International Journal of Applied Information Systems (IJAIS) – ISSN : 2249-0868
Foundation of Computer Science FCS, New York, USA
Volume 1– No.8, April 2012 – www.ijais.org

Regression Optimizer

A Multi Coverage Criteria Test Suite Minimization Technique

Saran Prasad
Cadence Design Systems India Pvt. Ltd.

Mona Jain
Cadence Design Systems India Pvt. Ltd.

Shradha Singh
Cadence Design Systems India Pvt. Ltd.

C.Patvardhan
Department of Electrical Engineering,
Dayalbagh Educational Institute
Dayalbagh, Agra

ABSTRACT
Regression test suites are developed and maintained throughout the lifetime of the software product. For testers, it is common practice to add new testcases to the existing regression test suite, with intent to test new features in the software product or to capture any newly discovered fault. Many a times the intention is to check whether the program is sufficiently tested or not. This is done by measuring code coverage. In case if not, then additional tests are added until the test suite has achieved a specified coverage level according to a specific criterion. Due to this continuous addition of testcases, regression test suites tend to grow in size. As a result, multiple testcases may exist which may test the same feature or same set of requirements. Test Suite minimization techniques identify redundant test cases from a test suite based on some criterion. In this paper we propose a novel test suite minimization technique which identifies redundancy in a given test suite based on multiple coverage criteria for example function, function call stack, line and branch coverage of given test cases. Paper also talks about the benefits of our approach over other existing test suite minimization techniques.

General Terms
Regression test suite minimization technique.

Keywords
Software testing, regression testing, test-suite reduction, test-suite minimization.

1. INTRODUCTION
Testing is an important activity during development of any software system. In the testing process, it is common practice to write testcases to test the specific functionality of the software. Regression test suite is a test suite which contains all the test cases to test the current functionality as well as the previously working functionality of the software. In other words, it is a test suite that can be used to perform testing of the software after it is changed or modified due to addition of new functionality. Regression test suites are an important artifact of the software-development process and, and they are maintained throughout the lifetime of a software product. In order to limit the size of the test suites there are specific techniques which are known as Test suite minimization techniques. These techniques try to limit the size of the regression test suite by identifying redundant testcases based on some coverage criteria or by making use of traceability matrix i.e. testcase to requirements mapping. These techniques have two major drawbacks. One, they are incapable of handling large size test suites because of computational complexity of test case comparisons. Secondly, while minimizing a test suite, they might reduce the ability of the test suite to provide same coverage level according to a specific adequacy criterion and same set of faults. Previous studies have shown that sometimes this reduction is small, but sometimes this reduction is significant. So there has to be a technique which should be capable of doing it in a moderate manner. Our approach has several features. It is capable of operating in a live testing environment with test suites having more than a million test cases. It identifies redundant test cases without any loss of fault detection capability of reduced test suite. The reduced test suite which is obtained with our approach also provides the same coverage as the original test suite. It is also innovative in the sense that it cuts short the data set on the basis of some criteria in iterations and does comparisons and helps in reducing the computational complexity of test suite reduction problem.

2. BACKGROUND
Testers usually write new test cases to test new functionality or feature in the software, and add them to the existing test suite. As a result, these test suites grow in size with the constant addition of test cases. In many practical scenarios there is the absence of traceability matrix which maintains testcase to requirement mapping. Ideally a tester should review this matrix before adding the testcase into main test suite, but in the absence of this mapping, testers simply write and add test cases to the suite. Old test cases are not reviewed before adding a new test case which may lead to redundancy. Multiple test cases may exist in a test suite which may satisfy the same requirements. There may be multiple set of two or more testcases which may collectively satisfy same requirement. In many practical cases, additional testcases are added in the existing test suite until the test suite has achieved a specified coverage level according to a specific criterion. For example, to achieve sufficient statement coverage for a program, one would add additional test cases to the test suite until each statement in that program is executed by at least one of the test cases. This again leads to redundancy and due

to existence of redundant testcases; size of the test suite grows tremendously. And this is the point where problem arises. Large test suite size is the pain area because test suite execution can be very expensive both in terms of compute resources as well as human resource time. More human resources are needed to evaluate the failures and do root cause analysis. There is wastage of resources, time, effort and money by running too many test cases every time without gaining any code coverage or capturing bugs. Large test suite size is the pain area because test suites are run on servers; they utilize compute resources. Test cases may also need human intervention to check the output and set up other machinery. So having too many test cases to run can be very expensive. If the same problem is viewed from a developer's perspective then practically, in any software company multiple developers work on a project. These developers work on the code base having millions lines of code which are maintained using version control mechanism. It is also a common practice to check in updated code into repository after regression test suites is run. For a code base having million LOC, test suite also contains million test cases. For a larger test suite, execution time is longer and developers are required to wait for a longer time for the code to be checked in. Large test suite size is thus, major problem during test suite maintenance and there is a requirement of techniques or mechanisms by which redundant test cases can be eliminated from the test suite.

3. LITERATURE REVIEW

A lot of research work has been done in the past to determine redundant test cases in a given test suite. As per literature work, the test-suite reduction problem may be stated as follows [1, 3]:

Given: Test suite T, a set of test-case requirements r1; r2…rn that must be satisfied to provide the desired test coverage of the program, and subsets of T, T1; T2…Tn, one associated with each of the ris such that any one of the test cases tj belonging to Ti can be used to test ri.

Problem: Find a representative set of test cases from T that satisfies all ris. The ris in the foregoing statement can represent various test-case requirements, such as source code statements, decisions, definition-use associations, or specification items.

Piles of efforts have also been put into research on how to reduce the test suite size of a previously acquired test suite while maintaining its effectiveness. (Yanjun et al., 2010) recommends Greedy algorithm and with the growth of test size and suggests the usage of greedy evolution and GRE for more general by analyzing the influencing factors and performance the running time of algorithms of distinctions of 6 classical algorithms viz greedy, greedy evolution, heuristics, GRE , ILP and GA. A method for test suite minimization that uses an additional testing criterion to break the ties in the minimization process, under specific conditions, their proposed approach can also accelerate the process of minimization [4].

Another work of Scott McMaster and Atif M. Memon [5] explains that Test-suite reduction typically employs sophisticated tools such as source-code analyzers and instrumentors to reduce the number of test cases in a given test suite; the obtained subset yields equivalent coverage with respect to some criterion [3, 2, 1, 6, 7]. Emerging trends in software development present new challenges for existing reduction techniques that may limit their applicability. First, developers rely heavily on reusable components. Source code of these components is usually not available, limiting the application of source-code level instrumentors and analyzers [8]. Second, developers use a combination of programming languages to implement systems. Certain static analyzers and source-code instrumentors may not be available (or may be too complex/expensive to execute) for some of these languages. For example, some static analyses become complex in object-oriented systems due to the presence of virtual function calls. Even if analysis techniques are available for each language, combining the results from different analyses may become complex.

Another research work by S.Selvakumar [9] presents a novel approach of model-based regression test minimization (Dynamic Dependence Graph) that uses the EFSM model dependence analysis to reduce a given regression test suite. This approach claims that it has good fault detection ability when compared to that of the Static Dependence Graph approach, by considering all the interaction patterns instead of ignoring the patterns of the same dependencies between transitions which occur during the traversal of the model in an iterative manner. Their initial experience shows that this approach may significantly reduce the size of regression test suites and also improve the fault detection capability [9].

Dennis Jeffrey and Neelam Gupta [10] in their work explain that that test suite minimization with respect to a given testing criterion can significantly diminish the fault detection effectiveness (FDE) of suites. Their work presents a new approach for test suite reduction that attempts to use additional coverage information of test cases to selectively keep some additional test cases in the reduced suites that are redundant with respect to the testing criteria used for suite minimization, with the goal of improving the FDE retention of the reduced suites. We implemented our approach by modifying an existing heuristic for test suite minimization. Our experiments show that our approach can significantly improve the FDE of reduced test suites without severely affecting the extent of suite size reduction.

4. NEED FOR TEST SUITE MINIMIZATION

When test suites become too large, they can be difficult to manage and expensive to run. They need more compute resources and more execution time. Thus there is a genuine need of minimization techniques which can identify redundant test cases from a given test suite and help in reducing the size of a given test suite. Resultant test suite will be known as "Reduced Test Suite". Thus test suite minimization techniques help in effective regression testing by:

- Reducing execution time
- Effective utilization of compute resources
- Reducing resource effort
- Cost saving.

5. COMPLEXITY OF TEST SUITE MINIMIZATION PROBLEM

Our requirement of having a new test suite reduction technique does not mean that there are no existing techniques. But the major area of concern with every test suite reduction approach is that, are they capable to operate on live testing environment or real world software programs? The main reason behind this could be the computational complexity of the test suite reduction problem.

The process of identifying redundancy among test cases is not so trivial, because of huge regression suite size. In a typical live testing environment a test suite may contain hundreds of test cases - sometimes even more than 10,000 test cases or even a million. In order to identify redundant test cases, each test case needs to be compared with every other test case which makes this a problem of $O(\mathbf{n^2})$ time complexity i.e. time required to perform comparisons is directly proportional to the square of the size of the number of test cases. Let's consider a typical test suite for a live product to be of 1 million test cases then $(\mathbf{1\ million})^{\mathbf{2}}$ comparisons are challenging.

This is further complicated by the fact that a test case to test case comparison is not a simple operation but could have multiple comparison criterions. For example test cases may be compared on the basis of statement coverage, function coverage, function call stack, branch or decision etc. With a typical live product containing millions of LOC, the line coverage comparison would be a time consuming operation. By line coverage comparison we specifically mean comparison of actual line numbers rather than comparison of percentage line numbers. These comparisons are definitely too time consuming that may limit the usability of the solution.

Hence to solve this we need to identify a test suite minimization technique which can reduce the complexity of the problem so that redundant test cases can be found within a reasonable amount of time with an acceptable degree of accuracy. Test Suite reduction should happen in such a way that the reduced test suite should be able to provide same coverage and capture same number of faults as the original test suite. The technique should be capable to operate on real testing environment with acceptable performance.

6. RESEARCH QUESTIONS

Few questions which needed a thorough research before finalizing the reduction approach are being explained here:

6.1 Definition of "test case redundancy" as per literature

The first step was naturally to define test case redundancy. When do we regard two test cases as being similar and by extension one of them as being redundant? The question is not trivial as there have been several alternative approaches in the literature. As per prior research works [11, 12, 13, 14, 15] "Test Case Redundancy" mean removing test cases from a test suite in such a way that reduced test suite should:

- Provide the same coverage (in terms of some coverage criteria like function flow, statement, and branch etc) of the software as the original test suite.
- Satisfy the same requirements as the original test suite i.e. effectiveness of the test suite should not be affected.
- Be capable to reveal same set of faults as the original test suite.

6.2 Our definition of test case redundancy

We have defined redundancy as "Test cases TC1 and TC2 are said to be redundant if they have same functional flow, same line coverage and same branch coverage. Functional flow or sequence of functions call should be same starting from the first function".

With this definition in place we are making sure that those test cases which will come up as redundant as per our approach will have same functional flow i.e. they will call same set of functions that too in same sequence, same line coverage and same branch coverage. Testcases whose execution generates the same functional flow shows test cases in a suite which are testing same features. So the probability that they are redundant is higher. Along with functional flow we are also comparing statement coverage and by same statement coverage we mean comparison of actual line numbers which got covered during test case execution. Although statement coverage is a good coverage metric but it is insensitive to control structures and does not adequately take into account statements which involve branching and decision-making i.e. the control structures. Because of this reason we have also considered the criteria of same branch coverage. Same branch coverage ensures same path during program execution.

6.3 Our definition of "Similar Testcases"

We have opted two definitions of "similar test cases"

- Test cases TC1 and TC2 are said to be similar if they call same set of functions starting from the first function OR
- Test case TC1 is similar to TC2 if functions called by TC1 is a proper subset of functions called by TC2, staring from the first function.

The similar test cases will get filtered at first step of our algorithm and that is the reason for taking them into consideration.

7. TEST SUITE MINIMIZATION ALGORITHM

With our invention we are presenting a generic test suite minimization approach for determining redundant test cases in a given test suite based on multiple coverage criteria. Our approach has the capability to operate on a real world testing environment. It can identify redundant test cases from test suites having multimillion test cases without any loss of fault detection capability of reduced test suite that too in adequate amount of time. It also ensures the same code coverage of the reduced test suite as provided by the original test suite.

As it has already been explained in the problem statement that the test suite reduction is a computationally expensive operation because of number of comparisons among the test cases, our approach tries to resolve it in steps. Rather than comparing each test case with another in a single iteration, it

forms the cluster of similar test cases at first step and then applies three coverage reduction criteria to identify redundant ones within each cluster. In this way it breaks the bigger problem into smaller chunks. Thus our algorithm identifies redundant test cases on the basis of following reduction criterion:

- Reduction Step 1: "Test case to Function" Binary Matrix Generation". This is done on the basis of same set of functions called. The output of this step is given as input to next step.
- Reduction Step 2: Use of Hierarchical Clustering to form clusters of similar test cases in the given test suite
- Reduction Step 3: Compare functional flow (sequence of functions call) within the clusters of similar test cases
- Reduction Step 4: Compare line coverage of test cases obtained in step 3.
- Reduction Step 5: Compare branch coverage of test cases obtained in step 4

Following section describes the reduction steps of our approach along with the reasoning of how these reduction steps helps in overcoming the limitations which persist in the existing techniques.

7.1 Step1: Generate a binary matrix having "Test case to Function" mapping

By "Testcase to Function Mapping" we mean "A relationship between testcase and functions in terms of all the functions that have been called during test case execution."
Using this relationship a binary matrix is deduced with test cases in the rows and functions in the column. A '1' entry in the matrix indicates a call to the corresponding function whereas a '0' indicates no function call.
In order to deduce this mapping, function coverage data is needed. Function coverage data can be collected either from Purecov or GCOV. There are other coverage analyzers also but our approach supports data of these two coverage analyzers. It can further be enhanced to support data from other coverage tools.
We have implemented our approach in the form of a java program. Our main program contains two parsers one for Purecov data and another for GCOV data. These are being explained in detail in next section.

7.1.1 Purecov Data Parser:

Output of Purecov coverage analyzer [16] is .pcv files i.e. Purecov generates coverage data in the form of .pcv files. Users are supposed to enable Purecov and generate instrumented binaries of source program. When testcases are run on these instrumented binaries then coverage results are generated and written in .pcv files. Purecov has the ability to generate individual .pcv file for every test case or a consolidated .pcv file for a given set of test cases. Our approach requires individual pcv files for every test case.

Detailed Steps:

- Our main program takes Purecov's .pcv file directory path as input.
- This directory contains all <testcasename.pcv> files generated by Purecov.
- Purecov parser converts each .pcv file into an export format (human readable format) by using Purecov's "–export" switch.
- It then reads each export file and picks all lines which starts from –fu option with #calls >0, "fu" provides information about a single function within a file.
- It then generates a relationship and saves test case name along with function name.
- Finally it generates binary matrix out of this relationship.

7.1.2 GCOV Data Parser:

GCOV [17] is an open source software which is freely available. When test cases are run on GCOV instrumented binaries, then .gcov files are created for every source file. These gcov files are in the human readable format. Users are supposed to save the .gcov files for individual test cases in different directories.

- Our main program takes path of above mentioned directories having .gcov files as input.
- It then starts reading main .gcov file, parses each line and picks function name from the line which starts with function keyword.

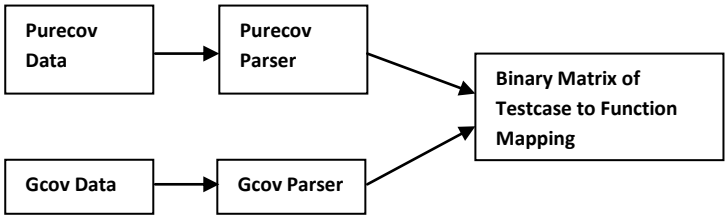

Fig.1. Reduction Step 1 Reading of Coverage Data and Its Conversion into Testcase to Function Mapping

The output of step 1 is a binary matrix of test case to function mapping. This matrix is provided as input to program which implements Hierarchical Clustering algorithm.

7.2 Step 2: Use of Hierarchical Clustering [18] to form clusters of similar test cases in the given test suite

Test suite reduction is not a trivial problem to solve because of number of comparisons among test cases. If total test cases are 'n', then each test case needs to be compared with every other test case to figure out the redundant one, hence there will be n^2 comparisons. Thus complexity becomes O (n^2). Let's consider a typical test suite for a live product to be of 1 million test cases then $(1\text{ million})^2$ comparisons are challenging. Because of the said problem our approach works in steps. Rather than taking the complete regression test suite in single iteration and comparing each test case with another, it forms the cluster of similar test cases at first step and then applies some filters to identify redundant ones within each

cluster. To break the bigger problem into smaller chunks, hierarchical clustering algorithm has been implemented. This algorithm makes use of hamming distance [19] concept to form clusters of similar test cases. For the implementation of this step:

- A binary matrix of test case to function mapping is required as input. This will be obtained from Step1.
- Testcases are grouped on the basis of functions being called by testcase (using a threshold i.e. the maximum number of allowable function calls mismatch).
- Specifically for this work, clustering is done taking threshold zero i.e. only those testcases are added to one cluster if all the function calls are identical.
- Subset filtration is also done at this step which gives the set of testcases with function calls being subset to another test case.

Detailed Steps:

- Take binary matrix which is generated in step 1with test cases as rows and functions as columns.
- Take a datastructure for saving final list of testcase indexes, for e.g. create an ArrayList of ArraList of Integer (say cluster).
- If there are n rows then for row i set i=0 and j=1
- Pick the $(j)^{th}$ row if j is not 0 (row 0 mean that particular row is already added in the cluster. A Boolean array is maintained with a flag value for every testcase)

Note: Hamming Distance Calculation Condition :- No point in calculating Hamming distance between rows i and j if they are zero. A row equals to 0 means all the values within the row is 0 and that it has already been added in the cluster.

- Take a temp data structure. Create a temp arraylist
- Put i in the ArrayList
    i. Calculate hamming distance between i and j
    ii. If distance is smaller than threshold , Merge ith and jth : For all places in the jth row where the value is 1 make the value at that place in the ith row equal to 1.
    iii. Set the jth row zero (in Boolean array this will make sure that the particular row has already been added in the cluster)
    iv. Put j (testcase index) in the ArrayList temp () and set j=1 else increment j by 1 i.e. j=j+1
    v. Repeat step i till j<n
- Once all the iterations from 6i to 6iv are done, all the test case indexes which belong to one cluster will be saved in "temp" arraylist. Put temp in the ArraList of ArrayList of Integer cluster
- Increment i by 1 i.e. repeat all the steps for all the remaining rows
- Go step 4 while i < n
- ArraList of ArrayList of Integer i.e. cluster will have all clusters of similar test cases.

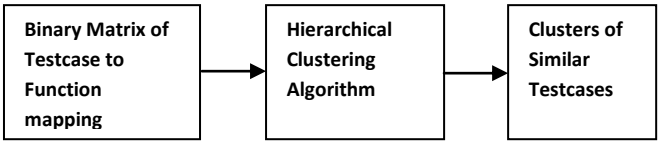

Fig.2. Reduction Step 2 Formation of Clusters of Similar Test

7.2.1 Merit of using hierarchical clustering:

The advantage of implementing hierarchical clustering algorithm is that it outputs clusters of similar test cases thereby helps in reducing the number of comparisons and enhances the computational efficiency of the reduction process. Once clusters of similar testcases are formed then test cases within the individual clusters are compared based on remaining coverage criteria e.g. function call stacks, line and branch coverage. The implementation of hierarchical clustering to form clusters of similar test cases is an innovative way to curtail the computational effort of test case comparisons.

7.3 Step 3: Compare functional flow (sequence of functions call) within the clusters of similar test cases

7.3.1 Call Stack

In a stack-based architecture, a thread in a running program has a call stack as a part of its state. Informally, the call stack is simply the series of currently active calls. Function activation records are pushed onto the call stack when they are called and popped when they return [5].

Next step of our reduction process is to compare function call sequences among the clusters of similar test cases. Coverage tools usually do not provide function call sequences. They only provide function coverage. Purecov and GCOV also do not generate function sequences. There are profilers and debuggers available which can be used to determine function call sequences during test case execution. In our case we have made use of a debugger called gdb (GNU Debugger) [20] to determine function sequences during test case execution. Few wrappers using java programming language have been written which automates the debugger running process and parses the output of the debugger and writes the function sequence neatly in output files which are further used for comparisons.

Our method is very smooth and quick which determines recursive functional flow during test case execution. Normally a debugger provides function call trace if the program execution is broken and not every time. But our approach makes use of debugger to print function call stack during normal execution as well and helps in capturing the function call sequences.

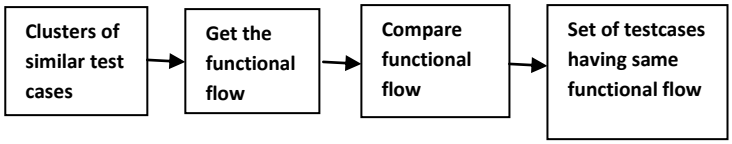

Fig.3. Reduction Step 3 Comparison of Functional Flow

7.4 Step 4: Compare line coverage of test cases obtained in step 3

Testcases obtained in step 3 are further compared on the basis of line coverage. By line coverage comparison we mean comparison of actual line numbers of every function of program source code which got hit during test case execution. The % line coverage that should match depends upon a threshold value which is configurable. Default threshold value is 0 which shows exact match.

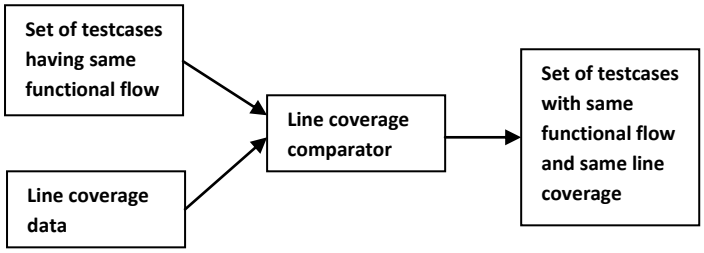

Fig.4. Reduction Step 4 Comparison of Line Coverage

By comparing actual line numbers for every function our approach makes sure that those two test cases are hitting same set of statements thereby increasing the chances of their being redundant.

7.5 Step 5: branch coverage of test cases obtained in step 4

Testcases obtained in step 4 are further compared on the basis of branch coverage. By branch coverage comparison we mean comparison of all branches or paths in the program which got taken during test case execution.

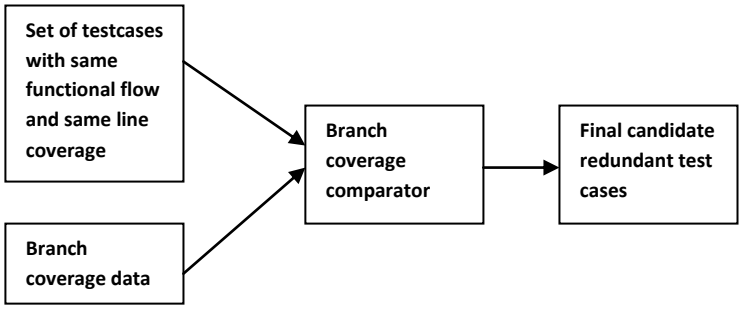

Fig.5. Reduction Step 5 Comparison of Branch Coverage

A branch is the outcome of a decision, it measures which decision outcomes have been tested. Branch coverage criteria give a more in-depth view of the program source code paths than simple statement coverage. That is why we have considered it at the final step of reduction.

By comparing branch coverage of two test cases our approach makes sure that those two test cases are covering the same paths in the program flow thereby increasing the chances of their being redundant.

8. EXPERIMENTAL STUDY

As already mentioned earlier we have implemented the reduction algorithms by using java programming language and ran two experiments to evaluate our test suite reduction technique. Our experiment was performed on a C Source with 2000 LOCs. Total number of test cases were 300 (randomly generated inputs using model, in which 10 intentionally were the copy and 10 were the superset of 20 text cases). Total time taken to perform the experiment was < 30 seconds on 8 GB intel core i7.

9. CONCLUSION

We have developed a new Test Suite Reduction approach which is innovative in the sense that it cuts short the data set on the basis of some criteria in iterations and does comparisons and helps in reducing the computational complexity of test suite reduction problem. It ensures to reduce the real world regression test suites on the basis of multiple coverage criteria namely function, function call stack, line and branch coverage in a moderate amount of time without any loss of percentage code coverage and fault detection capability of reduced suite. The only limitation which persists is the dependency of this approach on coverage data.

10. REFERENCES

[1] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong. Empirical studies of test-suite reduction. Journal of Software Testing, Verification, and Reliability, V. 12, no. 4, December, 2002.

[2] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. Proceedings of the International Conference on Software Maintenance, pages 34-43, November 1998.

[3] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. ACM Transactions on Software Engineering and Methodology (TOSEM) July 1993 Volume 2 Issue 3.

[4] Selvakumar Subramanian and Ramaraj Natarajan. A Tool for Generation and Minimization of Test Suite by Mutant Gene Algorithm. Journal of Computer Science 7 (10): 1581-1589, 2011. ISSN 1549-3636, © 2011 Science Publications.

[5] Scott McMaster and Atif M. Memon. Call Stack Coverage for Test Suite Reduction. Department of Computer Science, University of Maryland, College Park, MD 20742.

[6] W. Eric Wong, Joseph R. Horgan, Saul London, Aditya P.Mathur. Effect of test set minimization on fault detection effectiveness. Proceedings of the 17th International Conference on Software Engineering, p.41-50, 1995, Seattle, Washington, United States.

[7] W. E. Wong, J.R. Horgan, A. P. Mathur, and A. Pasquini. Test set size minimization and fault detection effectiveness: A case study in a space application. Proceedings of the 21st Annual International Computer

Software and Applications Conference, pages 522-528, August 1997.

[8] M. Morisio, C. B. Seaman, A. T. Parra, V. R. Basili, S. E.Kraft and S. E. Condon. Investigating and improving a COTS-based software development. Proceedings of the 22nd international conference on Software engineering, pages 32-41, Limerick, Ireland, 2000.

[9] S. Selvakumar, N. Ramaraj. Regression Test Suite Minimization Using Dynamic Interaction Patterns with Improved FDE. European Journal of Scientific Research ISSN 1450-216X Vol.49 No.3 (2011), pp.332 353 © EuroJournals Publishing, Inc. 2011. http://www.eurojournals.com/ejsr.htm

[10] Dennis Jeffrey and Neelam Gupta. Improving Fault Detection Capability by Selectively Retaining Test Cases during Test Suite Reduction. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 33, NO. 2, FEBRUARY 2007

[11] Gordon Fraser and Franz Wotawa. Redundancy Based Test-Suite Reduction, Institute for Software Technology Graz University of Technology Inffeldgasse 16b/2 A-8010 Graz, Austria.

[12] Hiroshi Inamura Ajay Chander Dinakar Dhurjati. Method For Test Suite Reduction Through System Call Coverage Criterion United States Patent Application Publication. Publication Number US 2009/0070746 A1. Pub. Date: Mar 12, 2009.

[13] Negar Koochakzadeh and Vahid Garousi. A Tester-Assisted Methodology for Test Redundancy Detection. Hindawi Publishing Corporation. Advances in Software Engineering Volume 2010, Article ID 932686, 13 pages DOI:10.1155/2010/932686

[14] Shin Yoo and Mark Harman. Pareto Efficient Multi-Objective Test Case Selection. King's College London Strand, London WC2R 2LS, UK {Shin.Yoo,Mark.Harman}@kcl.ac.uk

[15] Emanuela G. Cartaxo, Francisco G. O. Neto, Patr´ıcia D. L. Machado. Automated Test Case Selection Based on a Similarity Function

[16] Rational Purecov Tool: http://www.ing.iac.es/~docs/external/purify/purecov-4_1.pdf

[17] GCOV: A Test Coverage Program http://gcc.gnu.org/onlinedocs/gcc-3.0/gcc_8.html

[18] Hierarchical Clustering: http://www.stat.cmu.edu/~cshalizi/350/lectures/08/lecture-08.pdf; http://en.wikipedia.org/wiki/Hierarchical_clustering

[19] Hamming Distance: http://en.wikipedia.org/wiki/Hamming_distance.

[20] GDB: GNU Debugger http://www.gnu.org/s/gdb/