International Journal of Science and Research (IJSR)
ISSN (Online): 2319-7064
Impact Factor (2012): 3.358

A Comprehensive Survey of Recent Developments in Software Testing Methodologies

C. Prakasa Rao[1], P. Govindarajulu[2]

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

Assoc. Professor, PEC, Kandukur,

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

Abstract: ***Testing is one of the phases of all process models of software engineering. It remains the most indispensable part of software quality assurance. High reliability of software is expected in the real world as it, otherwise, becomes obsolete. It is more so with complex and machine critical applications. In this paper we provide a comprehensive survey of recent developments in software testing methodologies. Various approaches discussed in this paper include automatic generation of test cases, search based techniques, Just-In-Time quality assurance, static analysis, bad smell detection, early detection of concurrency problems, random testing, integration testing, combinatorial testing, model-based testing, test-driven approaches, dependency-based test case prioritization, state-based testing, adaptive testing and so on. This paper throws light into dependency structures for test case prioritization and test suite generation with minimum size test suites maximum coverage with discussion on empirical studies. The recent methodologies in software testing are focused in this paper besides finding potential gaps for future work.***

Keywords: Software testing, test-driven development, search-based methods, automatic software testing

1. Introduction

Software testing is one of the essential parts of software development process. A software test contains the definition for expected output besides the input that is used to execute the program. Many solutions came into existence for automatic testing of software. Test case generation with unit testing, integration testing and other approaches are found in the literature. The test case testing life cycle is presented in Figure 1. As there are phases in system development life cycle, in test life cycles also there are phases involved. Generally these phases are carried out in parallel with the development life cycle. In software testing, the coverage of test cases plays an important role to unearth all possible defects in the SUT. A common approach in generating test cases is to generate a test case for each coverage goal and combine them into a single test suite as explored in [27]. When a single goal at a time is considered, it generates more test suites or the size of test suite is more. In [12] a representative test suite or whole test suite is generated that will have full coverage besides reducing size of test suites.

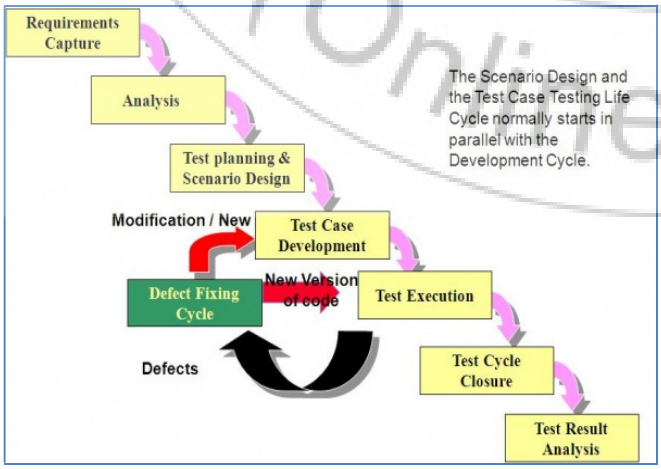


Figure 1: Test case testing life cycle

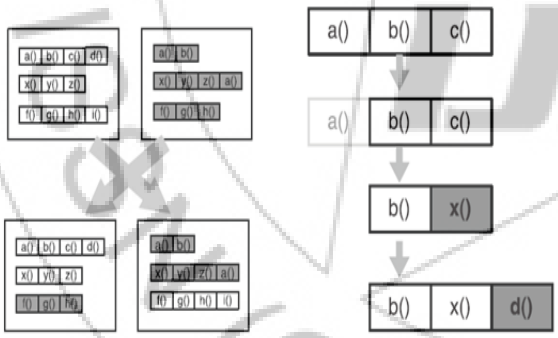
There are many approaches to test case generation. They are automatic test case generation approaches [6], [8], [3], [4], [31] and [12], search – based approaches [20], [4] and [31], architecture based approaches [29], Just-In-Time quality assurance [35], bad smell detection types and approaches [15], early detection of concurrency problems [22], static analysis for test case generation [19], interaction testing approaches [7], random testing approaches [5], test-driven approaches [34] and [17], dependency based test case prioritization techniques [30], [32], state based testing approaches [25], refactoring approaches [23], combinatorial testing [31], adaptive testing approaches [14] and [31], layer assessment approaches [10], model based approaches [31] and [24], state-based testing approaches [25] and integration testing [2]. The role of test sequence length in structural coverage of software testing was explored in [4]. There is evidence that there is relationship between test sequence length and structural coverage. Genetic algorithms are used as evolutionary approaches in test case generation as explored in [28], [16], [11] and [12].

In this paper our contributions include the review of literature in finding various methodologies employed for test case generation besides identifying potential research gaps that can help in directing for future research. The remainder of the report is structured as follows. Section 2 presents GA based methodologies. Section 3 provides dependency based solutions for software testing. Section 4 presents test-driven software testing approaches. Section 5 provides search-based techniques. Section 6 presents bad smells detection and resolution. Section 7 presents other approaches such as state-based testing, refactoring, architecture based solution, adaptive testing, combinatorial testing and so on. Section 8 presents the recent research and the gaps in the research that can help in planning for future work. Section 9 concludes the paper besides giving directions for future work.

2. GA Based Methodologies

2.1 Whole Test Suite Generation

Fraser and Arcuri [12] presented a novel approach to generate whole test suite that fulfills all coverage goals besides keeping the size of test suits small. They implemented a tool named EVOSUITE for efficiently testing the whole test suite generation. An evolutional approach using Genetic Algorithms (GA) is used to achieve this. The solution here is a test suite represented as T = {t1, t2, t3, …}. Here t1 represents a program that is used test a part of Software Under Test (SUT). In the same fashion, a test case is treated as sequence of instructions represented as t = {s1, s2, s3, …}. The test suite's length is considered as the sum of length of all test cases involved in test suite. It is represented as length(T) = $\sum_{t \in T} l_t$. The statement or instruction denoted as t is of four types namely primitive statement, constructor statement, field statement and assignment statement. Enumeration variables, numeric variables, Strings and Boolean variables come under primitives. The statements used to construct objects come under constructor statements while the statements that make use of public member variables and they are part of object are known as field statements. The assignment statements are instructions that assign values to variables or arrays or collections. As part of fitness function the notion of branch coverage is used for test criterion. Such fitness value is used to measure how close the test suite is that has maximum coverage. There is *bloat* control mechanism which avoid generating longer test cases. Test suite cross over and test suite mutation are the genetic operators used in the solution [25].



(a)Test Suite Crossover (b) Test Case Mutation
Figure 1: Search operators using GA [12]

In order to initialize first generation, random test cases are used. The tool which implemented this makes use of JUnit test cases and generates test suites for given Java source code. The tool also uses byte code instrumentation to have to obtain additional information when required. It also uses a security manager to deal with security issues. Experiments are made with many real time software products for testing. The experimental results are compared with single branch strategy [12]. The empirical results revealed that the whole test suite generation has higher performance when compared with single branch strategy as presented in Figure 2.

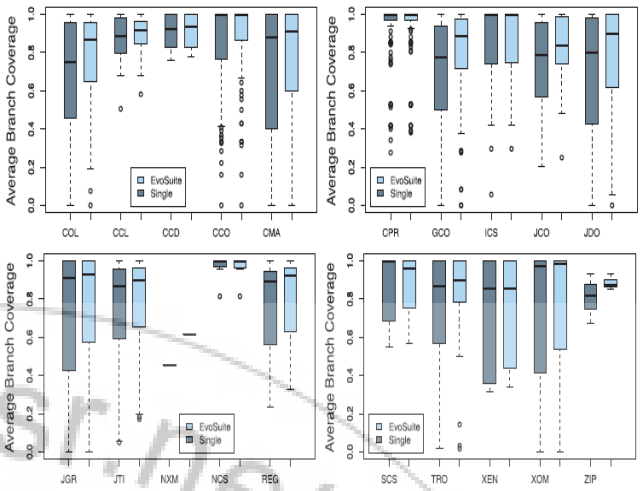


Figure 2: Average branch coverage [12]

From the experiments conclusions are made such as high coverage can be achieved using whole test suite generation besides producing smaller test suites. Evolutionary algorithms using GA performed better when the tool is compared with other tools that used different approaches to solving the problem. With respect to path coverage the whole test suite generation is compared with other tools such as CUTE [21] and DART [26] and it showed higher performance.

2.2 Other GA Based Methodologies

Baker and Babli [28] applied mutation testing for testing safety-critical software systems as SUT and experimented on improving the test quality. Program size is one of the characteristics considered for mutation testing. Their experimental results revealed that mutation testing provides measure for test quality. Andrews et al. [16] employed randomized unit testing using genetic algorithms along with a Feature Subset Selection (FSS) tool for assessing size in GA. This approach was proved to be more useful when compared with search-based approaches. Fraser and Zeller [11] introduced artificial detects called mutations into the program and presented an automated solution for generating test cases. The empirical results revealed that the approach could generate test suits that uncover more defects in the system.

3. Dependency Based Solutions

3.1 Test Suite Prioritization

Fault detection is the main goal of any test suite which has multiple test cases. However, some test cases depend on other test cases. Provided this fact, it is essential to identify dependency structure in order to priorities test cases. When test cases are prioritized, the resultant functional test suites can produce quality feedback that helps developers to focus on the issues and rectify problems. Haidry and Miller [32] studied the problem of test suit prioritization. They focused on a hypothesis that tells that dependencies among test cases can have their impact on the fault detection rate. The test case prioritization is the process of ordering test cases in order to increase the possibility of fault detection. The

number of defects unearthed in SUT can be called as rate of fault detection. In SUT some interactions should occur prior to other interactions causing dependencies problem [32]. Figure 3 shows a sample dependency structure.

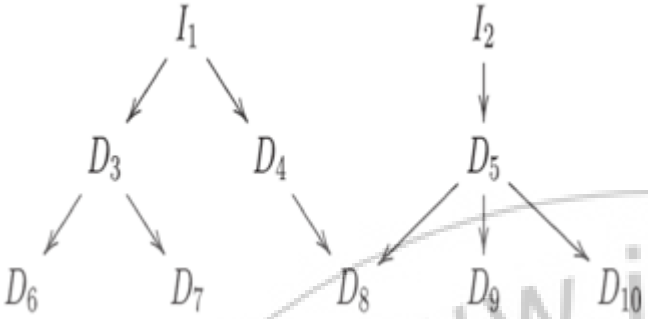


Figure 3: Sample dependency structure [32]

As shown in Figure 3, the root nodes do not have any dependencies while all other nodes do have dependencies. There are direct dependencies and indirect dependencies. For instance D6 is indirectly dependent on I1 and directly dependent on D3. Dependency is of two types namely open dependency and closed dependency. Open dependency refers to the fact that a test case needs to be executed before another test case but need not be immediately before the test case. The closed dependency says that a test case needs to be executed just before another test case based on the dependency. There might be some dependencies that are combination of closed and open dependencies. Two graph coverage measures are used to know dependency structures. They are known as DSP height and DSP volume. DSP is the acronym for Dependency Structure Prioritization. DSP volume refers to the number of dependencies. DSP height refers to the level of deepest dependencies. DSP volume can be computed as all indirect and direct dependencies of a test case while DSP height is computed as the height of all test paths and considering the one which has longest path. The test cases ordering are done using these two graph measures. Two sets of experiments are made to test both open dependencies and closed dependencies respectively. Figure 4 shows the artifacts collected from real world and the metrics for the SUT.

Artifact	Type	Lines of code	Functions	Faults	Tests	Depend-encies	Graph density	Unconnected tests	Maximum depth
$Elite_1$	component	2487	54	72	64	64	0.03175	3	17
$Elite_2$	component	2487	54	64	64	64	0.03175	3	17
GSM_1	unit	385	14	15	51	65	0.05098	0	6
GSM_2	unit	975	60	14	51	65	0.05098	0	6
CRM_1	component	1875	33	50	30	30	0.06897	0	6
CRM_2	component	1875	33	30	30	30	0.06897	0	6
MET	system	10,674	270	37	81	80	0.02469	0	6
CZT	component	27,246	756	27	548	314	0.00210	161	5
Bash (×6)	system	≈ 59,800	≈ 1051	3-8	1061	461	0.00094	460	12

Figure 4: Statistics of SUT [32]

As can be seen in Figure 4, it is evident that the Bash has highest dependencies while the CRM1 and CRM2 have least dependencies. These SUTs are used for experiments to know both closed and open dependencies and generate test suites with test case prioritization. The experiments are made to demonstrate the usefulness of test cases in order to increase fault detection rate. Average Percentage of Faults Detected (APFD) is the measure defined in [13] used to know the rate of fault detection. The more its value is the more the rate of fault detection.

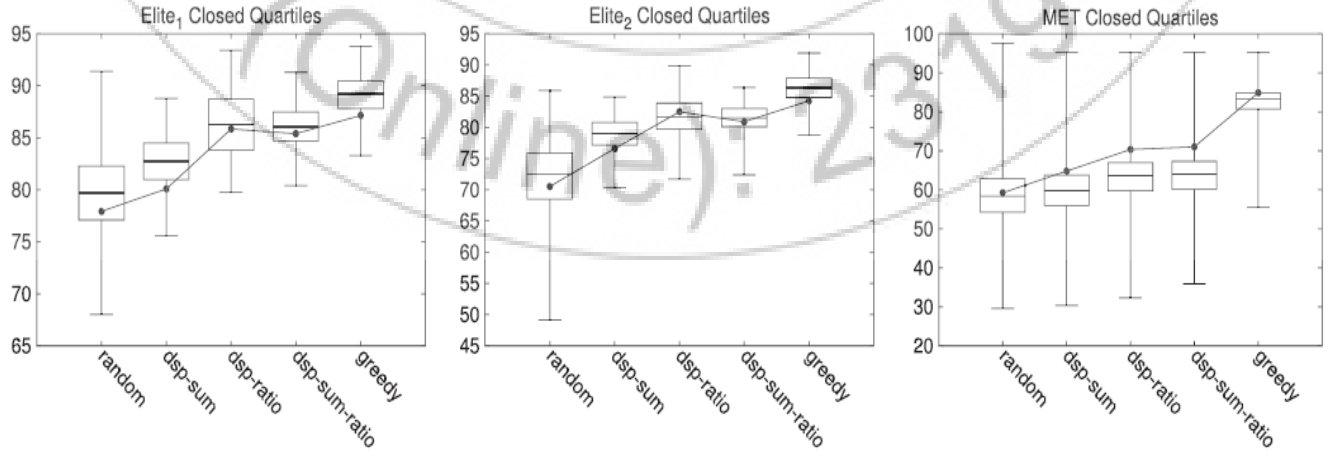


Figure 5: APFD for closed dependencies [32]

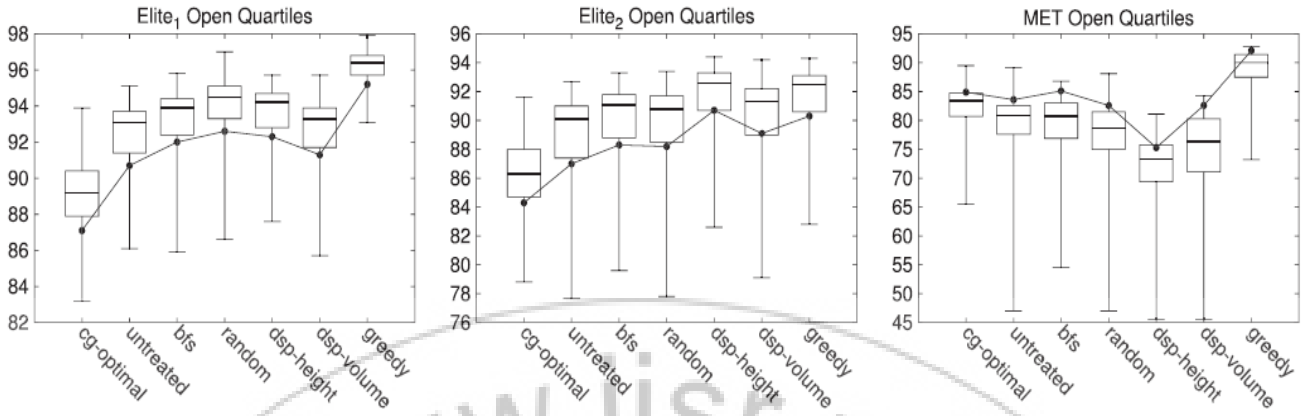
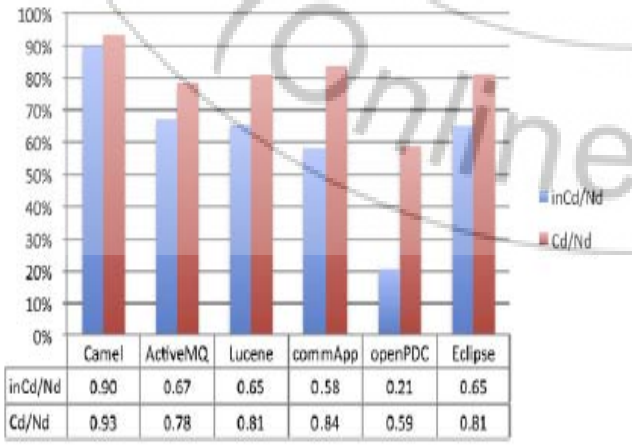


Figure 6: APFD for open dependencies [32]

As shown in Figure 5 and Figure 6, it is evident that the many algorithms are employed to test the APFD for closed and open dependencies respectively. There are DSP prioritization methods and other methods that do not use DSP measures. The empirical results revealed that the DSP prioritization methods achieved higher APFD when compared with non-DSP prioritization methods. Both experiments proved that DSP measures yield best performance in test case prioritization and also fault detection ratio for given SUT. There are three test case prioritization techniques that are close to the approach followed in [32]. They include history – based [13], knowledge – based [36] and model-based [9]. The first one takes information from prior execution cycles; the second one uses human knowledge for the task while the third uses a model of the system for test case prioritization.

3.2 Cyclic Dependencies and Quality of Software

Oyetoyan et al. [30] studied cyclic dependencies in SUT and the quality of the software. They made experiments on the object oriented metrics on cyclic dependencies to relate them with error-proneness. The results revealed that the cyclic components in software caused more defects in the system. This will have influence in software testing and maintenance. Refactoring such components is required in order to improve the quality of software. Another observation is that software complexity adds to the error-proneness.



	Camel	ActiveMQ	Lucene	commApp	openPDC	Eclipse
inCd/Nd	0.90	0.67	0.65	0.58	0.21	0.65
Cd/Nd	0.93	0.78	0.81	0.84	0.59	0.81

Figure 8: Defective components in cyclic group in class and package respectively [30]

As seen in Figure 8, it is evident that there is high rate of defective classes in cyclic groups. For instance Apache Camel exhibits 90% defects at class level and 82% at package level. This way other products and their defective components are presented in Figure 8.

4. Test-Driven Software Testing Approaches

Rafique and Mi˘si´c [34] studied the impact of test-driven development (TTD) on productivity and code quality. Developer's task size, test-driven approach has significant influence on the quality of software. TDD has positive effect on quality of software. Meta – analytical techniques were used to know the effectiveness of TDD in software quality. However, the productivity of TDD is inconclusive as it needed further research efforts. Wilkerson et al. [17] presented two approaches for software testing. The first approach is code inspection while the second approach is test-driven development. As far as reduction of defects is concerned, the code inspection has more advantages. However, it is proved to be expensive. When compared to traditional programming methods, the TDD approach has no significant improvement over the code inspection approach. Code inspection proved to be more effective than TDD.

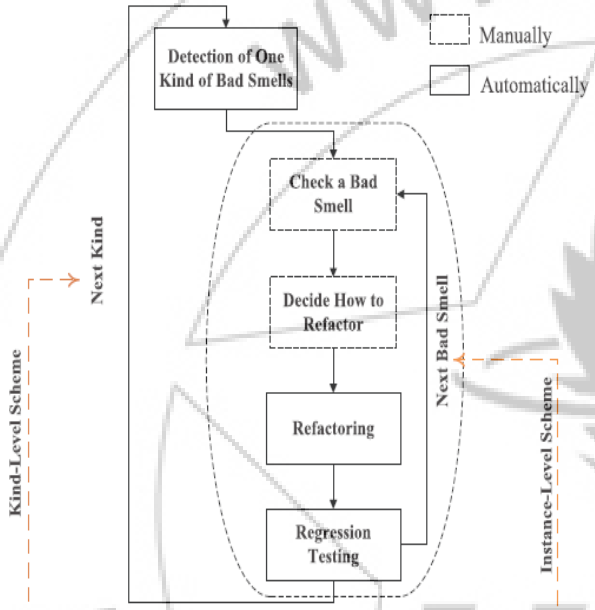
5. Search Based Testing Approaches

In [20] meta-heuristic algorithms were applied for search based testing. The algorithms were integrated with a tool named AUSTIN (Augmented Search-Based Testing). The tool is meant for structural data generation so as to cover all branches in SUT. A pseudo symbolic execution was used along with search based approaches. Hill climbing search algorithm is used for achieving this. Hill Climbing is one of the local search methods and also known as neighborhood search that starts with search space that is randomly chosen. The tool was built for testing applications built in C language. Thus the tool AUSTIN fills the gap between research and the industry application with respect to search-based testing solution for C programs. In [4] four search based techniques are compared. They include Genetic Algorithm (GA), Evolutionary Algorithm (EA), Hill Climbing (HC) and Random Search (RS). The performance of these algorithms was tested under different lengths of test sequences. The empirical results revealed that different algorithms provided performance differently based on SUT. However, a fact proven is that the length of test sequence

has its impact on structural coverage of SUT. In [31] search based software testing (SBST) was presented for automated test data generation. With respect to evolutionary algorithms, fitness function is used to guide the convergence of test cases.

6. Saving Effort through Bad Smell Detection

Potential problems can exist in SUT. The sign of such problems can be called as bad smell. Liu et al. [15] explored the concept of scheduling bad smell detection for resolving issues in the code. Towards this end, they presented a sequence of steps that can be used to achieve the desired results.



Different kinds of bad Instances of a specific smells are detected kind of bad smell are and resolved one after resolved one after the other the other

Figure 10: Procedure for detection resolution of bad smells [15]

As can be seen in Figure 10, it is evident that there is certain procedure to be followed to detect various kinds of bad smells such as duplicate code, long method, large class, long parameter list, useless class, useless method, useless field, primitive obsession, feature envy etc. Pair wise resolution sequences are constructed in order to detect and resolve bad smells with ease. This also reveals potential relationships among bad smells.

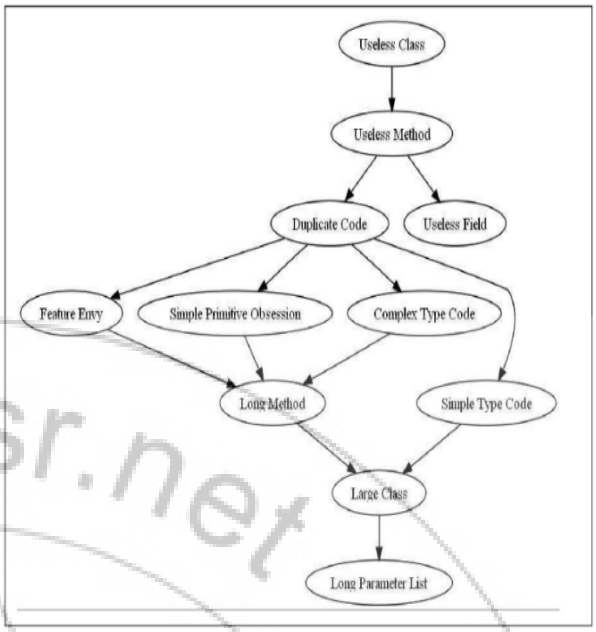


Figure 11: Illustrates pair wise resolution sequences [15]

Schematic overview of the pair wise resolution sequences is presented in Figure 11 that help in finding dependencies and relationships so as to detect bad smells easily. Commonly occurring bad smells in source code can be detected in this approach. When this knowledge is applied in testing software products, it can greatly help in identifying bad smells and generate a comprehensive report which can guide developers taking necessary steps [15].

7. Other Testing Mechanisms

Kamei et al. [35] focused on Just-In-Time quality assurance by unearthing potential errors in code in the early stages of system development life cycle. This solution overcomes the problems of other quality assurance approaches. The drawbacks it overcomes include coarse-grained prediction units, inability to identity relevant experts, and late predictions. Andrews [16] focused on randomized unit testing using GAs. Goues et al. [8] presented a generic method that detects problems software and repair it automatically. Their method is named "GenProg" which is based on GA. Delta-debugging and tree-structured differential techniques in order to repair software. Shousha et al. [22] presented a solution for early detection of concurrency issues such as deadlocks and starvation in software using UML modeling. Yilmaz [7] presented combinatorial interaction testing which covers arrays and collections which is test case - aware. Various configuration space models were explored for testing SUT. Mesbah et al. [3] proposed a novel method for automatic testing of AJAX – based modern applications. Fault detection was achieved using DOM-tree variants that can be used as test oracles. Their approach is known as invariant based automatically testing that supports plug-in for scalable and expandable solution.

Arcuri et al. [5] studied random testing. Their empirical study reveals the relationship between the random testing and quality of SUT. Sch¨ afer et al. [23] focused on accessibility and naming problems with respect to

refactoring in Java applications. They presented a tool for refactoring to overcome the issues. Hu et al. [14] proposed an adaptive testing approach with the help of a set of enhanced metrics. Their method is known as Modified Adaptive Testing (MAT) which makes use of history to achieve adaptive testing for higher reliability of SUT. Durelli et al. [33] reviews 25 years of software testing research in Brazil. Anand et al. [31] also present a comprehensive survey of literature on testing methodologies. Kakarontzas et al. [10] presents layer assessment approach for object oriented software. This approach can be used as a metric for white-box reuse. Holt et al. [25] presented an approach known as state-based testing (SBT) using test models that provide knowledge of behavior of SUT. They also evaluate the cost effectiveness of SBT. Lochau et al. [24] presents a model for integration testing. The model is model-oriented and delta-oriented. It is an integrated approach in generating and reusing of test artifacts for software testing.

8. Conclusions and Future Work

In this paper we studied various software testing methodologies used in the recent past. We have made a comprehensive review of recent developments in software testing methodologies. The testing approaches covered in this paper include dependency based approaches, genetic algorithms and evolutionary approaches, search based techniques, automated software testing methods, random testing, integration testing, interaction testing, layer assessment, model based testing, combinatorial testing, test-driven approaches, just-in-time software quality assurance, early detection of concurrency problems in software development cycle, static analysis, and architecture based approaches. This study also provides insights into experimental results pertaining to test driven approaches, dependency based solutions and GA based approach. Finally this survey presents potential research gaps that can be used for future work. We intend to work on a representative test suite generation combined with dependency structure based test case prioritization in future work.

9. Acknowledgements

I am Grateful to my family and fellow members of the teaching staff at the **Prakasam Engineering College.** I Sincere thanks to **Dr. Kancharla Ramaiah**, correspondent of Prakasam Engineering College for providing all the facilities. My Special thanks would always go to my parents. My acknowledgments go to my Uncle **Dr. C. Subba Rao** for his inspiration and sparing his precious time.

References

[1] 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.
[2] Alessandro Orso. Integration Testing of Object-Oriented Software. p1-105.
[3] Ali Mesbah,Arie van Deursen and Danny Roest. Invariant-Based Automatic Testing of Modern Web Applications. *IEEE*. 38 p35-53. 2012.
[4] Andrea Arcuri. A Theoretical and Empirical Analysis of the Role of Test Sequence Length in Software Testing for Structural Coverage.*IEEE*. 38 p497-519. 2012.
[5] Andrea Arcuri,Muhammad Zohaib Iqbal and Lionel Briand. Random Testing: Theoretical Results and Practical Implications. *IEEE*. 38 p258-277. 2012.
[6] Bertrand Meyer, Ilinca Ciupa, Andreas Leitner and Lisa (Ling) Liu. Automatic testing of object-oriented software. *Chair of Software Engineering*. p1-17. 2007.
[7] Cemal Yilmaz. Test Case-Aware Combinatorial Interaction Testing. *IEEE*. p1-29. 2012
[8] Claire Le Goues,ThanhVu Nguyen and Westley Weimer. GenProg: A Generic Method for Automatic Software Repair. *IEEE*. 38 p54-72. 2012.
[9] 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.
[10] George Kakarontzasa, Eleni Constantinoua, Apostolos Ampatzogloua and Ioannis Stamelosa. Layer assessment of object-oriented software: A metric facilitating white-box reuse. p350-366. 2013.
[11] Gordon Fraser, and Andreas Zeller. Mutation-Driven Generation of Unit Tests and Oracles, IEEE, VOL. 38, NO. 2, p1-15. 2012.
[12] Gordon Fraser and Andrea Arcuri. Whole Test Suite Generation.*IEEE*. 39 p276-291. 2013.
[13] 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.
[14] 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
[15] Hui Liu, Zhiyi Ma, Weizhong Shao, and Zhendong Niu. Schedule of Bad Smell Detection and Resolution: A New Way to Save Effort. *IEEE*. 38 p220-235. 2012.
[16] James H. Andrews,Tim Menzies and Felix C.H. Li. Genetic Algorithms for Randomized Unit Testing. *IEEE*. 37 p80-94. 2011.
[17] Jerod W. Wilkerson, Jay F. Nunamaker Jr and Rick Mercer. Comparing the Defect Reduction Benefits of Code Inspection and Test-Driven Development. *IEEE*. 38. p547-560. 2012.
[18] J. Malburg and G. Fraser, "Combining Search-Based and Constraint-Based Testing," Proc. IEEE/ACM 26th Int'l Conf. Automated Software Eng., 2011.
[19] Karthik Pattabiraman,Zbigniew T. Kalbarczyk, Member and Ravishankar K. Iyer. Automated Derivation of Application-Aware Error Detectors Using Static Analysis: The Trusted Illiac Approach. *IEEE*. 8 p44-57. 2011.
[20] Kiran Lakhotia. Search Based Testing . p1-177. 2009.
[21] 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.

[22] Marwa Shousha, Lionel C. Briand and Yvan Labiche. A UML/MARTE Model Analysis Method for Uncovering Scenarios Leading to Starvation and Deadlocks in Concurrent Systems. *IEEE*. 38 p354-374. 2012.
[23] Max Sch¨ afer, Andreas Thies, Friedrich Steimann and Frank Tip. A Comprehensive Approach to Naming and Accessibility in Refactoring Java Programs. *IEEE*. p1-27. 2012.
[24] Malte Lochaua, Sascha Lityb,∗, Remo Lachmannc, Ina Schaeferc and Ursula GoltzbaTU. Delta-oriented model-based integration testing of large-scalesystems. p64-84. 2014.
[25] Nina Elisabeth Holt a,⇑, Lionel C. Briand b and Richard Torkar. Empirical evaluations on the cost-effectiveness of state-based testing: An industrial case study. p891-910. 2004.
[26] 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.
[27] P. Tonella, "Evolutionary Testing of Classes," Proc. ACM SIGSOFT Int'l Symp. Software Testing and Analysis, pp. 119-128, 2004.
[28] Richard Baker and Ibrahim Habli. An Empirical Evaluation of Mutation Testing For Improving the Test Quality of Safety- Critical Software. *IEEE*. p1-32. 2010.
[29] Roberto Pietrantuono,Stefano Russo and Kishor S. Software Reliability and Testing Time Allocation: An Architecture-Based Approach.*IEEE*. p323-337. 2010.
[30] Tosin Daniel Oyetoyana, Daniela S. Cruzesa and Reidar Conradia. A study of cyclic dependencies on defect profile of software components. *IEEE*. p3163-3182. 2013.
[31] Saswat Ananda, Edmund K. Burkeb, Tsong Yueh Chenc, John Clarkd, Myra B. Cohene, Wolfgang Grieskampf, Mark Harmang, Mary Jean Harroldh and Phil McMinni. An orchestrated survey of methodologies for automated software test case generation. p1979-2001. 2013.
[32] Shifa-e-Zehra Haidry and Tim Miller. Using Dependency Structures for Prioritization of Functional Test Suites. *IEEE*. 39 p258-275. 2013.
[33] Vinicius Humberto Serapilha Durellia, Rodrigo Fraxino Araujoa,b and Marco Aurelio Graciotto Silva. 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. 2013.
[34] Yahya Rafique and Vojislav B. Mi˘si´. The Effects of Test-Driven Development on External Quality and Productivity: A Meta-Analysis. *IEEE*. X p1-24. 2012.
[35] Yasutaka Kamei,Emad Shihab and Naoyasu Ubayashi. A Large-Scale Empirical Study of Just-In-Time Quality Assurance. *IEEE*. p1-19. 2011.
[36] 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.

Author Profile

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.