Genetic Algorithm for Automatic Generation of Representative Test Suite for Mutation Testing

C. Prakasa Rao[†]

Research Scholar, Dept. of Computer Science, S.V. University, Tirupati, India

P. Govindarajulu[††]

Retd. Professor, Dept. of Computer Science, S.V. University, Tirupati, India.

Abstract

Discovering bugs in software towards quality of software is given paramount importance in research arena. Towards this end automatic test case generation became essential as manual test data generation and adding test oracles is tedious task. It is more so when there are no formal specifications to unearth the faults in test outcome. Therefore, it is important to generate representative test sets that ensure complete code coverage. Genetic Algorithms are proved to be very useful for generation of unit tests and well suited for testing object oriented software systems. They are well known for their capabilities to test complex objects through sequences of method invocations. In this paper we used genetic algorithm for generating representative test suite for mutation testing. We built a tool that demonstrates the proof of concept. The empirical results are encouraging.

Index Terms

Test suite generation, branch coverage, genetic algorithm, search based testing

1. Introduction

It is a well known fact that software testing is an indispensable part of System Development Life Cycle (SDLC) which improves quality of software under test (SUT). Unit testing is widely used for unearthing bugs in SUT [18]. Each test contains test data required to execute specific program and the expected output. Traditionally many techniques came into existence over the years to generate test cases for discovering faults in SUT. Automatic test case generation is complex and challenging task [28]. In the same fashion automated test data generation is also difficult task[28]. Search based test data generation is explored in [14], [31]. Recently the focus was switched towards generation of test suites for high code coverage. Still there is a problem of determining the expected outcome which is also known as oracle problem. Therefore it is important to take care of automatic test suite generation and test oracles. For many years, as found in the literature, it was common practice to generate a test case for every coverage goal and combine all test cases to form test suite. A problem in this approach is that the size of test suite becomes unpredictable. The reason behind this is that test case generated for fulfilling one goal might be useful for other goals as well. This characteristic is exploited of late to produce representative test suite that not only covers the whole code besides ensuring smaller size test suite.

There are many problems when one goal is targeted at a time. For instance, some branches are difficult to consider for coverage while some branches are infeasible. It is still open issues to find out how much time needs to be spent and difficulty prediction of coverage goals. In this paper we proposed a test suite generation approach using Genetic Algorithm (GA) for producing representative test suite generation that satisfies all coverage goals. The generated test suite can be used to have mutation testing so that the bugs in the SUT can identify the hidden bugs . Mutation testing is fault-based testing that is widely used [19],[26]. The case study example is in Fig. 1 and Fig. 2 which is used for empirical study. The example simulates a real world vending machine which takes coin as input and gives the requested.

```
1 public class VendingMachine {
2       private int credit;
3       private LinkedList<String>  stock;
4       private static final int MAX = 10;
5       public VendingMachine() {
6               credit = 0;
7               stock = new LinkedList<String>();
8       }
9       public void coin(int coin) {
10              if (coin != 10 && coin != 25 && coin != 100)
11                      return;
12              if (credit >= 90)
13                      return;
14              credit = credit + coin;
15              return;
16      }
17      public int getChoc(StringBuffer choc) {
18              int change;
19              if (credit < 90 || stock.size() <= 0) {
20                      change = 0;
21                      choc.replace(0, choc.length(), "");
22                      return (change);
23              }
24              change = credit - 90;
25              credit = 0;
26              choc.replace(0, choc.length(), (String) stock.removeFirst());
27              return (change);
28      }
29      public void addChoc(String choc) {
30              if (stock.size() >= MAX)
31                      return;
32              stock.add(choc);
33              return;
34      }
35      public int getCredit() {
36              return credit;
37      }
38 }
```

Fig.1. Example vending machine implementation code item to the customer

Instead of generating test cases independently and then combining them into a test suite, our approach is to generate whole test suite that is representative of all

Manuscript received February 5, 2015
Manuscript revised February 20, 2015

coverage goals. All test cases in test suite are generated at a time and the fitness function used in GA takes care of testing goals simultaneously. The GA is used to optimize the generation of test suite. GA technique starts with initial population that contains test suite which is randomly generated.

```
StringBuffer choc = new StringBuffer("xx");
VendingMachine v = new VendingMachine();
v.addChoc("c1");
v.addChoc("c2");
v.addChoc("c3");
v.coin(10);
v.coin(25);
v.coin(100);
int ch = v.getChoc(choc);
System.out.println("First  get, c: " + choc + ", ch: " + ch);
ch = v.getChoc(choc);
v.coin(100);
ch = v.getChoc(choc);
System.out.println("Second get, c: " + choc + ", ch: " + ch);
```

Fig. 2. Test suite containing multiple tests for VendingMachine class

The generated test suite represents the test coverage goals. Significantly smaller test suite is generated when compared with test suites generated based on a single goal in mind. Our solution towards generating whole test suite is representative of all coverage goals so as to reduce the size of test suite. Whole test suite generation was explored earlier in [18], [26], [27], [18], [34]. Our contributions in this paper are as follows.

- We developed a genetic algorithm that takes care of generation of representative test suite generation at a time in order to have smaller size test suite besides ensuring full code coverage. The generated test suite can also be used for mutation testing.
- We built a tool that is designed to be modular and planned to be extended in our future research. It is meant for demonstrating the proof of concept with respect to test suite generation with minimal size and full coverage. Mutant killing test cases are directly generated from the given source code.

The remainder of the paper is structured as follows. Section 2 reviews literature on the prior works in test suite generation. Section 3 presents the proposed algorithm and methodology. Section 4 presents results of the experiments. Section 5 concludes the paper besides providing directions for future work.

2. Related Works

This section review prior works on automated test suite generation and related aspects. Genetic algorithms are widely used for generating test cases automatically. Rodolph [11] analyzed convergence properties pertaining to genetic algorithms. Arcuri and Briand [34] proposed adaptive random testing for enhancing random testing. Baudry et al. [19] explored automatic test case optimization using bacteriologic algorithm. Zhang et al. [27] combined both static and dynamic approaches for automated test case generation. Godefroid et al. [28] proposed a new approach named Directed Automated Random Testing (DART) for random test case generation. Tonella et al. [26] explored evolutionary programming for test case generation. Visser et al. [32] explored test input generation using PathFinder. Inkumsah and Xie [16] integrated evolutionary testing and symbolic execution for improving structural testing of SUT. Islam and Csallaner [13] proposed a technique for generating mock classes and test cases in support of coding with respect to interfaces. Fraser and Arcuri [30] explored on the length of test cases for structural coverage of SUT. With respect to robustness testing Csallner et al. [29] explored JCrasher for testing robustness testing of Java applications.

Malburg and Fraser [31] proposed a hybrid approach using constraint and search-based testing to test software. Fraser and Zeller [28] focused on mutation-driven generation of test cases and test oracles. Arcuri and Fraser [14] also found it useful to have parameter tuning with respect to search based software engineering. Many researchers contributed to test suite generation, mutation testing and automated testing of SUT as explored in [31]-[32].

3. Proposed Approach to Test Suite Generation

In this section we describe the search – based approach we followed towards generating test suite which is representative of testing goals and also maximizes mutation score. With respect to search based testing, genetic algorithms have been around for many years for leveraging test data derivation. In fact they are most popular and they are of meta-heuristic in nature. The GA takes initial population which is randomly generated and reproduction is made iteratively using operators like crossover and mutation. Thus the GA is one of the evolutionary algorithms which make use of fitness function to have optimal solutions. This paper considers the generation of test suite for objects oriented programming. In fact, we tested the proposed solution with Java source code. A test case is considered a set of statements denoted as t={s1, s2, s3, …, sl) of length l. In the conventional approach test suites are generated based on individual goals. In this paper,

we considered whole test suite generation that considers all goals at a time and the test suite gets generated with smaller size besides full coverage. Especially we used branch coverage approach that guides the test suite generation. Fitness function is used keeping branch coverage in mind which estimates the closeness of test suite with respect to all branches (if, while etc.) in the program.

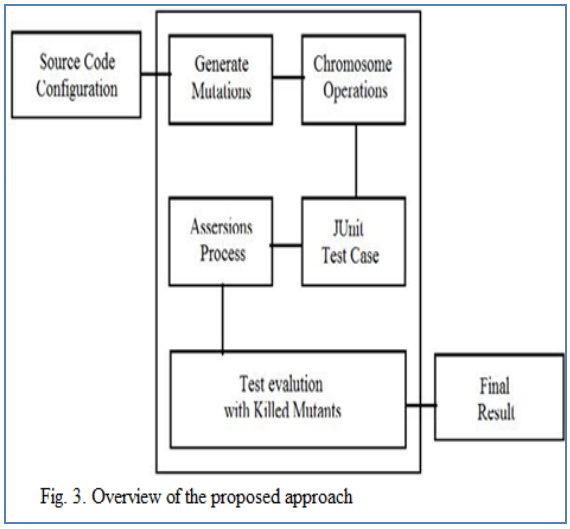

Fig. 3. Overview of the proposed approach

Fig. 3 shows the general framework that is used for test suite generation and mutation testing. The framework starts with source code configuration. It takes Java source code (currently works for Java application testing) as input and starts the genetic algorithm steps. In the process it makes use of mutations iteratively and performs chromosome operations. JUnit Test Case is used in order to have test cases. It has assertions process as well. Finally the test cases are evaluated and some mutants are killed. The final test suite is presented. Mutation operators are applied on Java byte-code level.

Operators of various kinds used in Java program are used to have mutants generated. Each operator can have different ways of execution and all possible ways are considered to have mutations. The mutation operators are used to perform various operations. The operators include delete call, delete field, insert unary operator, replace arithmetic operator, replace bitwise operator, replace comparison operator, replace constant, and replace variable.

4. Experimental Results

We built a tool in Java programming language. It is menu-driven and intuitive in nature. It takes Java source code as input and performs mutation testing through whole test suite generation. For given Java class presented in Fig. 1 many operators are involved and different mutants are generated. The sample details are as presented in Table 1, Table 2, Table 3, Table 4 and Table 5. The details are

Table 1 - Results for AORB operators

Operator	Mutant Name	Mutant Content	Chromosome
AORB	AORB_1	credit = credit * coin;	(line 30) void_coin(int):credit + coin => credit * coin
AORB	AORB_2	credit = credit / coin;	(line 30) void_coin(int):credit + coin => credit / coin
AORB	AORB_3	credit = credit % coin;	(line 30) void_coin(int):credit + coin => credit % coin
AORB	AORB_4	credit = credit - coin;	(line 30) void_coin(int):credit + coin => credit - coin
AORB	AORB_5	change = credit * 90;	(line 42) int_getChoc(java.lang.StringBuffer):credit - 90 => credit * 90
AORB	AORB_6	change = credit / 90;	(line 42) int_getChoc(java.lang.StringBuffer):credit - 90 => credit / 90
AORB	AORB_7	change = credit % 90;	(line 42) int_getChoc(java.lang.StringBuffer):credit - 90 => credit % 90
AORB	AORB_8	change = credit % 90;	(line 42) int_getChoc(java.lang.StringBuffer):credit - 90 => credit % 90

Table 2 – Some of the results of AOIS operators

Operator	Mutant Name	Mutant Content	Chromosome
AOIS	AOIS_1	if (++coin != 10 && coin != 25	((line 24) void_coin(int):coin
AOIS	AOIS_2	if (--coin != 10 && coin != 25	(line 24) void_coin(int):coin
AOIS	AOIS_3	if (coin++ != 10 && coin != 25	(line 24) void_coin(int):coin
AOIS	AOIS_4	if (coin-- != 10 && coin != 25	(line 24) void_coin(int):coin
AOIS	AOIS_5	if (coin != 10 && ++coin !=	(line 24) void_coin(int):coin
AOIS	AOIS_6	if (coin != 10 && --coin != 25	(line 24) void_coin(int):coin
AOIS	AOIS_7	if (coin != 10 && coin++ !=	(line 24) void_coin(int):coin
AOIS	AOIS_8	if (coin != 10 && coin-- != 25	(line 24) void_coin(int):coin
AOIS	AOIS_9	if (coin != 10 && coin != 25	(line 24) void_coin(int):coin
AOIS	AOIS_10	if (coin != 10 && coin != 25	(line 24) void_coin(int):coin
AOIS	AOIS_11	if (coin != 10 && coin != 25	(line 24) void_coin(int):coin

Table 3 – Some of the Results of ROR operators

Operator	Mutant Name	Mutant Content	Chromosome
ROR	ROR_1	if (coin > 10 && coin != 25 &&	(line 24) void_coin(int): coin != 10 => coin > 10
ROR	ROR_2	if (coin >= 10 && coin != 25	(line 24) void_coin(int): coin != 10 => coin >= 10
ROR	ROR_3	if (coin < 10 && coin != 25 &&	(line 24) void_coin(int): coin != 10 => coin < 10
ROR	ROR_4	if (coin <= 10 && coin != 25	(line 24) void_coin(int): coin != 10 => coin <= 10
ROR	ROR_5	if (coin == 10 && coin != 25	(line 24) void_coin(int): coin != 10 => coin == 10
ROR	ROR_6	if (true && coin != 25 && coin !=	(line 24) void_coin(int): coin != 10 => true
ROR	ROR_7	if (false && coin != 25 && coin !=	(line 24) void_coin(int): coin != 10 => false
ROR	ROR_8	if (coin != 10 && coin > 25 &&	(line 24) void_coin(int): coin != 25 => coin > 25
ROR	ROR_9	if (coin != 10 && coin >= 25 &&	(line 24) void_coin(int): coin != 25 => coin >= 25
ROR	ROR_10	if (coin != 10 && coin < 25 &&	(line 24) void_coin(int): coin != 25 => coin < 25

Table 4 – Results of AOIU operators

Operator	Mutant Name	Mutant Content	Chromosome
AOIU	AOIU_1	credit = -credit + coin;	(line 30) void_coin(int):credit => -credit
AOIU	AOIU_2	return -change;	(line 40) int_getChoc(java.lang.StringBuffer):change => -change
AOIU	AOIU_3	change = -credit - 90;	(line 42) int_getChoc(java.lang.StringBuffer):credit => -credit
AOIU	AOIU_4	return -change;	(line 45) int_getChoc(java.lang.StringBuffer):change => -change
AOIU	AOIU_5	return -credit;	(line 77) int_getCredit():credit => -credit

The results revealed that the mutants generated through whole test suite generation, it was proved that the algorithm is capable of generating test suite which is representative of all goals and the branch coverage is given main focus. The mutation score is computed by the application which tells the coverage dynamics. The higher code coverage can be achievable by getting less mutation score.

Table 5 – Results of COR operators

Operator	Mutant Name	Mutant Content	Chromosome
COR	COR_1	if (coin != 10 || coin != 25 && coin != 100) {	(line 24) void_coin(int): coin != 10 && coin != 25 => coin != 10 || coin != 25
COR	COR_2	if (coin != 10 ^ coin != 25 && coin != 100) {	(line 24) void_coin(int): coin != 10 && coin != 25 => coin != 10 ^ coin != 25
COR	COR_3	if (coin != 10 && coin != 25 || coin != 100) {	(line 24) void_coin(int): coin != 10 && coin != 25 && coin != 100 => coin != 10 && coin != 25 || coin != 100
COR	COR_4	if ((coin != 10 && coin != 25) ^ coin != 100) {	(line 24) void_coin(int): coin != 10 && coin != 25 && coin != 100 => (coin != 10 && coin != 25) ^ coin != 100
COR	COR_5	if (credit < 90 && stock.size() <= 0) {	(line 37) int_getChoc(java.lang.StringBuffer): credit < 90 || stock.size() <= 0 => credit < 90 &&
COR	COR_6	if (credit < 90 ^ stock.size() <= 0) {	(line 37) int_getChoc(java.lang.StringBuffer): credit < 90 || stock.size() <= 0 => credit < 90 ^ stock.size()

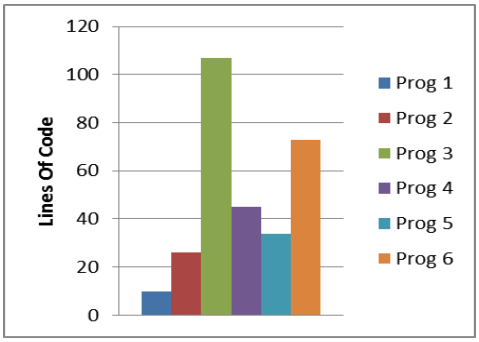

Figure 4 – LOC details of SUT

As can be seen in Figure 4, it is evident that the applications are presented in terms of the number of lines of code.

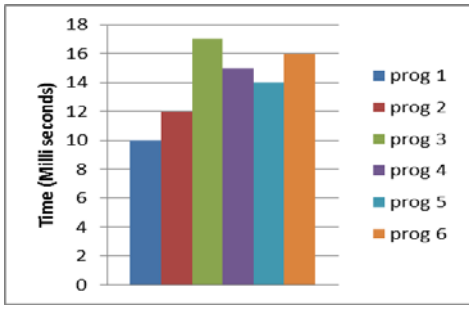

Figure 5 – Performance with different applications

As can be seen in Figure 5, it is evident that there is performance difference in generating test suites based on the size of the SUT.

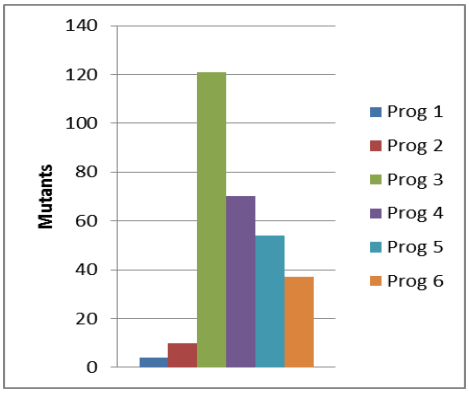

Figure 6 – Mutant dynamics of applications

As can be seen in Figure 6, it is evident that the number of mutants differs from each application.

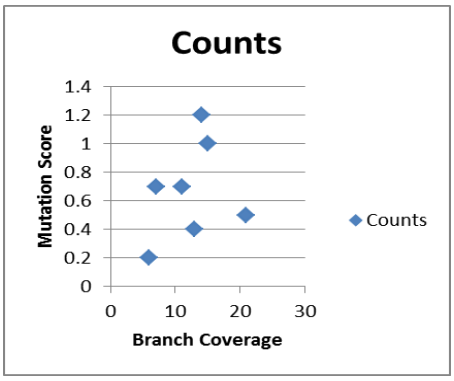

Figure 7 – Average branch coverage vs. average mutation score

As can be seen in Figure 7, it is evident that the results reveal the relationships between average branch coverage and average mutation score[26] using scatter chart.

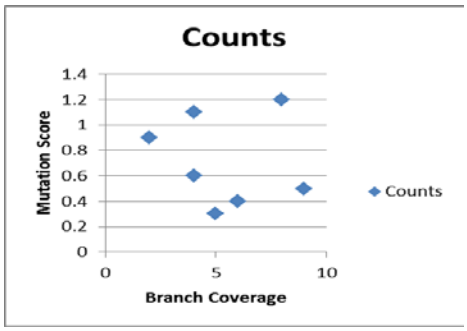

Figure 8 – Average branch coverage Vs. average mutation score.

As can be seen in Figure 8, it is evident that the results reveal the relationships between average branch coverage and average mutation score (proposed) using scatter chart.

5. Conclusions and Future Work

In this paper we studied the concept of generation of representative test suite which ensures complete code coverage. Coverage criteria play an important role in automatic test case generation. Traditional approaches targeted one particular coverage goal. From the recent experiments in software testing it is evident that the optimization of whole test suite generation is far better than the traditional method of targeting one coverage goal. Genetic algorithms have been applied successfully to generate unit tests for testing object oriented software. GA is one of the search based algorithms widely used to generate test cases. In this paper we applied GA for generating representative test suites and mutation testing. We built a tool to demonstrate the proof of concept. Mutation testing became easy with generation of test suite that covers all goals. The empirical results revealed that the application is capable of generating whole test suite which is representative of all test goals besides keeping it small in size. Our future work is to improve the tool for test suite prioritization.

Acknowledgments

I take this opportunity to sincerely acknowledge **Dr. Kancharla Ramaiah**, correspondent of Prakasam Engineering College for providing all the facilities, which buttressed me to perform my work comfortably. Foremost, I would like to express my sincere gratitude to fellow members of the teaching staff at the **Prakasam Engineering College**. My sincere thanks also goes to my Uncle **Dr. C. Subba Rao** for his inspiration and sparing his precious time. Last but not the least, I would like to thank my family : my parents **Murali Krihsna Rao** and

Vasantha Lakshmi, for giving birth to me at the first place and supporting me spiritually throughout my life.

References

[1] Alessandro Orso. (n.d). Integration Testing of Object-Oriented Software. p1-105.

[2] Ali Mesbah,Arie van Deursen and Danny Roest. (2012). Invariant-Based Automatic Testing of Modern Web Applications. *IEEE*. 38 p35-53

[3] Andrea Arcuri. (2012). A Theoretical and Empirical Analysis of the Role of Test Sequence Length in Software Testing for Structural Coverage.*IEEE*. 38 p497-519

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

[5] Bertrand Meyer, Ilinca Ciupa, Andreas Leitner and Lisa (Ling) Liu. (2007). Automatic testing of object-oriented software. *Chair of Software Engineering*. p1-17.

[6] Cemal Yilmaz. (2012). Test Case-Aware Combinatorial Interaction Testing. *IEEE*. p1-29.

[7] Claire Le Goues,ThanhVu Nguyen and Westley Weimer. (2012). GenProg: A Generic Method for Automatic Software Repair. *IEEE*. 38 p54-72.

[8] 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.

[9] George Kakarontzasa, Eleni Constantinoua, Apostolos Ampatzogloua and Ioannis Stamelosa. (2013). Layer assessment of object-oriented software: A metric facilitating white-box reuse. p350-366.

[10] Gordon Fraser and Andrea Arcuri. (2013). Whole Test Suite Generation.*IEEE*. 39 p276-291.

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

[12] 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.

[13] Hai Hua, Chang-Hai Jiang a, Kai-Yuan Cai a,b, W. Eric Wongc and Aditya P. Mathur d. (2013). Enhancing software reliability estimates using modified adaptive testing. p289-300

[14] Hui Liu, Zhiyi Ma, Weizhong Shao, and Zhendong Niu. (2012). Schedule of Bad Smell Detection and Resolution: A New Way to Save Effort. *IEEE*. 38 p220-235.

[15] James H. Andrews,Tim Menzies and Felix C.H. Li. (2011). Genetic Algorithms for Randomized Unit Testing. *IEEE*. 37 p80-94.

[16] Jerod W. Wilkerson, Jay F. Nunamaker Jr and Rick Mercer. (2012). Comparing the Defect Reduction Benefits of Code Inspection and Test-Driven Development. *IEEE*. 38 p547-560.

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

[18] Karthik Pattabiraman,Zbigniew T. Kalbarczyk, Member and Ravishankar K. Iyer. (2011). Automated Derivation of Application-Aware Error Detectors Using Static Analysis: The Trusted Illiac Approach. *IEEE*. 8 p44-57

[19] Kiran Lakhotia. (2009). Search Based Testing . p1-177

[20] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," Proc. 10th European Software Eng. Conf. Held Jointly with 13th ACM SIGSOFT Int'l Symp. Foundations of Software Eng., pp. 263-272, 2005.

[21] Malte Lochaua, Sascha Lityb,∗, Remo Lachmannc, Ina Schaeferc and Ursula GoltzbaTU. (2014). Delta-oriented model-based integration testing of large-scalesystems. p64-84.

[22] Marwa Shousha, Lionel C. Briand and Yvan Labiche. (2012). A UML/MARTE Model Analysis Method for Uncovering Scenarios Leading to Starvation and Deadlocks in Concurrent Systems. *IEEE*. 38 p354-374.

[23] Max Sch¨ afer, Andreas Thies, Friedrich Steimann and Frank Tip. (2012). A Comprehensive Approach to Naming and Accessibility in Refactoring Java Programs. *IEEE*. p1-27.

[24] Nina Elisabeth Holt a,⇑, Lionel C. Briand b and Richard Torkar. (2004). Empirical evaluations on the cost-effectiveness of state-based testing: An industrial case study. . . p891-910

[25] 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.

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

[27] Roberto Pietrantuono,Stefano Russo and Kishor S. (2010). Software Reliability and Testing Time Allocation: An Architecture-Based Approach.*IEEE*. p323-337.

[28] 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

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

[30] Tosin Daniel Oyetoyana, Daniela S. Cruzesa and Reidar Conradia. (2013). A study of cyclic dependencies on defect profile of software components. *IEEE*. p3163-3182.

[31] Vinicius Humberto Serapilha Durellia, Rodrigo Fraxino Araujoa,b and Marco Aurelio Graciotto Silva. (2013). A scoping study on the 25 years of research into software testing in Brazil and an outlook on the future of the area. p935-950.

[32] Yahya Rafique and Vojislav B. Mi˘si´. (2012). The Effects of Test-Driven Development on External Quality and Productivity: A Meta-Analysis. *IEEE*. X p1-24.

[33] Yasutaka Kamei,Emad Shihab and Naoyasu Ubayashi. (2011). A Large-Scale Empirical Study of Just-In-Time Quality Assurance. *IEEE*. . p1-19.

[34] 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.

Prakasa Rao Chapati received Master of Computer Applications degree from Madras University and Master of Technology degree in Computer Science & Engineering from Acharya Nagarjuna University. He is a research scholar in the department of computer science, Sri Venkateswara University. His research focus is on Software Testing to improve the Quality under Software Project Management perspective.

Prof P.Govindarajulu, Professor at Sri Venkateswara University, Tirupathi, has completed M.Tech., from IIT Madras (Chennai),Ph.D from IIT Bombay (Mumbai), His area of research are Databases, Data Mining, Image processing and Software Engineering