International Journal of Engineering Research and Development
e-ISSN: 2278-067X, **p**-ISSN: 2278-800X, www.ijerd.com
Volume 5, Issue 12 (February 2013), PP. 10-17

Producing Test Cases Robotically by Using Specification Mining and Code Coverage

Ms. S. Vydehi[1], Ms. Sathyabama B[2]
[1]B.Sc(cs)., MCA., M.phil(cs),Head, Department of Computer Science, Dr. SNS Rajalakshmi College of arts and science, Coimbatore, Tamil Nadu, India.
[2]B MCA., (M.Phil(cs), Department of Computer Science, Dr. SNS Rajalakshmi college of arts and science, Coimbatore, TamilNadu, India.

Abstract:- Mining specifications and using them for bug detection is a promising way to reveal bugs in programs. Existing approaches suffer from two problems. First, dynamic specification miners require input that drives a program to generate common usage patterns. Second, existing approaches report false positives, that is, spurious warnings that mislead developers and reduce the practicability of the approach. The time spent in testing is mainly concerned with generating the test cases and testing them. The goal of this paper is to reduce the time spent in testing by reducing the number of test cases. For this data mining techniques are incorporated to reduce the number of test cases. Data mining finds similar patterns in test cases which helped in finding out redundancy incorporated by automatic generated test cases. The final test suite is tested for coverage which yielded good results. Specification mining not only helps to automate coverage - driven simulation or formal verification, it can also provide useful information for diagnosis. A specification mining-based diagnosis framework is proposed that can be used to simultaneously understand the error and locate it. If not enough tests are available, the resulting specification may be too incomplete to be useful. To solve this problem Code coverage analysis is the process of finding areas of a program not exercised by a set of test cases, creating additional test cases to increase coverage and determining a quantitative measure of code coverage, which is an indirect measure of quality.

Keywords: Code Coverage, False positive, True Positive, Quality, Test case.

I. INTRODUCTION

The systematic production of high-quality software, which meets its specification, is still a major problem. Although formal specification methods have been around for a long time, only a few safety-critical domains justify the enormous effort of their application. The state of the practice, which relies on testing to force the quality into the product at the end of the development process, is also unsatisfactory. The need for effective test automation adds to this problem, because the creation and maintenance of the test ware is a source of inconsistency itself and is becoming a task of comparable complexity as the construction of the code.

II. AUTOMATICALLY GENERATING TESTCASES

Normally program verification tools do not prevent programming errors, they can quickly and cheaply identify oversights early in the development process, when an error can be corrected by a programmer familiar with the code. Moreover, unlike testing, some verification tools can provide strong assurances that a program is free of a certain type of error. These tools, in general, statically compute an approximation of a program's possible dynamic behaviors and compare it against a specification of correct behavior. These specifications often are easy to develop for language-specific properties such as avoiding null dereferences and staying within array bounds. Even when language properties are more difficult to express and check, they potentially apply to every program written in the language, so an investment in a verification tool can be amortized easily. On the other hand, specifications particular to a program, say of its abstractions or data types, may be difficult and expensive to develop because of the complexity of these mechanisms and the limited number of people who understand them. Also, as these specifications may apply to only one program, their benefits are correspondingly reduced. Program verification is unlikely to be widely used without cheaper and easier ways to formulate specifications. In the mined specifications it reflect normal rather than potential usage, dynamic specification mining observes executions to infer common properties. Typical examples of dynamic approaches include DAIKON for invariants or GK-tail for object states. A typical application of mined specifications is program understanding, and under the assumption that the observed runs represent valid usage; the mined specifications also represent valid behavior. The common issue of specification mining techniques, though, is that they are limited to the (possibly small) set of observed executions. If a piece of code is not executed, it will not be considered in the specification; if it is executed only once, there is no idea about alternative behavior. To address this problem, test case generation is used to systematically enrich dynamically mined specifications. Combined this way, both techniques benefit from each other: Dynamic specification mining profits from test case generation since additional executions can be observed to enrich the mined specifications. Test case generation, on the other hand, can profit from mined specifications, as their complement points to behavior that should be explored. The goal of this work is to explore the extent to which the quality of dynamically mined specifications can benefit from generated tests. How can we assess the benefits of such enriched specifications? For this purpose, they are used in static type state verification. Type state verification statically discovers illegal transitions. Its success depends on the completeness of the given specification: The more transitions are known as illegal, the more defects can be reported; and the more transitions are known as legal, the more likely it is that additional transitions can be treated as illegal. The enriched specifications are much closer to completeness than the initially mined specifications; and therefore, the static verifier should be much more accurate in its reports.

III. CODE COVERAGE ANALYSIS

Next the Code coverage analysis is selected in this research. It is the process of finding areas of a program not exercised by a set of test cases, creating additional test cases to increase coverage, and determining a quantitative measure of code coverage, which is an indirect measure of quality. An optional aspect of code coverage analysis is: Identifying redundant test cases that do not increase coverage. A code coverage analyzer automates this process. Coverage analysis is used to assure quality of set of tests, not the quality of the actual product. A coverage analyzer is not generally used when running set of tests through release candidate. Coverage analysis requires access to test program source code and often requires recompiling it with a special command.

IV. PROBLEM STATEMENT

On the testing side of software development process is now possible to automatically generate test cases that effectively explore the entire program structure; on the verification side, no it is possible to formally prove the absence of undesired properties for software as complex as operating systems. To push validation further, however, specifications of what the software actually should do are needed. Writing such specifications have always been hard and, so far, prohibited the deployment of advanced development methods. A potential alternative is specification mining i.e., extracting high-level specifications from existing code. In the context of this work, a specification is a model that is extracted from an existing program, rather than a manually written specification, which usually exists before the program is written. A typical application of mined specifications is program understanding, and under the assumption that the observed runs represent valid usage; the mined specifications also represent valid behavior. The common issue of specification mining techniques, though, is that they are limited to the (possibly small) set of observed executions. If a piece of code is not executed, it will not be considered in the specification; if it is executed only once, here alternative behavior is not known. To address this problem, test case generation is used to systematically enrich dynamically mined specifications. Combined this way, both techniques benefit from each other: Dynamic specification mining profits from test case generation since additional executions can be observed to enrich the mined specifications. Test case generation, on the other hand, can profit from mined specifications and code coverage, because their complement points to behavior and coverage should be explored. The goal of this work is to explore the extent to which the quality of dynamically mined specifications and coverage of testing can benefit from generated tests.

V. COMPARATIVE STUDY

In the research [1] DAIKON tool automatically traces the program execution but it have some issues they are: it address the granularity of instrumentation, which affects the amount of data gathered and time required for processing inferring loop invariants or relationships among local variables can require instrumentation at loop heads, at function calls. Limited to the set of observed executions and if a piece of code is not executed, it will not be considered in the specification; if it is executed only once derived in [2]. In research [4, 5, 6], without a priori specifications, it wants mutually enhance test generation and specification inference. At the same time, from a large number of generated tests, then it wants to have a way to identify valuable tests for inspection. Valuable tests can be fault-revealing tests or tests that exercise new program behavior. In research [8], the primary idea behind a formal method is that there is benefit in writing a precise specification of a system, and formal methods use a formal or mathematical syntax to do so. [11] Code coverage analysis provides metrics to quantify the degree of verification completeness Issues in code coverage were unable to relate specification coverage to branch coverage, the tool did not measure the sense of branch coverage that we were interested in. Using a different tool, the branch coverage of the suites used could be calculated, and then related to specification coverage accordingly. It also did not measure which specific bugs were detected by each individual suite. So it is possible that increasing specification coverage helps to detect a different set of bugs than are detected by increasing statement coverage. Finally we propose a dynamic specification mining it is a promising technique, but its effectiveness entirely depends on the observed executions. If not enough tests are available, the resulting specification may be too incomplete to be useful. By systematically generating test cases, our model explores previously unobserved aspects of the execution space. The resulting specifications cover more general behavior and much more exceptional behavior because of embedding a code coverage analysis.

VI. EXISTING SYSTEM

The main goal of system testing is to verify that requirements are successfully implemented into system under test. In order words, system testing assures that software system does what it is expected to do. On the testing side, it is now possible to automatically generate test cases that effectively explore the entire program structure; on the verification side, we can now formally prove the absence of undesired properties for software as complex as operating systems. The common issue of specification mining techniques, though, is that they are limited to the (possibly small) set of observed executions. If a piece of code is not executed, it will not be considered in the specification; if it is executed only once, we do not know about alternative behavior. To address this problem, test case generation is used to systematically enrich dynamically mined specifications. Combined this way, both techniques benefit from each other: Dynamic specification mining profits from test case generation since additional executions can be observed to enrich the mined specifications. Test case generation, on the other hand, can profit from mined specifications, as their complement points to behavior that should be explored. The goal of this work is to explore the extent to which the quality of dynamically mined specifications can benefit from generated tests. Dynamic specification mining is a promising technique, but its effectiveness entirely depends on the observed executions. If not enough tests are available, the resulting specification may be too incomplete to be useful.

Drawbacks

- Scalability and robustness of miners
- The limitation of choosing good training sets
- It might be necessary to exhaustively inspect each of the traces, which can be hundreds or thousands in number.

VII. PROPOSED METHODOLOGY

Currently, testing is still the most important approach to reduce the amount of software defects. Software quality metrics help to prioritize where additional testing is necessary by measuring the quality of the code. Most approaches to estimate whether some unit of code is sufficiently tested are based on code coverage, which measures what code fragments are exercised by the test suite. A specification mining-based diagnosis framework that can be used to simultaneously understand the error and locate it. Code coverage estimates the fraction of the execution paths of the code under test that are exercised by the test suite. Since code coverage metrics are simple to understand and efficient to compute, the use of code coverage metrics during the testing process is well-established. Furthermore, automatic tools have been created to help testers achieve high code coverage. The time spent in testing is mainly concerned with generating the test cases and testing them. The goal is to reduce the time spent in testing by reducing the number of test cases through the specification mining with the code coverage method. For this we have incorporated data mining techniques to reduce the number of test cases. Data mining finds similar patterns in test cases which helped us in finding out redundancy incorporated by automatic generated test cases.

Code Coverage

Code coverage analysis is the process of:

- Finding areas of a program not exercised by a set of test cases,
- Creating additional test cases to increase coverage, and Determining a quantitative measure of code coverage, which is an indirect measure of quality.

An optional aspect of code coverage analysis is to identifying redundant test cases that do not increase coverage. A code coverage analyzer automates this process. The use of coverage analysis to assure quality of set of tests, not the quality of the actual product. Generally a coverage analyzer is not used when running a set of tests through the release candidate. Coverage analysis requires access to test program source code and often requires recompiling it with a special command.

This work discusses the details is consider when planning to add coverage analysis to the test plan. Coverage analysis has certain strengths and weaknesses. A minimum percentage of coverage should be established to determine when to stop analyzing coverage. Coverage analysis is one of many testing techniques; we should not rely on it alone.

A. Algorithm Description

Advantages:

- Reuse familiar usage patterns, making them easier to understand.
- Focus on common usage, thus respecting implicit preconditions.
- Avoiding the meaningless tests.

STEP 1: INTRODUCES THE CONCEPT OF SPECIFICATION COVERAGE AND EXPLAINS THE PROCESS OF SPECIFICATION-BASED TESTING.

STEP 2: OUR TECHNIQUE IS FOR AUTOMATICALLY GENERATING A SPECIFICATION, SO THAT SPECIFICATION-BASED TESTING CAN BE CARRIED OUT ON A PROGRAM WITHOUT A FORMAL SPECIFICATION. WE THEN GIVE A FORMAL DEFINITION OF SPECIFICATION COVERAGE AS A METRIC.

STEP 3: GENERATING CODE COVERAGE WHICH HAS THREE DIFFERENT VARIETIES OF CODE COVERAGE: STATEMENT COVERAGE, BRANCH COVERAGE, AND DEFINITION-USE (DEF-USE OR DU) COVERAGE.

STEP 4: SPECIFICATION COVERAGE IS THE SET OF INVARIANTS PRODUCED BY TAUTOKO USING THE ENTIRE TEST POOL.

STEP 5: BUG DETECTION IS THE RATE OF A TEST SUITE IS THE RATIO OF THE NUMBER OF FAULTY VERSIONS DETECTED TO THE NUMBER OF FAULTY PROGRAM VERSIONS. A TEST SUITE DETECTS A FAULT, IF THE OUTPUT OF THE FAULTY VERSION DIFFERS FROM THE OUTPUT OF THE CORRECT VERSION, WHEN BOTH ARE RUN OVER THE TEST SUITE.

B. Implementation

This work makes the following contributions:

- A new approach is introduced, called specifications mining, for learning formal correctness specifications. Since specification is the portion of program verification still dependent primarily on people, automating this step can improve the appeal of verification and help improve software quality.
- The observation is used that common behavior is often correct behavior to refine the specifications mining problem into a problem of probabilistic learning from execution traces.
- A practical technique is developed and demonstrated for probabilistic learning from execution traces. This technique reduces specification mining to the problem of learning regular languages, for which off-the-shelf learners exist.
- Finally the *Code coverage analysis* is the process of finding areas of a program not exercised by a set of test cases, Creating additional test cases to increase coverage, and Determining a quantitative measure of code coverage, which is an indirect measure of quality.

a. Creating a Crossword Application

In this phase a crossword application is created with the questions and answer. The question will be display on the top of the application if the answer is generated as wrong the particular letter will be marked in red color, if it is right the letter will be displayed in black color. This is applied for all the individual letters. The answer should be started from the number which is corresponds to the question number.

b. Test Case Generation Initial Model

In this phase a java program is taken as an input and type state is generated for initial model. Mining type state automatically by obtaining such type states from program by developing an object behavior model that captures the object at runtime. A behavior model for an object is a nondeterministic finite state automaton where states are labeled with the values of fields that belong to object, and transitions occur when a method invoked on particular object changes the state. To mine such models from a program run, the object behavior model instruments the program such that the values of all fields from an instance of the class are captured after every method call. Each set of field values uniquely describes the state of the object, and the sequence of states and method calls uniquely describes the object behavior model for the observed instance of the class. The advantage of using field values to label states is that equivalent object states are easy to detect, since they have the same labeling. The challenge with using concrete field values is that the number of states in a model may become very large, and the models are difficult to compare. The value of variable is the object identifier for the particular object. In another run of the program, this value might be different due to scheduling, and hence the states would be different in both models. However, this difference is not important both for the behavior of the class. Hence, instead of using concrete values, our object uses abstract values: Complex fields are mapped to null or not null, numerical fields are mapped to less than, equal to, or larger than zero, and Boolean fields remain unchanged. Finally we create test case for the object behavior models.

c. Test Case Generation Enriched Model

In this phase the test case is developed for enrich model, and needs to contain all relevant states and transitions for all methods in all states. To test object behavior model, it is run on a set of projects and mined type states from the test suite executions for a set of interesting classes. Unfortunately, for the investigated classes the thing is found that the most type states only contained a fraction of all transitions. In particular, most type states were missing transitions for failing methods, which renders mined type states useless for type state verification. Most defects due to wrong usage of a class raise exceptions and are therefore easy to detect and fix. Thus, a specification miner tool will seldom record misuse and exceptions when tracing normal application executions. Unfortunately, the same problem of missing exceptions is observed when tracing test suites. Most developers do not test for exceptions. One explanation for this is that triggering an exception often only covers a few lines. To generate a complete model, lots of tests are required. Usually, developers do not have enough time to write so many tests. Also, developers tend to skip tests which they consider to be too obvious or are convinced that they should work. One way to approach this problem is to use test case generation to create new tests that execute previously unknown states and transitions. The general idea of combining specification mining with test case generation the original idea to generate tests specifically targeted at enriching type state automata. In this phase, the enriched type state features and generating the test cases are mainly considered.

ALGORITHM

```
Algorithm: Enrich Typestate Automaton
Require: Test Suite T = (t1; . . . ; tn)
Require: Initial typestate $M_{init} = (V_{init}; E_{init})$
Require: Methods to investigate M
Ensure: Enriched Typestate $Mf_{inal} = (V_{final}; E_{final})$
1: procedure ENRICH (T; $M_{init}$, M)
    2: T’ = □ T’ is the set of new tests
    3: for all s □ $V_{init}$\{start, ex} do
        4: $M_s \leftarrow$ {Methods invoked in s}
        5: for all m ∈ Mn\Ms do
            6: if hasParameters(m) then
                7: $S_m \leftarrow$ {^s ∈ $V_{init}$| j ∃(^s; ^s;m) 2 Einit}
                8: for all ^s ∈ Sm do
                    9: p $\leftarrow$ getPath(T;$M_{init}$; s; ^s)
                    10: if length(p) < 1 then
                        11: T’:add(suppressCallsOn(T; p))
                    12: end if
                13: end for
            14: else
                15: $R_s \leftarrow$ {(s; s’; n) ∈ $E_{init}$}
                16: for all t ∈ Rs do
                    17: T’:add(appendCall(T;$M_{init}$, t, m))
                18: end for
            19: end if
        20: end for
    21: end for
    22: $M_{final} \leftarrow M_{init}$
```

```
23: for all t ∈ T' do
24: $M_{new} \leftarrow$ run(t)
25: $M_{final} \leftarrow$ merge ($M_{new}$,$M_{final}$)
26: end for
27: end procedure
```

d. Test Case Using Enriched Model Generate

In this phase the setup for the test case generation approaches ENRICH and GENERATE is as follows: For the ENRICH approach, the initial model from the subject test suite is used and generates test cases such that every method of the subject class is called in every state of the initial model. For the GENERATE approach, we first generate a test suite for each subject targeting branch coverage. Then, models are created based on the resulting minimized test suites, and iteratively refine it by deriving new test cases. The number of iterations is set limited and a maximum number of test cases per iteration is used to keep the overhead of trace generation and model learning in reasonable bounds.

e. Identifying Code Coverage

Code coverage analysis is the process of:

- Finding areas of a program not exercised by a set of test cases,
- Creating additional test cases to increase coverage, and
- Determining a quantitative measure of code coverage, which is an indirect measure of quality.
  An optional aspect of code coverage analysis is:
- Identifying redundant test cases that do not increase coverage.
  A code coverage analyzer automates this process.

Coverage analysis is used to assure quality of set of tests, not the quality of the actual product. A coverage analyzer is not used when running your set of tests through the release candidate. Coverage analysis requires access to test program source code and often requires recompiling it with a special command.

Coverage analysis has certain strengths and weaknesses. It must be choose from a range of measurement methods. A minimum percentage of coverage should be established to determine when to stop analyzing coverage. Coverage analysis is one of many testing techniques. Code coverage analysis is sometimes called test coverage analysis. The two terms are synonymous. The academic world more often uses the term "test coverage" while practitioners more often use "code coverage". Likewise, a coverage analyzer is sometimes called a coverage monitor.

f. Performance

The model is evaluated by comparing the initial model, enrich model (ENRICH), enrich model (GENERATE) and complete model.

True Positives (TP)

Reported lists the number of defects for which the verifier reports a violation. Actual gives the number of cases where the reported method call exactly matches the call that raises the exception. For all numbers of reported errors, higher values are better.

False positives (FP)

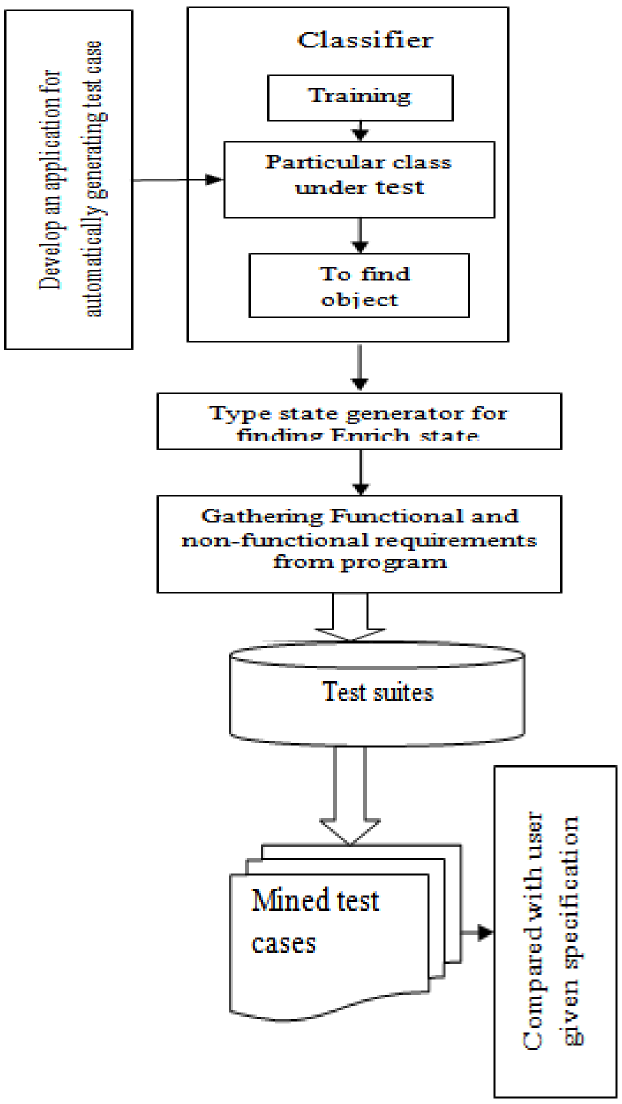
This analysis may miss method calls which potentially cause false positives or negatives. Information of different paths through a method is merged together. Thus, the analysis is path insensitive, which may cause false positives. Whenever the analysis is unable to determine the state of an object, it simply assumes that the object can be in any possible state. This may again generate false positives.

Code coverage

Code coverage analysis is the process of:

- Finding areas of a program not exercised by a set of test cases,
- Creating additional test cases to increase coverage, and
- Determining a quantitative measure of code coverage, which is an indirect measure of quality.

Fig:9.1 Producing Test Cases Robotically By Using Specification Mining And Code Coverage


Develop an application for automatically generating test case
Classifier
Training
Particular class under test
To find object
Type state generator for finding Enrich state
Gathering Functional and non-functional requirements from program
Test suites
Mined test cases
Compared with user given specification

C. BLOCK DIAGRAM

- A developer starts building an application and uses classes from a library that are unknown to them.
- To help the developer avoid bugs due to incorrect usage of those classes, her IDE supports lightweight type state verification. Whenever the developer changes a method that uses classes of for which a specification is available, the IDE launches the type state verifier.
- The verifier then analyzes all changed methods and looks for incorrect usage of classes; if it finds a violation, it is presented to the user. Obviously, many defects (true positives) and report as few false alarms (false positives) as possible are found.

VIII. CONCLUSION

This paper proposed a scalable dynamic specification-mining and code coverage analyzer for generating test cases; it entirely depends on the observed executions of the program, if not enough tests are available; the resulting specification may be too incomplete to be useful. By systematically generating test cases proposed research explores previously unobserved aspects of the execution space. The resulting specifications cover more general behavior and much more exceptional behavior by embedding the concept of code coverage analyzer. Despite the improvements by our approach, dynamic type state specification mining still produces a considerable number of false positives, which are partly due to technical limitations of our type state verifier implementation. The future direction is to reduce the false positive rate by implementing the more accurate dynamic type state verifier models to mine the appropriate specifications in the program to generate test cases automatically.

REFERENCES

[1]. M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin "Dynamically Discovering Likely Program Invariants to Support Program Evolution"

[2]. Lorenzoli D, Mariani L, and Pezze M, "Automatic Generation of Software Behavioral Model

[3]. Reliable Mining of Automatically Generated Test Cases from Software Requirements Specification (SRS) Lilly Raamesh and G. V. Umas," Proc. 30th Int'l Conf. Software Eng., pp. 501-510, 2008

[4]. Mutually Enhancing Test Generation and Specification Inference Tao Xie and David Notkin

[5]. Ammons G, Bodı´k R, and Larus J, "Mining Specifications," Proc. 29th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages, pp. 4-16, Jan. 2002

[6]. Static Specification Mining Using Automata-Based Abstractions Sharon Shoham Eran Yahav Stephen Fink Marco Pistoia

[7]. Online Testing with Model Programs Author: Margus Veanes, Colin Campbell and Wolfram Schulte

[8]. Using Formal Specifications to Support Testing R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick

[9]. Davide Lorenzoli, Leonardo Mariani and Mauro Pezzè, "Automatic Generation of Software Behavioral Model"

[10]. Invariant-Based Automatic Testing of AJAX User Interfaces Ali Mesbah and Arie van Deursen

[11]. Expression Coverability Analysis: Improving code coverage with model checking Graeme D. Cunningham, Institute for System Level Integration, Livingston, Scotland

[12]. Code Coverage Based Technique for Prioritizing Test Cases For Regression Testing

[13]. K.K.Aggrawal, Yogesh Singh, A.Kaur

[14]. Towards a Framework for Generating Tests to Satisfy Complex Code Coverage in Java Pathfinder Matt Staats

[15]. Specification Coverage as a Measure of Test Suite Quality Michael Harder Benjamin Morse Michael D. Ernst

[16]. Evaluation of Structural Testing Effectiveness in Industrial Model-driven Software Development Mahdi Sarabi

Books:

[17]. Cay Horstmann, "Big Java", John Wiley & Sons, 4th Edition, 2009.

[18]. John Ferguson Smart, "Java Power Tools", O'Reilly Media, Inc., 2008.

[19]. Lee Copeland, "Practioners guide to software testing".

[20]. David Lo, Khoo Siau Cheng, Jiawei Han, "Mining Software Specifications: Methodologies and Applications".

[21]. Paulo Borba, Ana Cavalcanti, Augusto Sampaio, "Testing Techniques in Software Engineering".