Comprehensive Testing Tool for Automatic Test Suite Generation, Prioritization and Testing of Object Oriented Software Products

C. Prakasa Rao *prakashrao.pec@gmail.com*
Research Scholar
Sri Venkateswara University,
Tirupathi, Chittoor(Dt) AP, India

P. Govindarajulu *pgovindarajulu@yahoo.com*
Retd Professor, Dept. of Computer Science,
Sri Venkateswara University,
Tirupathi, Chittoor (Dt) AP, India

Abstract

Testing has been an essential part of software development life cycle. Automatic test case and test data generation has attracted many researchers in the recent past. Test suite generation is the concept given importance which considers multiple objectives in mind and ensures core coverage. The test cases thus generated can have dependencies such as open dependencies and closed dependencies. When there are dependencies, it is obvious that the order of execution of test cases can have impact on the percentage of flaws detected in the software under test. Therefore test case prioritization is another important research area that complements automatic test suite generation in objects oriented systems. Prior researches on test case prioritization focused on dependency structures. However, in this paper, we automate the extraction of dependency structures. We proposed a methodology that takes care of automatic test suite generation and test case prioritization for effective testing of object oriented software. We built a tool to demonstrate the proof of concept. The empirical study with 20 case studies revealed that the proposed tool and underlying methods can have significant impact on the software industry and associated clientele.

Keywords: Software Engineering, Testing, Test Case Prioritization, Dependency Structures, Branch Coverage.

1. INTRODUCTION

Software testing can improve quality of Software Under Test (SUT). Unit testing [3] is one of the important tests made. However, generating test cases automatically has been around in the research circles. Nevertheless, it is a complex and challenging task [5]. Other approaches to test case generation are based on search [2], [4]. The recent focus is on the high coverage while generating test cases automatically. To solve this problem automatic test suite generation came into existence. Test suite can reduce number of test cases as it can provide representative test cases with high coverage. Test suite generation targets many goals at a time and generates optimal test suites. A problem with considering a single goal is that it cannot guarantee of high coverage and may produce infeasible ones. In this paper we proposed a mechanism for automatic test suite generation based on Genetic Algorithm (GA). The test suites thus generated are used for mutation testing which is widely used technique [2], [4]. GA is used to have an evolutionary approach to generate test suites that can best represent high code coverage.

Test suites are used to detect faults. However, there are some test cases that depend on others. This will provide way to further research on the dependencies and improving detection of fault rates. In order to unearth hidden faults, it is essential to consider dependencies. To achieve this dependency structures are to be discovered. Haidry and Miller [8] focused on the test case prioritization but they did not discover dependency structures automatically. In this paper, we focus on the automatic discovery of dependencies. Figure 1 shows both open and closed dependencies.

$$\begin{tikzcd}
 & & I_1 \arrow[dl] \arrow[dr] & & & I_2 \arrow[d] & \\
 & D_3 \arrow[dl] \arrow[dr] & & D_4 \arrow[dr] & & D_5 \arrow[dl] \arrow[d] \arrow[dr] & \\
D_6 & & D_7 & & D_8 & D_9 & D_{10}
\end{tikzcd}$$

FIGURE 1 : Sample Dependency Structure [1].

As can be seen in Figure 1, the root nodes are I1 and I2 that are independent on other nodes. Other nodes have dependencies. Dependencies are classified into two categories. They are open and closed dependencies. The D6 is indirectly dependent on I1 and directly dependent on D3. Therefore there is closed dependency on D3 and open dependency on I1 (no immediate dependency). When both kinds of dependencies are considered, the quality of test case prioritization improves. DSP volume and DSP height are the two measures used in test case prioritization. DSP height indicates depth while DSP volume indicates the count of dependencies. In [6] different measure is used known as Average Percentage of Faults Detected (APFD). APFD value is directly proportional to fault detection rate.

In the literature many kinds of test case prioritization techniques were found. They are knowledge based [18], history based [11] and model based [25]. More on these techniques is covered in Related Works section of this paper. Our main contribution in this paper is the development of a comprehensive tool that supports test suite generation, automatic discovery of dependency structures and test case prioritization. The tool employs the methods we proposed in our earlier work to demonstrate the proof concept. The remainder of the paper is structured as follows. Section 2 reviews literature related to test case generation and test case prioritization. Section 3 focuses on comprehensive testing tool for automatic test suite generation and prioritization. Section 4 presents experimental results while section 7 concludes the paper besides providing future work.

2. RELATED WORKS

This section provides review of literature pertaining to whole test suite generation, automatic extraction of dependency structures and prioritization of test cases for improving bug detection rate.

2.1 Whole Test Suite Generation

Many researchers tried to generate test suites that can reduce number of test cases as they provide representative test cases. Moreover test suits can also provide maximum coverage. Towards achieving this, Rodolph [13] focused on genetic algorithms while Arcuri and Briand [26] threw light into adapting random testing. Bacteriologic algorithm [27], hybrid approach with static and dynamic methods [28], Directed Automated Random Testing (DART) [29] are the other approaches found in the literature. PathFinder [30], evolutionary programming [21], integrated evolutionary programming [31], Mock classes and test cases [32], exploring length of lest cases for code coverage [33], JCrasher [34] are other approaches found for testing Java applications. Malburg and Fraser [2] combined both search-based and constraint based approaches to test software. Mutation-driven generation [1] and parameter tuning [3] and mutation testing [4], [7] were other useful researches found.

2.2 Automatic Generation of Dependency Structures

Ryser and Glinz [9] focused on dependency charts for reflecting dependencies among various scenarios for improving testing. They used natural language processing techniques for mapping system requirements and descriptions. State charts are used to refine the representations. This will have clear picture of dependency structures based on which test cases are extracted while making use of dependency charts. With respect to dependencies Kim and Bae [10] opine that there are two features namely accidental and essential in a system. The former depends on the latter thus producing different levels in the system. These researchers employed an approach to align the features pertain to the two classes of features based on the dependencies. The accidental features were found to be dynamic and changed often while the essential features do not change frequently. However, they only focused on the dependency structures and did not consider test case prioritisation.

2.3 Test Case Prioritization

This section provides test case prioritization categories that are close to our approach in this paper. They are history-based, knowledge-based and model-based approaches.

2.3.1 History-Based Prioritization

Rothermel et al. [11] used execution information for improving rate of fault detection. Their techniques are based on code coverage, probability of fault finding, untested code and so on. A greedy algorithm is used to know tests that have highest coverage. Li et al. [12] focused on many non-greedy algorithms for test case prioritization. These algorithms include genetic algorithms, hill climbing, and so on. They achieved high fault detection rate. Jeffrey and Gupta [14] also depended on the paths covered while running the code. They found the affecting paths and considered them as relevant. The more relevancy demands more importance. Wong et al. [15] focused on the changes in the source code from the previous runs in order to complete test case prioritization with respect to regression testing. Li [16] followed an approach known as code-coverage-based technique for test case prioritization. He could find the relationships among code blocks and prioritize test cases based on the code coverage concept. He also coupled the solution with symbolic execution for more accuracy. Later on in [17] the technique was improved further which throws light into test suite generation and also priority in the execution of test cases.

2.3.2 Knowledge-Based Prioritization

Ma and Zhao [18] focused on program structure analysis in order to find faults in the software. They used ranking given to modules that indicated testing importance of the module. This is based on the importance of module and fault proneness. Their approach improved fault detection rate when compared with random prioritization. Krishna moorthi and Sahaaya [19] used software requirements as basis for test case prioritization. They used fault impact, traceability, completeness, complexity, implementation, and requirements change and customer priority for automatic test case prioritization. In [20] research was done in the similar lines. In [21] case-based ranking approach was followed. It is a machine learning based mechanism employed for test case prioritization. It depends on the user knowledge pertaining to program. Zhang et al. [28] used data mining approach known as clustering for test case prioritization. They rank the test cases based on the clustering results. In [22] combinatorial interaction testing is employed to define prioritization. Based on the interactions and benefit analysis test cases are prioritized. In [23], UML models are used to generate test case priorities based on the underlying interaction levels. Adaptive Random Testing [24] focused on utilization of knowledge that has been categorized to prioritize test cases. It was proved to be inexpensive when compared with other techniques.

2.3.3 Model-Based Prioritization

Kundu et al. [25] explored test case generation and then prioritization for object oriented programs. Their approach is to take UML sequence diagrams as input and generate sequence graphs and prioritize the test cases. It makes use of weights concept such as message weight and edge weight. Based on the weights certain metrics are used. They include sum of message weights of a path, sum of message weights for all edges and weighted average edge weights. The results reveal the interactions between the components of the system. This will also provide a kind of trace that can be used to prioritize test cases. In this paper we proposed mechanisms for whole test suite generation and test case prioritization besides testing them using a tool that we built in Java programming language.

3. COMPREHENSIVE TESTING TOOL FOR AUTOMATIC TEST SUITE GENERATION, PRIORITIZATION

We built a tool that facilitates automatic testing of object oriented applications. The tool includes features such as automatic test suite generation, automatic discovery of dependency structures and prioritization of test cases. The following sub sections provide insights of the underlying features of the tool. This work of ours is an extension to our previous work [45], [46], and [47].

3.1 Automatic Test Suite Generation

This section provides details of our approach for automatic test suite generation. It is a search-based approach to generate whole test suite based on testing goals. It also maximizes mutation score. For test data derivation for search based testing, GAs are widely used. In fact, GAs are meta-heuristic in nature and are very popular for search based testing. GA makes use of initial population and uses operators like mutation and crossover to have next generation. Thus GA is considered to be an evolutionary algorithm that can provide optimal solutions. In this paper, we consider the generation of test suite for object oriented source code developed in Java. Let t be the test case t= {s_1, s_2, s_3, …, s_l) with certain length *l*. In the traditional approaches a single goal is used to generate test suites. However, in this paper, we use multiple goals at a time and generate whole test suite that is representative for all tests with high coverage. Branch coverage is used to guide test suite generation process. To this effect, fitness function is used. The methodology is shown in Figure 3.

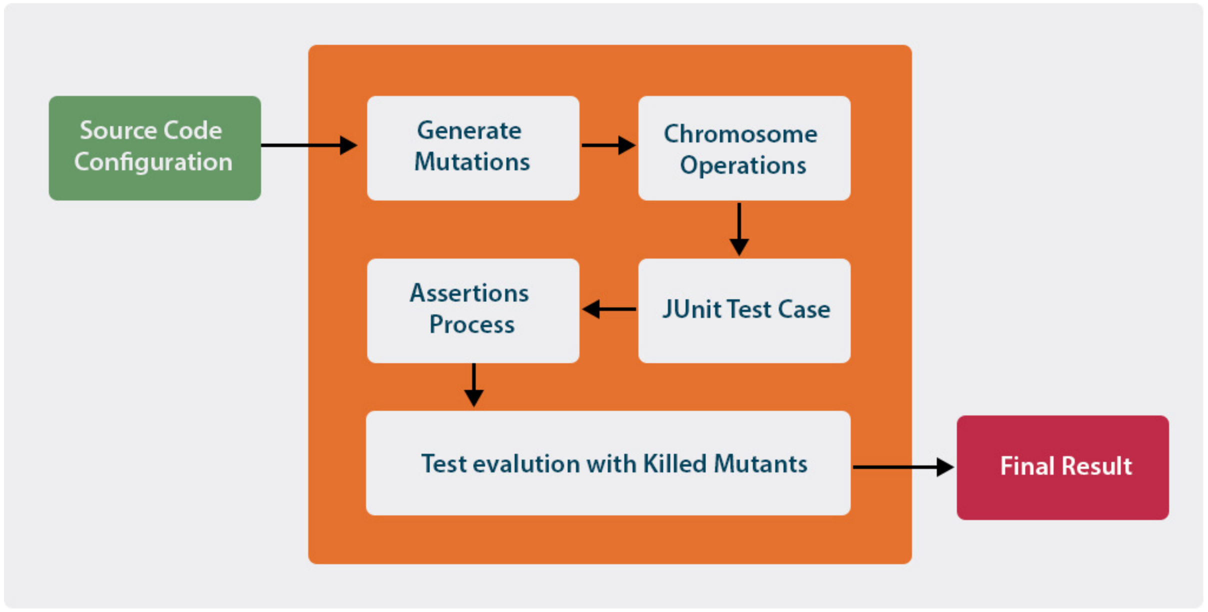


FIGURE 2: Methodology for proposed whole test suite generation.

As shown in Figure 2, the proposed solution starts with source code configuration. The framework takes Java source code and generates mutations in iterative fashion. It also involves in chromosome operations. JUnit is used for API required to generate test cases. JUnit has its assertions process. The generated test cases are evaluated and some of the mutants are eliminated or killed. The mutation operators are used at Java byte code level. The final test suite is generated. The GA operators are used for performing various operations. The operators at Java code level are used to have different mutants generated.

3.2 Automatic Discovery of Dependency Structures and Test Case Prioritization

There are many approaches found in the literature for test case prioritization. They include adaptive random test case prioritization [39], program structure analysis [44], combinatorial interaction testing [43], use case based ranked method [41], Prioritization of Requirements for Test (PORT) [37], cost prioritization [38], function level and statement level techniques [42], regression testing [36], test costs and severity of faults [42], and scenario-based testing [40]. The approach we followed in this paper is close to that of [8] with difference in the approach for test case prioritization. Overview of our architecture for test case prioritization is shown in Figure 3.

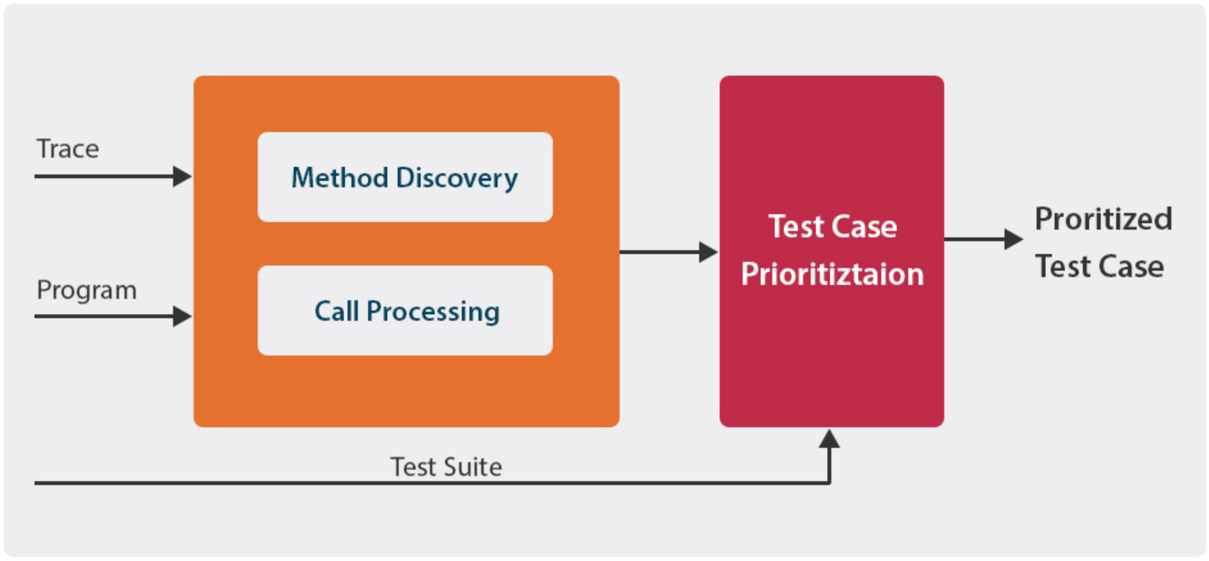


FIGURE 3: Proposed methodology for test case prioritization.

As shown in Figure 2, program execution traces and program given as input. The method discovery is the process of finding methods in the program using reflection API. The list of methods is further used to prioritize test cases. The call processing is the process of identifying the method calls in the traces and determining the order in which methods are to be tested in order to have high level of fault detection rate. The meta data associated with calls can help in making well informed decisions. The test case prioritization module is responsible to actually prioritize test cases that are supplied to it in the form of test suite and provide results. The results contain test cases that have been prioritized. The priority when used will improve the percentage of fault detection.

3.3 Test Case Prioritization Algorithm

This sub section provides the pseudo code for the test case prioritization. This code is implemented in our tool in order to demonstrate the proof of concept.

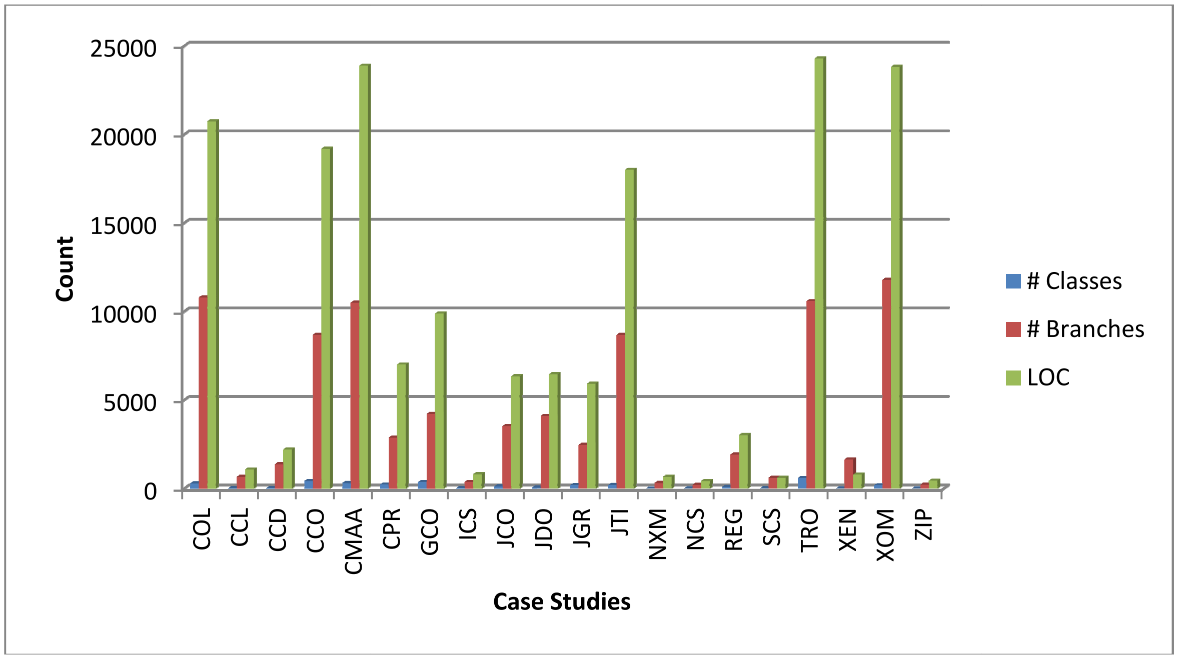
Pseudo Code for Test Case Prioritization
Input : Execution Traces(***ET***), Program(***P***) and Test Suite(***TS***)
Output : Prioritized Test cases(***PT***)

```
Step 1 : Initialize a vector (M)
Step 2 : Initialize a another vector (MM)
Step 3 : Discover methods from P and populate M
Step 4 : for each method m in M
            step 4.1 : scan TS
            step 4.2 : associate meta data with calls
            step 4.3 : add method m to vector MM
Step 5 : end for
Step 6 : for each mm in MM
            step 6.1 : analyze TS
            step 6.2 : correlate with mm
            step 6.3 : add corresponding m to PT
Step 7 : end for
Step 8 : return PT
```

As seen, the Pseudo Code for implementing test case prioritization. The algorithm takes execution traces, program and test suite. After processing, it generates prioritised test cases. First of all it discovers methods and makes a collection object to hold it. For each method test suite is verified and the meta data is associated with calls. This is an iterative process which ends with a vector or collection containing methods associated. Then the test case prioritization considers the dependencies and test suites available to have a list of test cases in the order of priority. Prioritized test cases are the final result of the algorithm.

4 . EXPERIMENTAL RESULTS

The tool we built was tested with 20 real time applications. The details of the applications in terms of number of classes, number of branches and LOC are as shown in Figure 4. The tool demonstrates the proof of concept and discovers dependency structures from given program. The tool can distinguish between open and closed dependencies as described earlier in this paper. The tool supports both automatic test suite generation and test case prioritization with automated discovery of dependency structures.



Case Study	COL	CCL	CCD	CCO	CMAA	CPR	GCO	ICS	JCO	JDO	JGR	JTI	NXM	NCS	REG	SCS	TRO	XEN	XOM	ZIP
# Classes	298	15	22	421	306	231	370	29	118	61	193	199	1	11	91	12	591	7	185	4
# Branches	10795	662	1369	8683	10503	2874	4214	373	3531	4098	2467	8681	310	209	1922	607	10585	1645	11794	219
LOC	20741	1078	2205	19190	23881	7008	9886	809	6339	6452	5924	18003	661	421	3020	606	24297	788	23814	441

FIGURE 4: Details of Case Studies.

As can be seen in Figure 4 the case studies considered for experiments include Colt, Commons CLI, Commons Codec, Commons Collections, Commons Math, Commons Primitives, Google Collections, Industrial Case Study, Java Collections, JDom, JGraphT, Joda Time, NanoXML, Numerical Case Study, Java Regular Expressions, String Case Study, GNU Trove, Xmlenc, XML Object Model and Java ZIP Utils. The experiments are made with proposed approach and the similar approach EvoSuite besides a single objective approach.

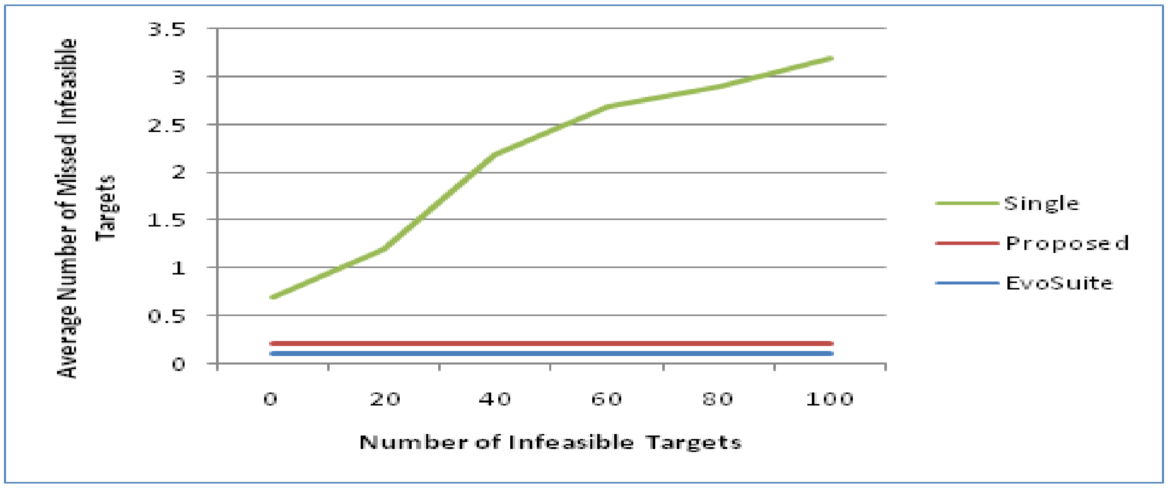


FIGURE 5: Comparison of proposed approach with other approaches.

As shown in Figure 5, it is evident that the proposed approach is compared with other approaches. The proposed approach is better than the single objective approach. This is because generating test suite with multiple objectives in mind can reduce number of test cases besides being able to cover all branches. With respect to average branch coverage Evosuite and the proposed method outperform single branch approach.

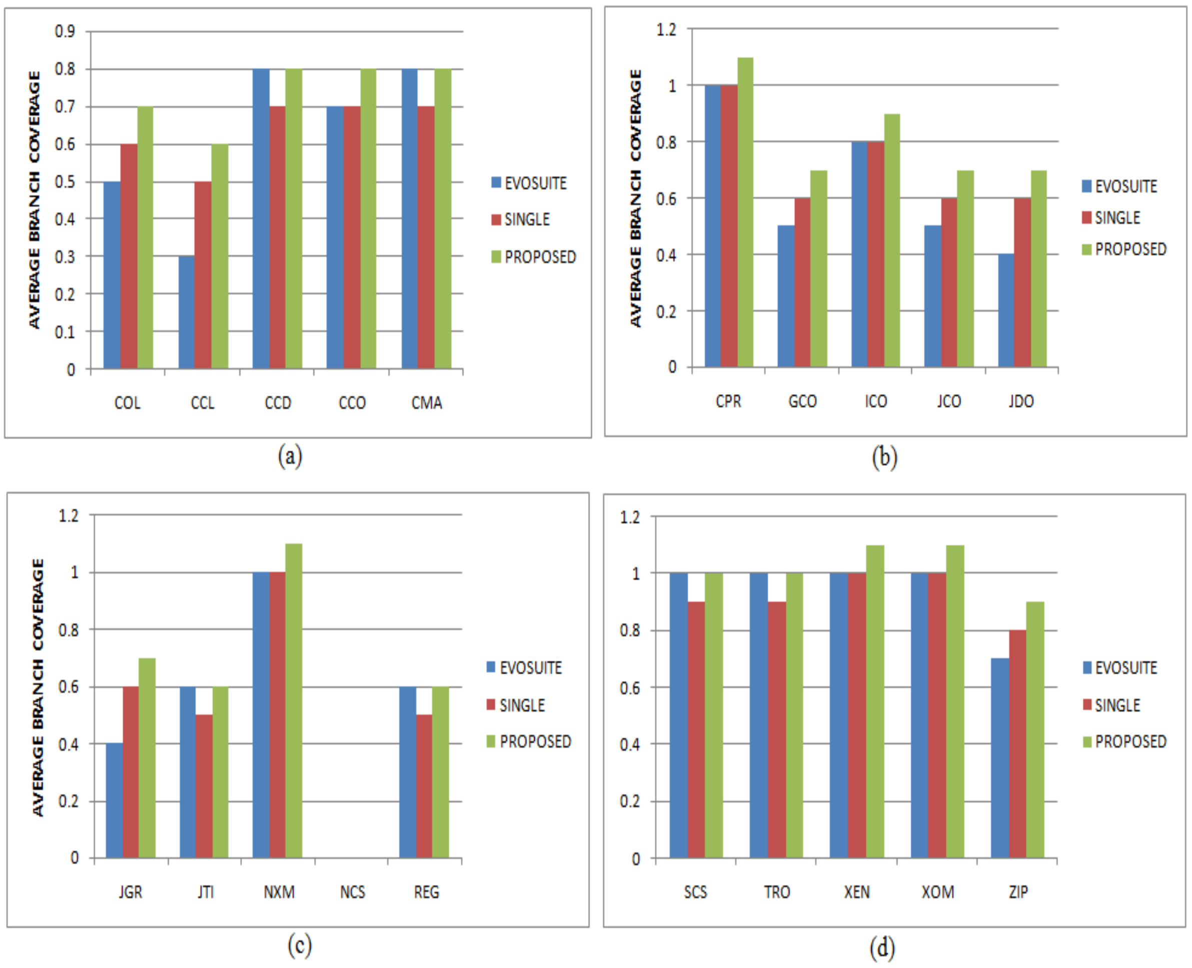


FIGURE 6: Average Branch Coverage comparison for all case studies

As shown in Figure 6, the average branch coverage of the proposed system and Evosuite perform better than the single branch strategy. The reason behind this is that when test suite is generating with multiple objectives in mind, the generated test cases will be less besides being able to cover all possible branches. In horizontal axis case studies and in vertical axis the average branch coverage is presented.

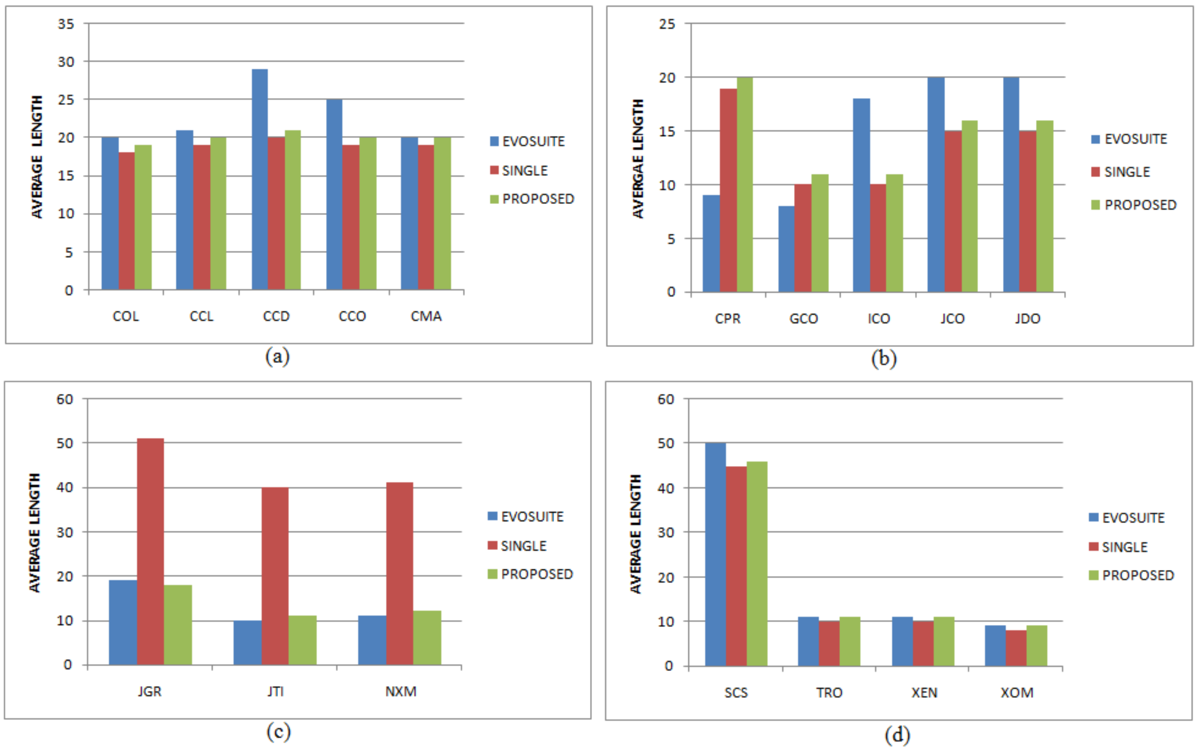
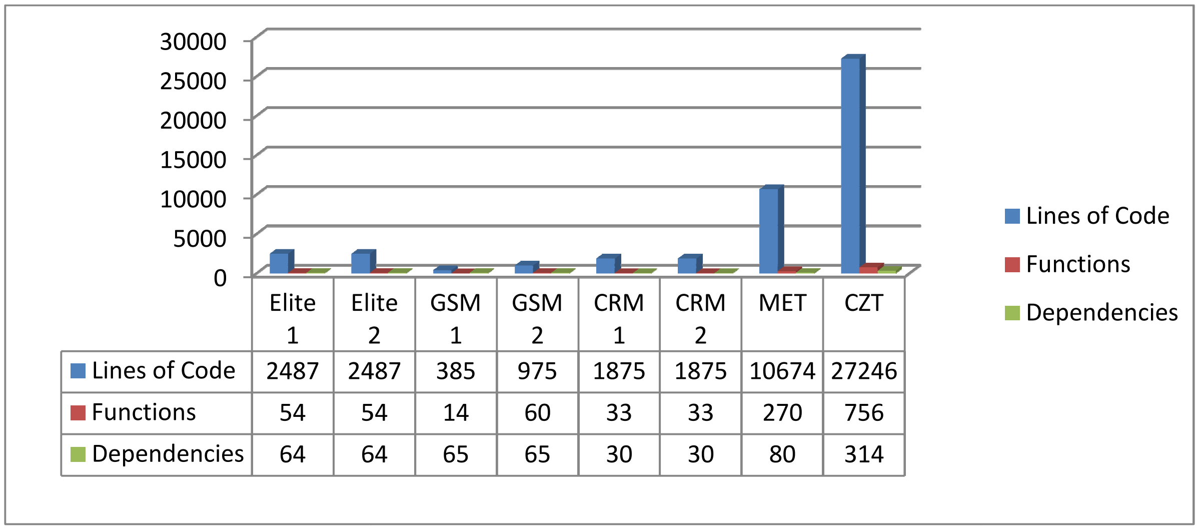


FIGURE 7: Average test suite length for all case studies.

As shown in Figure 7, the average test suite length of the proposed system and Evosuite perform better than the single branch strategy. The reason behind this is that when test suite is generated with multiple objectives in mind, the generated test cases will be less besides being able to cover all possible branches. In horizontal axis case studies and in vertical axis the average length is presented.



	Elite 1	Elite 2	GSM 1	GSM 2	CRM 1	CRM 2	MET	CZT
🟦 Lines of Code	2487	2487	385	975	1875	1875	10674	27246
🟥 Functions	54	54	14	60	33	33	270	756
🟩 Dependencies	64	64	65	65	30	30	80	314

FIGURE 8: Case study applications used for test case prioritization.

As shown in Figure 8, the case study applications and their statistics in terms of lines of code, number of functions and number of dependencies are presented. These case studies are used to apply the proposed test case prioritization approach with automated discovery of dependencies.

The results revealed that the prioritization can help increase the number of faults detected. This is the ultimate aim of the proposed tool in this paper.

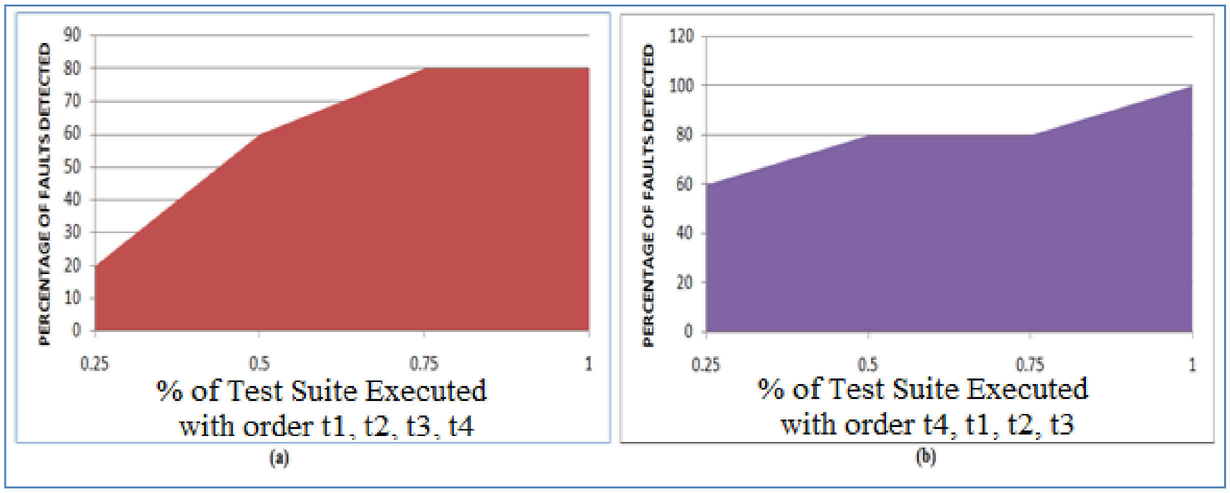


FIGURE 9: Faults detection dynamics with order t1, t2, t3, t4 (a) and t4, t1, t2, t3 (b).

As shown in Figure 9, it is evident that the percentage of test suites executed is presented in horizontal axis while the vertical axis presents percentage of faults detected. Test case prioritization can have its impact on the fault detection. This is the important hypothesis that has been tested with the proposed tool and underlying methodologies. As the results revealed, the prioritized test cases were able to detect more faults. Thus the hypothesis has been tested and proved to be positive.

5. DISCUSSION

In this paper, our research work is focused on three aspects, 1. Automatic test suite generation, 2. Automatic discovery of dependency structures and 3.Test case prioritization. We have followed Automatic Test Suite approach to generate Chromosomes of the given input program. The whole test suite generation, with proposed methodology as illustrated in Figure 2, count produce comparable results with Evosuite. Further the usage of GA operators identifies test cases which to be included in whole test suite. This is a representative and demonstrates high coverage of SUT. The mutation operators are used at Java byte code level to identify possible test cases. The results are obtained with several case study projects. Figure 4 shows different statistics of the case studies. The results presented in Figure 5 reveals the average number of missed infeasible targets between the proposed approach, EvoSuite and Single approaches. The proposed approach outperforms Single approach, as whole test suite is representative of multiple objectives and ensures high coverage. The proposed method is comparable with the Evosuite with little difference in performance.

Apart from the above mentioned results, the average branch coverage for all case studies were observe with our method and compared with Evosuite and Single. The rationale behind this is that the proposed approach and Evosuite follows whole test suite generation concept that considers representation of multiple objectives so as to reduce the number of test cases besides ensuring high branch coverage. The results are presented, which in Figure 6 ,that reveals the aforementioned outcomes. The comparative results show that the proposed approach has slight performance than the Evosuite.

Another significant contribution in this paper is automatic discovery of dependency structures. In the literature[7],[8] and [9],It is mentioned that dependency structures were used to prioritize test cases. As prioritization can improve the fault detection ratio, the dependencies of function calls in terms of open and closed dependencies can influence the results. In this regard, we proposed a methodology and algorithm which extracts dependency structures automatically. In [8], the dependencies are manually extracted, our methodology automate the procedure of extracting dependencies. Since manual prioritization is very time taking, this is significant improvement in our research. Our algorithm ensures that the dependencies are automatically extracted and used to prioritise test cases. The prioritization is non-trivial and hard to achieve complexities in the case studies with thousands of branches. The process of extraction of dependencies is not simple with very complex projects. Therefore proposed methodology is extraction of dependencies automatically paves way for significant speed and performance in software industry with respect to testing. The proposed methodology for test case prioritization includes the process of extracting dependencies a blend of reflection API for method identity. By using execution traces, the call processing insights in runtime behaviour of SUTs. This knowhow paves way for automatic prioritization of test cases. Further this results in improving the fault detection ratio. The methodology is in Figure 3 and the results that shows the performance improvement are presented in Figure 9. In Figure 9(b), clearly shows test case prioritization performance improves significantly.

6. COMPARATIVE RESULTS

In this section, the two approaches results were compared with the TRO case study. The number of faults detected is compared with automatic discovery of dependency structures in the proposed methodology and the manual identification of dependency structures in [8] .Figure 10 shows the comparison results.

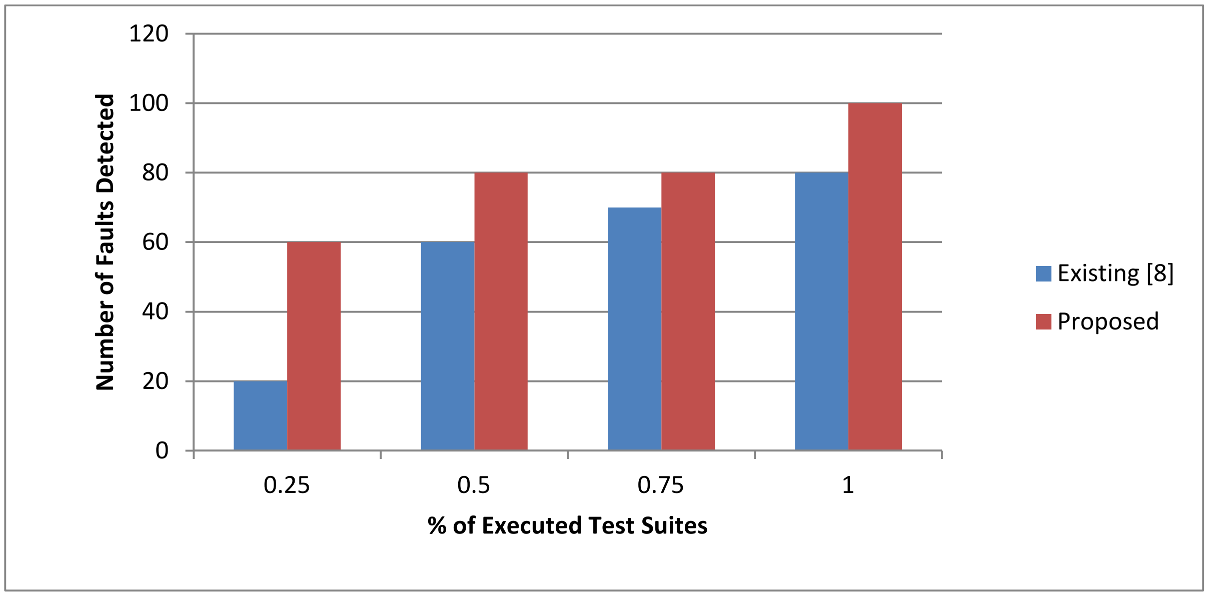


FIGURE 10: Comparison between Existing and Proposed Methodology with TRO project

As shown in Figure 10, the performance of the two approaches for given SUT is recorded. The results reveals that, the proposed methodology, determines the dependency structures automatically which is comparable with that of [8], were dependency structures are manually identified and prioritization performed. The increased count value in the fault detection is due to the fact that the test case prioritization can help in running test cases in proper order that will reflect in the increase of fault detection ratio.

7. CONCLUSIONS AND FUTURE WORK

In this paper our focus is three aspects pertaining to software testing. First, we proposed and implemented a mechanism for generating automatic case suite generation with multiple objectives besides achieving complete coverage. Second, we focused on the automatic extraction of dependency structures from SUT so as to improve prioritization of test cases. We proposed and implemented a methodology for this purpose. Third, we built a tool that demonstrates the automated test suite generation, automatic discovery of dependency structures and test case prioritization for improving the percentage of flaws detected. The integrated functionality has been tested. Our empirical results reveal significant improvement in the percentage of detection of flaws with the new approach. This research can be extended further to adapt our approaches to Software Product Lines (SPLs) in future. Software product line is the modern approach in software development that takes care of set of products with core and custom assets. The new products are derived based on the variabilities in the requirements. The SPL approach improves software development in different angles especially when software is delivered as product to clients. SPLs and their configuration management is complex in nature. Testing such applications need improved mechanisms that can leverage the characteristics of SPL for improving quality.

8. REFERENCES

[1] Gordon Fraser, and Andreas Zeller. (2012). Mutation-Driven Generation of Unit Tests and Oracles, IEEE, VOL. 38, NO. 2 p1-15.

[2] J. Malburg and G. Fraser, “Combining Search-Based and Constraint-Based Testing,” Proc. IEEE/ACM 26th Int’l Conf. Automated Software Eng., 2011.

[3] Gordon Fraser, and Andrea Arcuri. (2013). “Whole Test Suite Generation “ IEEE, VOL. 39, NO. 2, p276-291.

[4] A. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, and T. Vos, “Symbolic Search-Based Testing,” Proc. IEEE/ ACM 26th Int’l Conf. Automated Software Eng., 2011.

[5] Saswat Ananda, Edmund K. Burkeb, Tsong Yueh Chenc, John Clarkd, Myra B. Cohene, Wolfgang Grieskampf, Mark Harmang, Mary Jean Harroldh and Phil McMinni. (2013). “An orchestrated survey of methodologies for automated software test case generation.” p1979 2001.

[6] Richard Baker and Ibrahim Habli. (2010). An Empirical Evaluation of Mutation Testing For Improving the Test Quality of Safety- Critical Software. IEEE. p1-32.

[7] A. Arcuri and L. Briand, “A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering,” Proc. 33rd Int’l Conf. Software Eng., pp. 1-10. 2011.

[8] Shifa-e-Zehra Haidry and Tim Miller. (2013). Using Dependency Structures for Prioritization of Functional Test Suites. IEEE. 39 p258-275.

[9] J. Ryser and M. Glinz, “Using Dependency Charts to Improve Scenario-Based Testing,” Proc 17th Int’l Conf. Testing Computer Software, 2000.

[10] J. Kim and D. Bae, “An Approach to Feature Based Modeling by Dependency Alignment for the Maintenance of the Trustworthy System,” Proc. 28th Ann. Int’l Computer Software and Applications Conf., pp. 416-423, 2004.

[11] G. Rothermel, R.H. Untch, C. Chu, and M.J. Harrold, “Prioritizing Test Cases for Regression Testing,” IEEE Trans. Software Eng., vol. 27, no. 10, pp. 929-948, Sept. 2001.

[12] Z. Li, M. Harman, and R. Hierons, "Search Algorithms for Regression Test Case Prioritization,"IEEE Trans. Software Eng., vol. 33, no. 4, pp. 225-237, Apr. 2007.

[13] G. Rudolph, "Convergence Analysis of Canonical Genetic Algorithms," IEEE Trans. Neural Networks, vol. 5, no. 1, pp. 96-101, Jan. 1994.

[14] D. Jeffrey and N. Gupta, "Experiments with Test Case Prioritization Using Relevant Slices," J.Systems and Software, vol. 81, no. 2, pp. 196-221, 2008.

[15] Wong, J.R. Horgan, S. London, and H. Agrawal, "A Study of Effective Regression Testing in Practice," Proc. Eighth Int'l Symp. Software Reliability Eng., pp. 230-238, 1997.

[16] J. Li, "Prioritize Code for Testing to Improve Code Coverage of Complex Software," Proc. 16th IEEE Int'l Symp. Software Reliability Eng., pp. 75-84, 2005.

[17] J. Li, D. Weiss, and H. Yee, "Code-Coverage Guided Prioritized Test Generation," J. Information and Software Technology, vol. 48, no. 12, pp. 1187-1198, 2006.

[18] Z. Ma and J. Zhao, "Test Case Prioritization Based on Analysis of Program Structure," Proc. 15th Asia-Pacific Software Eng. Conf., pp. 471-478, 2008.

[19] R. Krishnamoorthi and S.A. Sahaaya Arul Mary, "Factor Oriented Requirement Coverage Based System Test Case Prioritization ofNew and Regression Test Cases," Information and Software Technology, vol. 51, no. 4, pp. 799-808, 2009.

[20] H. Srikanth, L. Williams, and J. Osborne, "System Test Case Prioritization of New and Regression Test Cases," Proc. Fourth Int'l Symp. Empirical Software Eng., pp. 62-71, 2005.

[21] P. Tonella, P. Avesani, and A. Susi, "Using the Case-Based Ranking Methodology for Test Case Prioritization," Proc. 22nd IEEE Int'l Conf. Software Maintenance, pp. 123-133, 2006.

[22] X. Qu, M.B. Cohen, and K.M. Woolf, "Combinatorial Interaction Regression Testing: A Study of Test Case Generation and Prioritization," Proc. IEEE Int'l Conf. Software Maintenance, pp. 255-264, 2007.

[23] F. Basanieri, A. Bertolino, and E. Marchetti, "The Cow_Suite Approach to Planning and Deriving Test Suites in UML Projects," Proc. Fifth Int'l Conf. Unified Modeling Language, pp. 275-303, 2002.

[24] B. Jiang, Z. Zhang, W. Chan, and T. Tse, "Adaptive Random Test Case Prioritization," Proc. IEEE/ACM Int'l Conf. Automated Software Eng., pp. 233-244, 2009.

[25] D. Kundu, M. Sarma, D. Samanta, and R. Mall, "System Testing for Object-Oriented Systems with Test Case Prioritization," Software Testing, Verification, and Reliability, vol. 19, no. 4, pp.97- 333, 2009.

[26] Andrea Arcuri,Muhammad Zohaib Iqbal and Lionel Briand. (2012). Random Testing: Theoretical Results and Practical Implications. IEEE. 38 p258-277.

[27] B. Baudry, F. Fleurey, J.-M. Je´ze´quel, and Y. Le Traon, "Automatic Test Cases Optimization: A Bacteriologic Algorithm," IEEE Software, vol. 22, no. 2, pp. 76-82, Mar./Apr. 2005.

[28] S. Zhang, D. Saff, Y. Bu, and M. Ernst, "Combined Static and Dynamic Automated Test Generation," Proc. ACM Int'l Symp. Software Testing and Analysis, 2011.

[29] P. Godefroid, N. Klarlund, and K. Sen, “DART: Directed Automated Random Testing,” Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation, pp. 213-223, 2005.

[30] W. Visser, C.S. Pasareanu, and S. Khurshid, “Test Input Generation with Java PathFinder,” Proc. ACM SIGSOFT Int’l Symp. Software Testing and Analysis, pp. 97-107, 2004.

[31] K. Inkumsah and T. Xie, “Improving Structural Testing of Object-Oriented Programs via Integrating Evolutionary Testing and Symbolic Execution,” Proc. 23rd IEEE/ACM Int’l Conf. Automated Software Eng., pp. 297-306, 2008.

[32] M. Islam and C. Csallner, “Dsc+Mock: A Test Case + Mock Class Generator in Support of Coding against Interfaces,” Proc. Eighth Int’l Workshop Dynamic Analysis, pp. 26-31, 2010.

[33] A. Arcuri, “A Theoretical and Empirical Analysis of the Role of Test Sequence Length in Software Testing for Structural Coverage,” IEEE Trans. Software Eng., vol. 38, no. 3, pp.497-519, May/ June 2011.

[34] C. Csallner and Y. Smaragdakis, “JCrasher: An Automatic Robustness Tester for Java,” Software Practice and Experience, vol. 34, pp. 1025-1050, 2004.

[35] W. Eric Wong, J. R. Horgan, Saul London, Hira Agrawal. (1997). A Study of Effective Regression Testing in Practice. IEEE. 8 (1), p 264-274.

[36] Greegg Rothermel, Sebastian Elbaum, Alexey Malishevsky, Praveen Kallakrui and Brian Davia (2011), The impact of test suite granularity on the cost-effectiveness of Regression Testing, University of Nebraska – Lincoln, p1-12.

[37] Hema Srikanth, Laurie Williams, Jason Osborne. (2000). System Test Case Prioritization of New and Regression Test Cases. Department of Computer Science. 2 (4), p1-23.

[38] J. Jenny Li. (2001). Prioritize Code for Testing to Improve Code Coverage of Complex Software. CID. p1-18.

[39] Jiang, B; Zhang, Z; Chan, WK; Tse, TH. (2009). Adaptive random test case prioritization. International Conference On Automated Software Engineering. 4 (24), p233-244.

[40] Johannes Ryser. (2000). Using Dependency Charts to Improve Scenario-Based Testing. International Conference on Testing Computer Software TCS. 18 (3), p1-10.

[41] Paolo Tonella, Paolo Avesani, Angelo Susi. (1997). Using the Case-Based Ranking Methodology for Test Case Prioritization. IEEE.p1-10.

[42] Sebastian Elbaum. (2001). Incorporating Varying Test Costs and Fault Severities into Test Case Prioritization. Proceedings of the 23rd International Conference on Software Engineering. 23 (5), p1-10.

[43] Xiao Qu, Myra B. Cohen, Katherine M. Woolf. (2006). Combinatorial Interaction Regression Testing: A Study of Test Case Generation and Prioritization. Department of Computer Science and Engineering, p149-170.

[44] Zengkai Ma. (2001). Test Case Prioritization based on Analysis of Program Structure. Department of Computer Science, p149-170.

[45] C. Prakasa Rao and P.Govindarajulu. (2015). “Automatic Discovery of Dependency Structures for Test Case Prioritization” . IJCSNS. 15 (n.d), p-52-57.

[46] C. Prakasa Rao and P.Govindarajulu. (2014). “A Comprehensive Survey of Recent Developments in Software Testing Methodologies”, IJSR. 3 (n.d), p-1889-1895.

[47] C. Prakasa Rao and P.Govindarajulu. (2015). “Genetic Algorithm for Automatic Generation of Representative Test Suite for Mutation Testing”. IJCSNS. 15 (n.d), p-11-17.