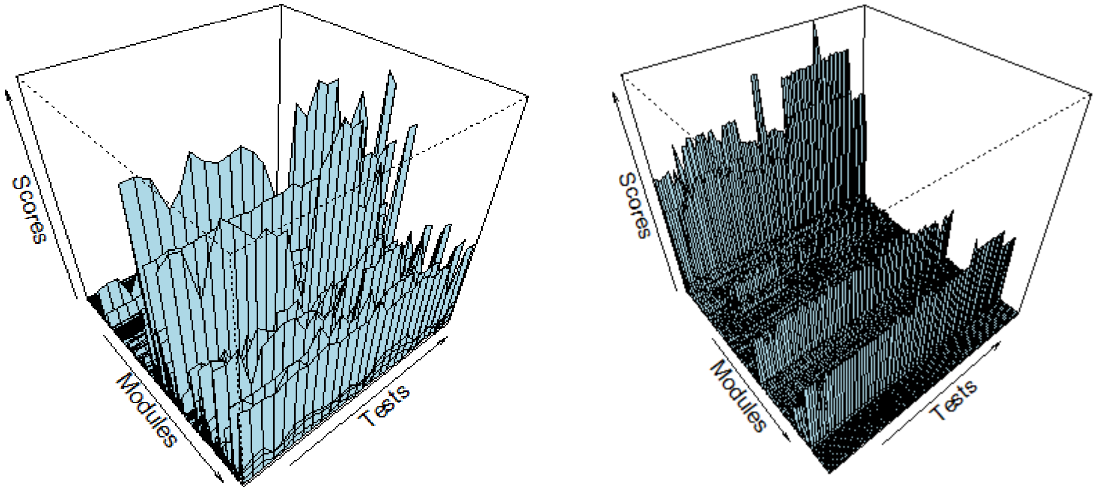

Figure 1: Coverage Scores

One can observe that some modules are showing similar coverage patterns for the given test suites. Intuitively, these patterns make sense because of the hardware concurrency and inter-dependency i.e. when a test is exploring a part of a design, it will be activating its' related logic as well. However, to qualify these observations we need to answer the following questions:

1. Are these patterns mere statistical coincidences or they do represent correlated modules?
2. Can these patterns be identified algorithmically?

The answer to the first question begs for some empirical studies, which will be shown in the later part of this paper. The answer to the second question is "Yes" as statistical learning techniques [6] [8] [9] provides a number of methods to cluster similar behaviors.

III. Related Work

The main motivation of our work was Monica Farkash's pioneering work in the application of statistical techniques to different challenges in VLSI verification[3][4][18]. Monica used coverage to cluster tests [18]. However, we used RTL hierarchal information and statistical analysis of the modules' coverage to form clusters of the RTL modules and used these clusters to generate regression test-lists. Arafeen etc. [11] make use of user-requirements through term extraction of the documentation and by forming test clusters based upon this information. They also use fault mutations to test their heuristics. This is in clear contrast to our work that uses RTL hierarchal information and coverage collection and does not require user-requirements documentation.

III. Background

A. Tests Classification

A test in our case is a process that simulates a design. It uses stimuli to drive the design, constraints to restrain the inputs to the legal values, monitors to watch the progress and checkers to validate the outputs [5]. A test can be directed in which case it is checking a specific scenario or constrained random, in which case maybe checking a variety of cases. We use coverage [2] as a measure of the quality of a test. We call a test *obsolete* if it no more relevant to the verification i.e. it is generating coverage below a certain threshold. We call a test *redundant* if it is covering a functionality unrelated to the changes made in the design. Finally, the tests that cover the affected areas of the design are called *retestable* and these are the tests that should be run after the modifications [10].

In any verification environment, after any changes are committed, we want to divide the available test suite into these three classes i.e. obsolete, redundant and retestable. Generally, it is done by experts in the design but our effort is directed toward providing quantitative measures to assist in these decisions and create this *retestable* subset of the available tests' set dynamically.

B. Regression Suites

Regression is defined as running a group of tests after the design goes through some changes. The *retestable(defined above)* subset of the available test list is the starting point. A number of strategies can be employed depending upon the design verification methodology to define different regression suites [13]. A *soft* regression list may consist of a small number of tests constrained by the time-limits while a *nightly* regression list may not be constrained by the time and can cover a larger design state space.

C. Fault Simulation

Fault simulators [12] are used to test the quality of the verification environment. They provide metrics for controllability and observability of the verification stimuli. In our setting, we used a fault mutated design to test the quality of dynamically generated regression and resolution test lists.

IV. Methodology

Our methodology has two distinct components. The first one is about data collection and analysis and the second one about the application of the insight gained from the data analysis to dynamically generate regression lists. The outcome of the first step is a generation of the clusters of the related RTL modules. The result of the second step is a dynamically generated regression list to be used for the verification of the change(s) made in the design.

A. Data Collection and Analyses

Coverage data provides us a measure of how well a design is being exercised. A number of Coverage metrics are available like line coverage, condition coverage, branch coverage, path coverage, toggle coverage and functional coverage [2]. Functional coverage provides great insight into design evaluation but most of the times it has to be put in manually, so its existence is not guaranteed and hence we ignored it for this work. Nearly all the other coverage metrics are auto-generated and any one of them can be chosen as similarity criteria. However, we want to choose the one with the maximum amount of information i.e. one that shows maximum variance [13]. Therefore this choice is design dependent.

Once we have data available to us, the statistical question we are facing is what modules are correlated in terms of the RTL coverage obtained through the available tests. *Unsupervised learning* [6] is a machine learning process to draw inferences from a data-set without any outputs. The most common *unsupervised learning* algorithms are called *cluster analysis*, which is used to find patterns or grouping in an exploratory data-set. The idea is based upon some notion of similarity/dissimilarity between each pair of the items in the set [8] [9]. The items most similar to each other are grouped together. It is called unsupervised because we are trying to find out structure in the data without any response variable. A word of caution here is that these methods always return some grouping and the analyst have to decide if it is true to the design intentions. We will show how to overcome this shortcoming through baseline comparisons and experimentation.

Figure 2 shows an overview of our data collection and analysis process. Here are the few details about the individual steps involved in it:

1. Coverage Data Collection

We collect coverage data for all the available tests for all the types of the code-coverage including line, toggle, branch, condition etc. except the assertion coverage. We ignore the functional coverage and the assertion coverage as these are not globally available and we are interested in how and when different modules show similar coverage behaviors. The coverage data is generated and stored per test inside a coverage database [15].

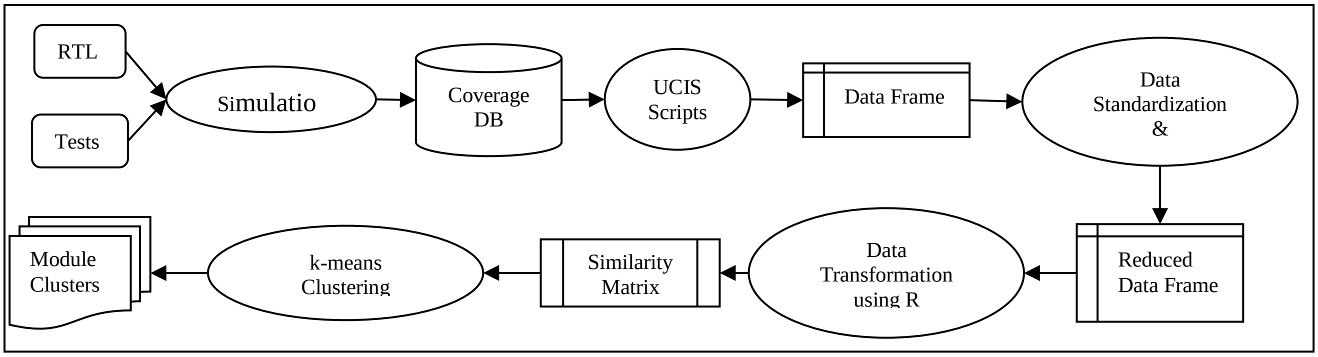

Figure 2: Data Collection and Analyses

2. *Coverage Data Extraction*

UCIS, the standardized interface to coverage databases [1], provides methods to extract coverage data. We collect coverage per module per test based. The output of this process is a Data-Frame (essentially a table with labeled columns)[17] in a comma-separated format. Each row of this table provides per module per test per seed coverage information. Each row also contains test's run time, depth of the module in the design hierarchy and the RTL filename. A few sample lines with selected columns of the table for one of our blocks are shown in Table 1.

Table 1: Coverage Data-Frame

Module	**Test**	**Time**	**Line**	**Toggle**	**Branch**
Bisr_ctl	Bar0_d	987	28	14	28
Bisr_dp	Bar0_d	987	13	914	4
Arb_rot	Bar0_d	987	5	3	2

3. *Data Standardization*

Data Standardization is a key process in data mining to convert the available data into a format that allows comparative analysis of different techniques and choices. In our application, the coverage scores for the different types of the code-coverage (line coverage, toggle coverage, expression coverage etc.) may vary by different margins and hence we need to *standardize* these numbers to make them comparable and to generate *a global score*. The process of the data standardization is performed by subtracting the average of each column from each of its entry and dividing it by its standard deviation [16]. We ignore the negative signs as we are only interested in coverage values. The global score is a product of coverage value of a module with its depth in the design [4] and we enhance our Data-Frame with an additional column called *Gscore* that shows this measure.

4. *Data Reduction*

An important step in the application of the data mining techniques is a reduction of the data to a useful subset without losing pertinent information. For this purpose, we use two different methods. The first one uses statistical information present in the data and the second one uses our knowledge of the design practices. The first method reduces the number of columns and the second method reduces the number of rows. These two techniques let us reduce the Data-Frame size around 70%.

a. *Selecting the best Column*

We look at what columns of the coverage Data-Frame contain most useful information with respect to the clustering of the modules. For this purpose, we look at the variance of each of the columns [13] [6] of the coverage Data-Frame including *GScore* and choose the one with the most variance. In most of our case studies, *GScore* proved to be the best discriminator but there were the cases when toggle coverage or line coverage gave us better results.

b. *Discarding the Rows*

In any large scale VLSI design, there are two distinct categories of the modules:

- *Common modules*: These are the modules which perform well-known functionality and are used repeatedly across different parts of the design (and in fact across different projects). The well-known examples of these are FIFOs, Arbiters, and Counters.
- *Application specific modules*: These are the modules created to implement the features of a design. The examples of these are input-packet processer, crypto-unit, compression-unit etc. They define blocks of the design that are verified individually before becoming part of the full-chip verification. Most of the tests are written at this level for functional verification.

Changes in the common modules are rare and generally require verification through the complete regression suite. We only focus here on application specific modules. All of our common modules sit in a common area and hence the list of these modules can be auto-generated and used to remove the related rows from the coverage Data-Frame.

5. *Similarity Matrix among Modules*

A similarity matrix among modules is a table with rows labeled with module names and the column labeled with test names. It is generated from the reduced coverage Data-Frame generated in the previous step. Figure 1 shows a perspective of two of these matrices from two different designs. This matrix provides us a mean to compare the behavior of different modules for the available tests. If for any two modules associated row vectors are behaving similarly, these modules are related otherwise they are not.

6. *Clustering the modules*

A k-means clustering method [6] tries to divide n observations into k groups using the distance from the group mean. We used this method to cluster the modules based upon similarity matrix generated in the previous step. Another choice is hierarchal clustering but not feasible for our case because of a large number of modules. The modules with relatively smaller distances (larger similarity) from each other are clustered together.

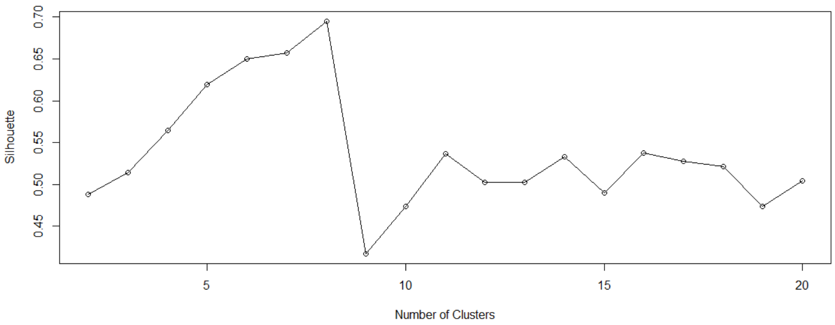

Figure 3: Silhouette for a design

The challenge with k-means clustering is that we have to make an initial guess for the k (the number of clusters) and increase or decrease the number to fit the clusters to our understanding of the design. To automate this process, we borrowed the idea “silhouette” from statistical learning methods [16]. Here is a formal definition of *Silhouette*:

> *(Average distance to those in the nearest neighboring cluster – Average distance to those in my cluster)/ The maximum of those two averages*

An ideal clustering will result in a silhouette of 1(or -1) and bad clustering will result in a silhouette of 0. For example, Figure 3 shows silhouette calculations for one of our case studies using R [17]. It shows that for this case, the best number of clusters will be 8.

7. *Validating the Clusters*

Anytime clustering is performed, it returns a grouping of items. Sometimes, this grouping is meaningful and other times it is not. Probably the best cluster validation technique is a design review to see if the clustering is true to our understanding of the design. Our observations from the case studies found clusters to be true to our design understanding around the optimum values generated by our Silhouette algorithm.

8. *Clusters’ Data-Frames*

The clustering information generated becomes a part of the Data-Frame as an additional column, categorizing each module in terms of a cluster number it belongs to. It effectively divides the coverage Data-Frame into as many smaller Data-Frames, one for each cluster.

B. *Regression list Generation*

We want to create a suitable regression list right after the changes are made to the RTL. What is important for us, is to figure out the best subset of the available test suite to cover the *radius of influence* of this change. This is a two-step process as shown in Figure 4. The first step is to find the related modules and the second step is to choose the best tests for these modules. The flow shows the steps and entities involved in this process.

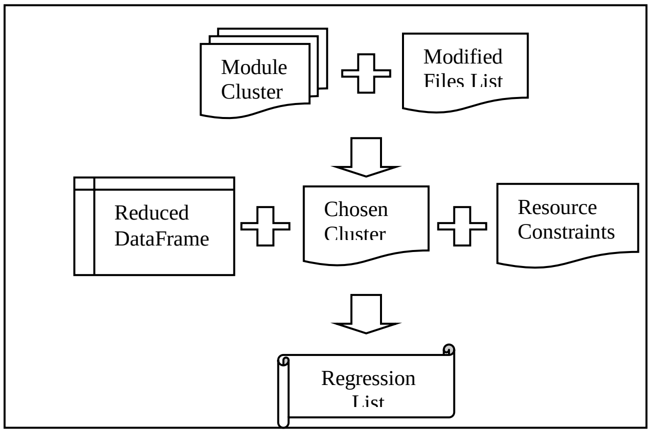

Figure 4: Regression list generation flow

Table 2: Tests list for a cluster

Tests	**aScore**	**sd**	**time**	**CV**
basic_msix	2482	269	1005	75.99
basic_swi_rsp_err	2482	269	1476	75.99
basic_prp_sml_qsize	2479	268	1239	75.87
perf_bw_basic_1_1028	2478	268	2039	75.88
basic_multi	2475	267	2323	75.75

1. *Modules' Selection*

Our coverage database contains files information. Most of the version control systems provide commands to generate the list of the files modified since the last check-out of the design. Moreover, generally, in our setup, we have one Verilog module per file. Therefore, using the names of the modified files, we find the modified modules. Once we have found the list of modified modules, we perform a two-step process to identify the related affected modules:

a) We include all the modules instantiated in the modified module(s), as well as those which are instantiated in the same module as the modified module(s) are.

b) We use the clustering information generated in coverage data analysis to find the rest of probably affected modules.

The outcome of this process is a list of modules that must be validated to ensure that the changes in the design have not broken anything else. These list of modules is used to select the cluster Data-Frame.

2. *Tests' Selection*

The selected Data-Frame contains tests information like runtime, accumulative score, and statistical variance metrics like average, SD and CV (coefficient of variation) [13] as shown in Table 2. The test can be graded and ranked according to any of these measures depending upon the application. For example, for a *resolution test* list, we may want to use tests with high variance like high CV and for *regression list*, we may find aggregate score (a score in Table 2) more useful.

The user can define a threshold on the number of tests to be selected for regression as well a threshold in terms of the execution time of the tests, to generate the final regression list. We call these preferences as *resource constraints* as shown in Figure 4.

V. The Case Studies

A. *The Experimental Setup*

The objective of these case studies is to show a measurable advantage of our methodology over a regression system without it. For this purpose, we used a fault simulator called Certitude [12]. A fault simulator checks the quality of verification environment by inserting artificial faults into the design. It runs reference simulations for every test without fault activations and then runs the tests with fault activations. It compares the tests' output to report the quality of tests and verification environment.

Certitude also provides a facility called ComputeMetric to measure the quality of a verification environment. ComputeMetric uses statistical sampling of the faults and the tests, to generate metrics for the quality of verification environment. In our experiment, verification environment remains same, only the selection of tests is different. We are also only interested in controllability (activation score in Certitude) and observability (propagation score in Certitude) and not interested in detection score as detection score measure the checkers quality.

The ComputeMetric results of the whole test-suite for the complete design is our controlled experiment. The ComputeMetric results of the each of module clusters and their associated selected tests' list constitute our heuristic experiment. We also used a specific random seed in the R-based analysis for repeatable results [17].

B. *Design A*

Design A was a Queue management and submission block. It consists of 231 modules with 13811 instances. The verification environment consists of 45 tests. All 45 tests in the test suite were run once to collect code coverage as well as tests' run time. UCIS scripts [1] were run to generate a comprehensive Data-Frame for the block. After data reduction phase, we were left with 34 modules that were related to Design A feature set. The output of the Silhouette run on this reduced coverage Data-Frame is shown in Figure 3 and it suggests 8 as the best number of clusters for the design. We validated the clustering with the actual design and found it close to the true understanding of the design. That finished our data analysis and cluster generation step of the design.

For the controlled part of our experiment, we ran Certitude with 34 modules and 45 tests to generate ComputeMetric results for a full block. The results of this run form the first row of Table 3. The total number of injected faults were 12688 and faults in the sample were 300. Propagation score has a margin of error of +/- 4.54.

Table 3: Experimental results for design A

	Number of Modules	**Number of Tests**	**Activation Score**	**Propagation Score**
Full-Block	34	45	94.60%	76.02%+-4.54
Cluster-1	7	10	94.33%	74.32%+-4.44
Cluster-2	11	10	96.45%	73.55%+- 4.75
Cluster-3	7	10	94.33%	74.28%+-4.32

The last three rows of Table 3 list the results of these three experiments. The results show statistically equivalent results for cluster-based runs and Full-Block run. That is a big win as we ran only 10 tests to achieve similar results while using an analytical process without the help of expert knowledge.

C. Design B

Design B is a Serial Line Interface. It consists of 296 modules and its verification environment consist of 112 tests. All 122 tests were run to collect code coverage, followed by UCIS scripts for coverage Data-Frame generation. After the data reduction phase, the number of modules left was 92. The output of the Silhouette run for Design B is shown in Figure 5. It suggests a peak value at 3 clusters mark. This matches with our design and accurately show the three-way division of our design. After looking at various options, we found the local minima at 7 in Figure 5 as the best case for the design clustering.

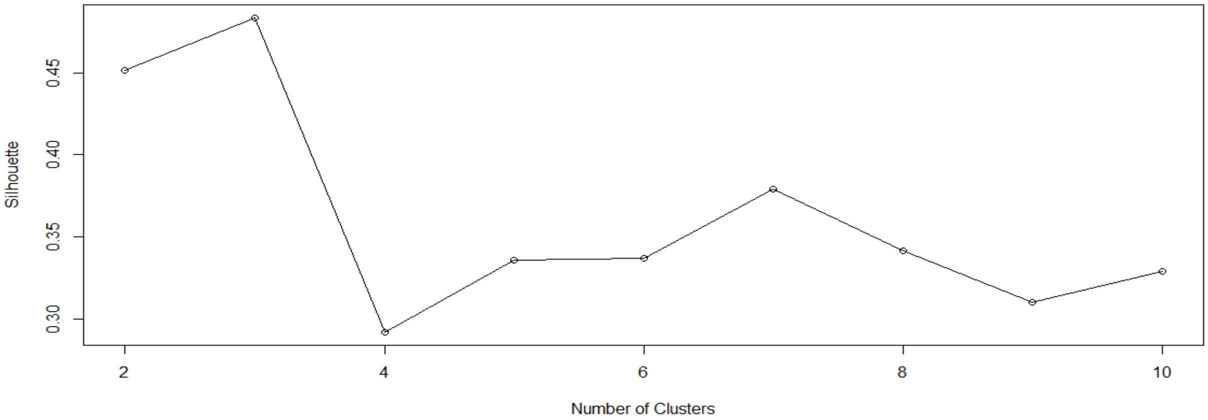

Figure 5: Silhouette for design B

For the controlled part of the experiment, we ran Certitude with 92 RTL files and 112 tests. The first row in Table 4 shows the results of this run. The total number of faults were 33050 and the faults in the sample were 300 and the margin of error was 5%. The last three rows show the results of Certitude runs for different clusters with 300 faults sample size and 5% margin of error.

Again the results show that we can get statistically equivalent validation quality using the clustering methods while running fewer tests.

Table 4: Experimental results for design B

	Number of Modules	**Number of Tests**	**Activation Score**	**Propagation Score**
Full-Block	92	112	92.31%	86.43%+-4.72
Cluster-1	18	20	97.30%	85.43%+-4.44
Cluster-2	8	20	96.45%	83.55%+-4.75
Cluster-3	11	20	95.43	82.78%+-4.56

VII. Conclusions

Regressions are a process that verifies if the proposed changes in the design have broken anything else or not. A regression suite consists of a representative set of tests. This set is manually selected by designer and verifier based on their insight of the design. Our proposed methodology takes away the guess work to create regression suites and provides quantities measures to make the right choices, dynamically with minimal interaction. We have proposed and implemented a set of heuristics based on statistical methods to implement this methodology. We have shown that our methods provide savings of the resources in terms of the simulation licenses as well as the time to run the regression because of fewer targeted tests while achieving a similar level of confidence in the verification efforts. Moreover, a bigger saving is in terms of human resources, as fewer tests are needed to be debugged.

Acknowledgements

We used “R” statistical programming language [6] [17] for the most of the analysis. We are thankful to everyone involved in the development of R and its associated packages. We are also thankful to Monica Farkash to provide us a copy of her dissertation.

References

[1] Yehia, Ahmed. “*UCIS Applications: Improving Verification Productivity, Simulation Throughput, and Cover Closure Process*” DVCON 2014, San Jose, 2014.

[2] Synopsys, Inc., “*Coverage Technology User Guide*,” Synopsys, Inc., Mountain View, CA, 2012

[3] Farkash, Monica etc., "*Mining Coverage Data for Test Set Coverage Efficiency*", DVCON 2015, Santa Clara, CA.

[4] Farkash, Monica etc., "*Regression Optimization using Hierarchical Jaccard Similarity and Machine Learning*", DAC 2013, Austin, TX.

[5] Spear, C., and Tumbush, G. 2012. *System Verilog for Verification*. Third Edition. Springer, ISBN 978-1-4614-0714-0, 608 pages, 2012. DOI= 10.1007/978-1-4614-0715-7

[6] Gareth, James etc. 2014. *An Introduction to Statistical Learning*. Fourth Edition. Springer, ISBN 978-1-4614-7138-7, 440 pages, 2014. DOI= 10.1007/978-1-4614-7138-7

[7] Sedgewick, Robert, and Wayne, Kevin. 2011. *Algorithms*. Fourth Edition. Springer, ISBN 978-0-3215-7351-3, 992 pages, 2011.

[8] Yim, Odilia and Ramdeen, Kylee T. “*Hierarchical Cluster Analysis: Comparison of Three Linkage Measures and Application to Psychological Data*”, The Quantitative Methods for Psychology, vol. 11, no.1, 2015.

[9] Finch, H. “*Comparison of Distance Measures in Cluster Analysis with Dichotomous Data*”, Journal of Data Science, 3(1), 85-100, 2005.

[10] Biswas, S. etc. “Regression Test Selection Techniques: A Survey”, Informatica 35, 289-321, 2011.

[11] Arafeen, M.J., and Do, Hyunsook. “*Test Case Prioritization Using Requirements-Based Clustering*”, IEEE Sixth International Conference on Software Testing, Verification, and Validation (ICST), 312-321, 2013. DOI = 10.1109/ICST.2013.12

[12] Synopsys, Inc., “*Certitude User Manual*,” Synopsys, Inc., Mountain View, CA, 2013

[13] Mandel, John. 1964. “The Statistical Analysis of Experimental Data” Wiley and Sons, DOI= 10.1002/bimj.19680100407

[14] Singh, Ashima, “ *Prioritizing Test Cases in Regression Testing using Fault Based Analysis*”, International Journal of Computer Science Issues, Vol. 9, Issue 6, No 1, November 2012.

[15] Synopsys, Inc., “*Coverage Technology User Guide*,” Synopsys, Inc., Mountain View, CA, 2012

[16] Foreman, J.W., “*Data Smart*”, Wiley, 2013. ISBN: 978-1-118-66146-8

[17] Matloff, Norman, “”The Art of R programming”, No Starch Press, 2011. 978-1-59327-410-8

[18] Farkash, Monica etc., "*Coverage Learned Targeted Validation for Incremental HW Changes*", DAC 2014, San Francisco, CA.