International Journal of Computer Science and Software Engineering (IJCSSE), Volume 4, Issue 10, October 2015
ISSN (Online): 2409-4285 www.IJCSSE.org Page: 256-263

Coverage Criteria for Search Based Automatic Unit Testing of Java Programs

Ina Papadhopulli[1] and Elinda Meçe[2]

[1, 2] Department of Computer Engineering, Polytechnic University of Tirana, Tirana, Albania

[1]*ipapadhopulli@fti.edu.al*, [2]*ekajo@fti.edu.al*

ABSTRACT

Genetic Algorithms are among the most efficient techniques to automatically generate unit test cases today. The aim of automatically unit testing is to generate test suites which achieve high coverage and to minimize the length of the test suites. The chosen coverage criterion guides the search and directly affects the results. Among the existing criteria the mutation criterion is the strongest, but it is difficult to be applied practically. Combining several criteria is possible since usually they aim at different parts of the unit under test. In this article we have used EvoSuite with different configurations to automatically generate test suites for six open source projects in order to investigate how the combination of coverage criteria influences the coverage achieved and the test suite length. Based on our experiments we find that the usage of multiple criteria increases the overall coverage and mutation score with the cost of a considerable increase in test suite length.

Keywords: *Structural Testing, Test Case Generation, Search Based Software Testing, Coverage Criteria, Mutation Score.*

1. INTRODUCTION

Software is becoming every day more complex and is being incorporated more and more into a different number of systems. All the software development phases has been adapted to produce these complex software systems, but especially the testing phase is of critical importance and testing thoroughly today's software systems is still a challenge. Almost 30-60% of the resources are spend on the testing process [13]. Testing is not longer seen as an activity that starts only after the coding phase. It is a well-known fact that it is a lot more expensive to correct defects that are detected during later system operation. Considering past experiences, inadequate end ineffective testing can result in social problems and human/financial losses. Even though according to [20] testing is still largely ad hoc, expensive and unpredictably effective. Dealing with software with thousands or even millions of lines of code during testing is practically impossible so there is a growing necessity to automate this process as much as possible. Based on the level of artificial intelligence available today fully automating the testing process is impossible. We can say that automatic testing is not the automation of testing process but it is the usage of automation to support this process.

Testing consists of two phases: generating the test cases and generating the test oracle to evaluate the results produced by the execution of the software for a given test case. Automatically generating the test oracle is still a challenge and there exists few research publications regarding this topic [12]. In contrast there exists a lot of research effort for the generation of test cases automatically. The research publications are very promising and contain evaluations of the techniques even for real software.

This article is focused on structural testing at the unit level. According to [9] more than 50% of the research is focused on unit testing. Every module of the software must be tested before proceeding to the other stages of the development cycle. There exist two main approaches for automatic structural testing: Symbolic Execution (SE) [17] and Search-Based Software Testing (SBST). Even though these approaches have been published more than three decades ago, there is still interest in both of them. SE and SBST are theoretically completely different but in the latest years there exist attempts to combine both approaches [18][24] in order to develop tools which take advantage of this combination to overcome the weaknesses each approach separately has.

In this article we are focused in the SBST technique [22][23]. Search Based Software Engineering (SBSE) is an engineering approach in which optimal or near optimal solutions are sought in a search space of candidate solutions. The search is guided by a fitness function that distinguishes between better and worse solutions. SBSE is an optimization approach and it is suitable for software testing since test case generation is often seen as an optimization or search problem. It was

first introduced in 1976 with the work of Miller and Spooner. The first steps towards automation were in 1990. It has been applied on all software engineering phases, but according to [19], 59% of the overall SBSE literature is concerned with Software Engineering applications relating to testing (SBST). If we refer to the classification of testing techniques based on the strategy for selecting test data, most of the research for SBST has been spend in structural testing. Genetic algorithms (GAs) are among the most frequently applied search-based optimization methods in test data generation. Since SBST techniques are heuristic by nature, they must be empirically investigated in terms of how costly and effective they are at reaching their test objectives and whether they scale up to realistic development artifacts. However, approaches to empirically study SBST techniques have shown wide variation in the literature.

Choosing the fitness function affects the results achieved by SBST. The fitness function is a mathematical representation of the coverage goal the search should achieve. There are different coverage goals each of them aims at covering certain parts of the unit under test. These different coverage criteria verify the quality of a test suite. There is the possibility to use a combination of coverage criteria to guide the search. In this case the fitness function of the combined criteria is a mathematical combination of each fitness function depending on the fact whereas the constituent criteria are conflicting or non-conflicting. The scope of this paper is to study how does search-based testing scale to combinations of multiple criteria.

The rest of this paper is organized as follows: In the second section we present an overview of SBST mainly focused on GAs. In the third section we explain in what unit testing of java programs consists and in the fourth section the main characteristics of EvoSuite testing tool are presented. In the fourth section we list and explain the most used coverage criteria for unit testing. In the fifth section the experimental setup is presented and in the sixth section the results achieved are discussed. We conclude finally with the conclusions we have come preparing and accomplishing this study.

2. SBST

A. Overview of SBSE

SBSE is a meta-heuristic engineering approach. SBST is one example of SBSE. According to [14] SBST has been used to automate the testing process in the following areas:

- The coverage of specific program structures, as part of a structural, or white-box testing strategy;
- The exercising of some specific program feature, as described by a specification;
- Attempting to automatically disprove certain grey-box properties regarding the operation of a piece of software, for example trying to stimulate error conditions, or falsify assertions relating to the software's safety;
- The verification of non-functional properties, for example the worst-case execution time of a segment of code.
- The most important search-based optimization methods used for test automation are genetic algorithms, hill climbing, ant colony optimization and simulated annealing, which have successfully been applied in solving a wide range of software testing problems at different testing levels. Genetic algorithms (GAs) are among the most frequently applied search-based optimization methods in unit test data generation [15].

B. Genetic Algorithms

Genetic Algorithms (GAs) are inspired by natural evolution. They were first introduced by Holland in 1975. Today GAs are used for optimization in testing real life applications. The most important components in GA are:

- Representation of individuals: genotype (the encoded representation of variables) to phenotype (the set of variables themselves) mapping
- Fitness function: a function that evaluates how close an individual is to satisfy a given coverage goal
- Population: the set of all the individuals (chromosomes) at a given time during the search
- Parent selection mechanism: selecting the best individuals to recombine in order to produce a better generation
- Crossover and mutation: the two types of recombination used to produce new individuals
- Replacement mechanism: a mechanism which replace the individuals with the lowest fitness function in order to produce a better population.

C. How does the GA work?

The space of potential solutions is searched in order to find the best possible solution. This process is started with a set of individuals (genotypes) which are generated randomly from the whole population space

(phenotype space). New solutions are created by using the crossover and mutation operators. The replacement mechanism selects the individuals which will be removed so that the population size does not exceed a prescribed limit. The basis of selection is the fitness function which assigns a quality measure to each individual. According to the fitness function, the parent selection mechanism evaluates the best candidates to be parents in order to produce better individuals in the next generation. It is the fitness function which affects the search towards satisfying a given coverage criteria. Usually the fitness function provides guidance which leads to the satisfaction of the coverage criterion. For each individual the fitness is computed according to the mathematical formula which represents how close is a candidate to satisfy a coverage goal, e.g. covering a given branch in the unit under test. GAs are stochastic search methods that could in principle run for ever. The termination criterion is usually a search budget parameter which is defined at the beginning of the search and represents the maximum amount of time available for that particular search.

3. UNIT TESTING OF JAVA PROGRAMS

Regarding the classification of testing based on the level of testing, unit testing is the lowest level. During unit testing a single program unit (usually a procedure or module) is tested and the code is completely visible. Testing Java programs at the unit level means generating test cases to cover a Java class [2]. Usually the JUnit framework is used to write the test cases. The definitions of test suite and test cases are:

- A test suite **T** is a set of test cases **t**: $T = \{t_1, t_2, \ldots, t_n\}$
- Test case **t** in Java unit testing is a program that executes the Class Under Test (CUT) and consists of a sequence of statements $t = \{s_1, s_2, \ldots, s_m\}$

The statements for Java unit test cases are:

1. *Primitive statements: declaration of variables e.g. int a = 15;*
2. *Constructor statements: construction objects of any given class e.g. String s = new String("Test");*
3. *Method statements: calling the methods of any given class e.g. char b = s.charAt(2);*
4. *Field statements: accessing the fields of any given class e.g. int c = ob.size;*
5. *Assignments statements: assign values to the fields of any given class e.g. ob.size = 17;*

4. EVOSUITE

In order to measure the effect of combining different coverage criteria for automatic unit test case generation we have used in this article EVOSUITE [8] which is a tool for automatic unit testing of Java programs. Some of EvoSuite's main characteristics are listed below:

- Usage: Command Line or Eclipse plugin
- Input: Bytecode of the target class
- Output: JUnit test cases
- Technology: Genetic Algorithms
- Open Source
- Operating systems where can be used: Mac, Linux
- Official website: www.EvoSuite.org

A common approach in the literature is to generate a test case for each coverage goal whereas EvoSuite uses whole test suite generation which means that the entire test suite is optimized towards satisfying all goals at the same time. According to the authors of EvoSuite [7], whole test suite generation is more effective than trying to satisfy one goal at a time. The reason is that some coverage goals are infeasible or more difficult to satisfy and in the testing scenario where the amount of resources is limited trying to accomplish these goals makes the distribution of the resources not equal. According to [6][7] whole test suite generation achieves higher coverage than single branch test case generation and produces smaller test suites than single branch test case generation.

5. COVERAGE CRITERIA

A. Types of Coverage Criteria

Automatic unit testing is guided by a structural coverage criterion. There exist many coverage criteria in literature, each of them aims at covering different components of a CUT. Below is a list of coverage criteria for structural testing for Java programs.

1. Line Coverage: the goal is to execute all the lines (non-comments line) of the CUT. It is perhaps the most used criterion in practice.
2. Branch Coverage: the goal is to cover all the branches of the CUT. Covering a branch means executing the branch at least twice by taking once the true branch and twice the false branch.
3. Modified Condition Decision Coverage: [21] the goal is that every condition within a decision has taken on all possible outcomes at least once, and every condition has been shown to independently affect the decision's outcome (the condition of

interest cannot be masked out by the other conditions in the decision). This criterion is stronger than branch coverage.

4. Mutation: the goal is to kill all the mutants. Killing a mutant means that the output of a test case on the program is different from the output of the test case on the mutant.
5. Weak Mutation: the goal is to weakly kill all the mutants. Weakly killing a mutant means that the internal state of the program immediately after execution of the mutated component must be incorrect [16]. This criterion is weaker than Mutation.
6. Method coverage: the goal is to call all the methods of the CUT by executing the test suite.
7. Top-level Method Coverage: the goal is to call all the methods of the CUT directly which means that a call to the method appears as a statement in a test case of the test suite. This criterion is stronger than Method Coverage.
8. No-Exception Top Level Method Coverage: the goal is to call all the methods from the test suite via direct invocations, but with parameters that lead to a normal execution of the methods (not generating exceptions). This criterion is stronger than Top-level Method Coverage.
9. Direct Branch Coverage: the goal is that each branch in a public method of the CUT to be covered by a direct call from a unit test, but makes no restriction on branches in private methods.
10. Output Coverage: the goal is to call all the methods with parameters that cover all the different types of output the method can return. E.g. if the method's type is Boolean the method should be called twice in order to return once a true value and once a false value.
11. Exception Coverage: the goal is to cover all the feasible undeclared exceptions (if exceptions are unintended or if thrown in the body of external methods called by the CUT).

Exception Coverage, Method Coverage, Top-level Method Coverage, No-exception Top-level Method Coverage have a fitness function which provides no guidance during the search [1].
Mutation criterion is considered the gold criterion in research literature [11]. This criterion is difficult to apply and computationally expensive and it is practically only used for predicting suite quality by researchers. The reasons why mutation testing cannot be used for testing real software are:

1. The number of mutants for a given system can be huge and it is very expensive to run the test against all the mutants.
2. Equivalent mutants which are mutants that only change the program's syntax, but not its semantics and thus are undetectable by any test

The criteria implemented in EvoSuite are [1]: Line Coverage, Branch Coverage, Direct Branch Coverage, Output Coverage, Weak Mutation, Exception Coverage, Top-level Method Coverage, No-exception Top-level Method Coverage.
The most used criterion is branch coverage [10] but even though it is an established default criterion in the literature, it may produce weak test sets, and software engineering research has considered many other criteria. Another option is to combine different coverage criteria.

B. Combination of Coverage Criteria

The search is guided by the coverage criteria the resulting test suite should satisfy. The coverage criteria are translated to a mathematical formula which is the fitness function whose work is to evaluate the individuals during the search. It is possible to use more than one criterion to guide the search. In this case if the combined criteria are non-conflicting than the resulting fitness function is a linear combination of each fitness functions. The aim of this work is to study how does search-based testing scale to combinations of multiple criteria for unit testing in Java programs.

6. EXPERIMENTAL EVALUATION

In this work we aim to answer the following research questions:

- RQ1: How does the combination of multiple criteria affect the coverage of each criterion?
- RQ2: How does the combination of all the criteria affect the mutation score of the suite?
- RQ3: Which of the criteria (except Weak Mutation) used separately achieves the highest mutation score?
- RQ4: How does the number of mutants of the CUT affect the mutation score?
- RQ5: How do multiple criteria affect the number of suite's test cases and their size?
- RQ6: How does the adding of weak mutation criterion affect the size of the test cases?

Table 1: Name of the Projects Selected for the Experiments, the Number of Classes They Contain, the Downloading Source

Project Name	*No. of Classes*	*Source*
MathPareser	48	SourceForge
MathQuickGame	25	SourceForge
java.util.Regex	92	jdk 1.8.0/src
Refactoring	87	SourceForge
Library	22	CodeCreator.org
StudentManagementSystem	24	CodeCreator.org
Total	***298***	

7. EXPERIMENTAL SETUP

A. System Characteristics

For the experiments we used a desktop computer running Linux 32 bit Operating System, 1 GB of main memory and a Intel Core 2 Duo CPU E7400 2.8GHz x 2 Processor.

B. EvoSuite Usage

We run EvoSuite from the command line (Eclipse plugin is not currently available for Linux OS).

C. Subject Selection

Selecting the classes under test is very important since this selection affects the results of the experiments. We chose open source software. We selected 6 projects with a total of 298 testable classes. The projects and their sources are listed in Table 1. Three of the projects were downloaded from SourceForge which is today the greatest open source repository. Two projects were downloaded from CodeCreator. The remaining project is a Java library.

Due to the limited physical resources and the large amount of time to run all the experiments on a single class we chose randomly 150 classes from the six projects listed in table 1. To overcome the randomness of the genetic algorithms each experiment is repeated 5 times. For 6 classes, EvoSuite quits without generating any output. It is not the scope of this work to investigate the reasons why this happened. It is relevant to mention here that the results were in many cases affected by the *security manager* of EvoSuite which restrict the test execution of the test suite in order to protect the system against unsafe operations (e.g. access the file system).

Parameters of GA: During the experiments we let the parameters of the GA to their default values, whereas regarding the search budget depending on the experiment values of 1min or 5 min were used. The values of four of the most relevant parameters are:

- Population size: 50 test suites
- Chromosome length: 40 test cases
- Probability of mutation: 0.75
- Probability of crossover: 0.75

Defining the parameters of GAs to obtain the optimal results is difficult and a lot of research effort is dedicated to this topic [3][4].

Table 2: Coverage Results for Each Configuration, Average of All Runs for All Cuts

Criterion	Line	Branch	Exception	Weak Mutation	Output	Top-Level	Method No Exception	Direct Branch
ALL (1 min)	59.7	52.6	83.7	62.4	47.9	93.4	89.6	49.8
ALL (5 min)	60.7	53.4	83.9	69.7	48.2	93.8	90.7	53.1
Only one (5 min)	61.1	53.5	83.9	74.3	48.6	81	80.5	52.8

7. RESULTS AND DISCUSSION

RQ1: How does the combination of multiple criteria affect the coverage of each criterion?

Experiment 1: For each of the classes we run EvoSuite with the following configurations:

1. All criteria with search budget of 1 min
2. All criteria with search budget of 5 min
3. Each criterion separately with search budget of 5 min

The results of the experiments are presented in table 2.

Based on the results we can say that running EvoSuite for 5 min achieves higher coverage than running EvoSuite for 1 min for each of the criterion. This result was expected since the individuals improve during the search and more time results in better solutions.

Regarding Line, Branch, Weak Mutation, Output coverage criterion using the ALL combination results in

lower coverage since in this case the fitness function is a combination of all fitness functions and aims at covering different parts of the program (e.g. not just the lines). The biggest difference is for Weak Mutation. For Method No Exception coverage using the ALL combination increases the coverage with 13% since the search here takes advantage of the guidance provided from the other criteria. The same is for Top-Level Method criterion.

RQ1: For criteria whose fitness guides the search, the performance between the combination of all criteria and each criterion separately converges nearly to the same value. For criteria with low guidance the ALL - combination increases the performance.

RQ2: How does the combination of all the criteria affect the mutation score of the suite?

RQ3: Which of the criteria (except Weak Mutation) used separately achieves the highest mutation score?

Experiment 2: For each of the classes we run EvoSuite with the configurations:

1. All criteria with search budget of 5 min
2. All criteria (except Weak Mutation) with search budget of 5 min
3. Each criterion separately with search budget of 5 min

Since mutation score is the measure used in the strongest criterion (Mutation Coverage), here we have used it to measure the quality of the generated test suite. The results of the average mutation scores of each configuration are given in table 3.

From the results we can say that the criterion which achieves the higher mutation score is Weak Mutation and this was expected since the goal in this criterion is to weakly kill all the mutants. Using the ALL combination results in higher mutation score than each criterion separately (except Weak Mutation). Exception coverage results in the lowest mutation score. Based on the experiments we conclude that if it is not possible to use Mutation Criterion than the next best criterion to achieve high mutation scores is branch coverage. Given the achieved mutation scores we can say that automatically generated test suites are still far from achieving high mutation scores despite the chosen criterion.

RQ2: Given enough time the combination of all criteria achieves higher mutation score than each criterion separately (except Weak Mutation).

RQ3: In our experiments the next best criterion to achieve high mutation scores is branch coverage.

RQ4: How does the number of mutants of the CUT affect the mutation score?

Experiment 3-4: For each of the classes we run EvoSuite with the configuration:

1. Mutation criterion with search budget of 5 min

Table 3: Mutation Scores for Each Configuration, Average of All Runs for All Cuts with Search Budget of 5 Min

Criterion	Line	Branch	Exception	Weak Mutation	Output	Top-Level	Method No Exception	Direct Branch	ALL(without Weak Mutation)	ALL (5 min)
Mutation Score	15.6	25.8	0.2	28.3	16.6	23	15.3	21.2	24.5	26.1

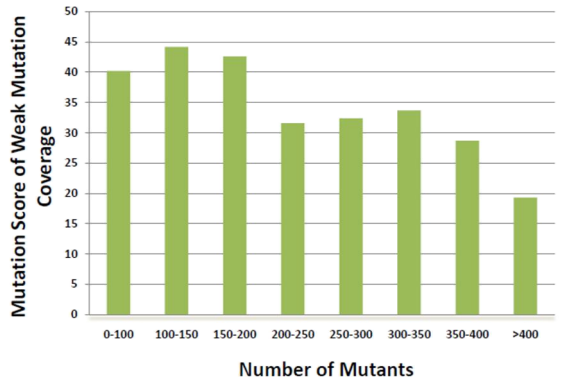

Fig. 1. Dependency of the mutation score achieved by Weak Mutation Coverage to the number of mutants

Since the criterion implemented in EvoSuite which achieves the higher mutation score is weak mutation, in Fig. 1 is presented the dependency of the mutation score of Weak Mutation Coverage to the number of mutants. Because of the fact that the number of mutants is not the only factor that affects the mutation score we did not expect a strong dependency between them in the lowest intervals, but if the number of mutants is high it obviously affect the mutation score achieved.

RQ4: Given the limited search budget the number of mutants in the CUT affects the mutation scores achieved by EvoSuite.

RQ5: How does multiple criteria affect the size of the test cases?

RQ6: How does the adding of weak mutation criterion affect the size of the test cases?

Here we refer to the size of a test case as the number of statements after the minimization phase (without assertions).
Automatically generated JUnit tests need to be manually checked in order to detect faults because automatic oracle generation is not possible today. This is the reason why not only the achieved coverage of the generated test suite is important, but the size of the test suite is of the same importance [5].
Experiment 5-6: For each of the classes we run EvoSuite with the configurations:

1. All criteria with search budget of 5 min
2. All criteria (except Weak Mutation) with search budget of 5 min
3. Each criterion separately with search budget of 5 min

In 35 runs of EvoSuite the minimization phase timeout and these runs were not taken into consideration. It is not the scope of this work to investigate why the minimization phase did not conclude properly.
The number of tests in the test suite is not relevant in respect to the size of the test suite, because having many short size tests is not a problem for the tester who is detecting faults. The overall size of the suite is very important and from the results we can say clearly that using the ALL combination results in test suites whose number of lines of code is bigger than using each of the criteria separately. This is a considerable drawback in the use of several combined criteria since the test cases became more complex and for the tester to deal with them becomes more difficult.
RQ5: Using all the criteria increases the test suite size by more than 50% that the average test suite size of each constituent criterion used separately.
RQ6: Adding weak mutation criterion increases the test suite size approximately with 20%.

7. CONCLUSIONS

This paper concerns the coverage criteria used to guide the search during automatic unit test generation. Combining several criteria is possible and this work is focused on the effect of combination to the coverage achieved and the size of the test suite generated. The usage of multiple criteria increases the overall coverage and mutation score with the cost of a considerable increase in test suite length. Based on the average coverage achieved by using only a criterion or several criteria we can say that a lot of improvements need still to be made to enable the usage of automation to really support unit testing. To increase the coverage there is still necessary to find different fitness functions or to adapt them during optimization. There is the need to further investigate the conditions that make certain problems more difficult to be optimized with a genetic algorithm.

Table 4: suite Size for Each Configuration, Average of All Runs for All Cuts with Search Budget of 5 Min

Criterion	Line	Branch	Exception	Weak Mutation	Output	Top-Level	Method No Exception	Direct Branch	ALL(without Weak Mutation)	ALL (5 min)
Coverage	61.1	53.5	83.9	74.3	48.6	81	80.5	52.8	59.7	69.3
Mutation Score	15.6	25.8	0.2	28.3	16.6	23	15.3	21.2	24.5	26.1
No. of tests	12.3	12.9	6	14.2	7.5	12.4	11.7	10	14.2	15.1
Test Suite size	14.2	15.7	6.7	19.4	10	10.3	12.4	10.2	22.3	27.8

REFERENCES

[1] "J. Miguel Rojas, J. Campos1, M. Vivanti, G. Fraser, A. Arcuri, "Combining Multiple Coverage Criteria in Search-Based Unit Test Generation" in Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 436-439, 2011
[2] G. Fraser, A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software". In Proceedings of ESEC/FSE, 2011.
[3] A. Aleti, L. Grunske, "Test Data Generation with a Kalman Filter-Based Adaptive Genetic Algorithm". Journal of Systems and Software, 2014.
[4] A. E. Eiben, S. K. Smit, "Parameter tuning for configuring and analyzing evolutionary algorithms". Journal: Swarm and Evolutionary Cmputation, pages 19-31, 2011.
[5] G. Fraser, A. Arcuri, "Handling test length bloat". In Proceedings of ICST, 2013.
[6] G. Fraser, A. Arcuri, "Evolutionary Generation of Whole Test Suites". In Proceedings of QSIC, 2011.

[7] G. Fraser, A. Arcuri, "Whole Test Suites Generation". In Proceedings of QSIC, 2012
[8] G. Fraser, A. Arcuri, "EvoSuite at the SBST 2015 Tool Competition". In Proceedings of International Conference in Software Engineering (ICSE) 2015
[9] I. Papadhopulli, N. Frasheri, "A Review of Software Testing Techniques", In Proceedings of RCITD, 2014.
[10] K. Lakhotia, P. McMinnb, M. Harman, "An empirical investigation into branch coverage for C programs using CUTE and AUSTIN". Journal of Systems and Software, 2010
[11] G. Fraser, A. Arcuri, "Achieving Scalable Mutation-based Generation of Whole Test Suites". Empirical Software Engineering 2014.
[12] G. Fraser, A. Zeller, "Mutation-Driven Generation of Unit Tests and Oracles," IEEE Transactions on Software Engineering, 2012.
[13] S. Ali, H. Hemmati, "A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation," IEEE Transactions on Software Engineering, 2010.
[14] P. McMinn, "Search-based Software Test Data Generation: A Survey", Software Testing, Verification and Reliability, pp. 105-156, June 2004.
[15] A. Aleti, L. Grunske, "Test Data Generation with a Kalman Filter-Based Adaptive Genetic Algorithm". Journal of Systems and Software, 2014.
[16] P. Amann, J. Offut, "Introduction to Software Testing", 2008
[17] C. Cadar, K. Sen, "Symbolic Execution for Software Testing: Three Decades Later". Communications of ACM, pages 82-90, 2013
[18] J. Galeotti, G. Fraser, A. Arcuri, " Extending a Search-Based Test Generator with Adaptive Dynamic Symbolic Execution", In Proceedings of International Symposium on Software Testing and Analysis (ISSTA) 2014
[19] M. Harman, S. A. Mansouri, Y Zhang , "Search-based software engineering: Trends, techniques and applications". Journal CSUR, 2012
[20] A. Bertolino, "Software testing research: Achievements, challenges, dreams". In Future of Software Engineering, 2007. FOSE'07, IEEE, 2007.
[21] M. Whalen, G. Gay, D. You, M. P.E. Heimdahl, M. Staats "Observable Modified Condition/Decision Coverage", In Proceedings In Proceedings of International Conference in Software Engineering (ICSE), 2013.
[22] I. Papadhopulli, N. Frasheri, "Today's Challenges of Symbolic Execution and Search-Based for Automated Structural Testing", In Proceedings of ICTIC, 2015.
[23] F. Gross, G. Fraser, A. Zeller, "Search-based system testing: high coverage, no false alarms". In Proceedings of International Symposium on Software Testing and Analysis (ISSTA), 2012.
[24] J. Galeotti, G. Fraser, A. Arcuri, " Extending a Search-Based Test Generator with Adaptive Dynamic Symbolic Execution", In Proceedings of International Symposium on Software Testing and Analysis (ISSTA) 2014.