ISSN (Online) 2393-8021
ISSN (Print) 2394-1588

IARJSET

International Advanced Research Journal in Science, Engineering and Technology

ICRAESIT - 2015

B V Raju Institute of Technology, Narsapur

Vol. 2, Special Issue 2, December 2015

Mutation Analysis Algorithms for Testing Object Oriented Classes

T. Shyam Kumar[1], Dr. Amjan Shaik[2], Dr. Neelakantappa.M[3]

Senior Software Engineer, Century Software SDN BHD, Malaysia[1]

Professor of IT, BVRIT, Narsapur, Medak, TS, India[2, 3]

Abstract: In this paper, we present TEST, an approach that automatically generates unit tests for object-oriented classes based on mutation analysis. By using mutations rather than structural properties as coverage criterion, we not only get guidance in where to test, but also what to test for. This allows us to generate effective test oracles, a feature raising automation to a level not offered by traditional tools.To assess the quality of test suites, mutation analysis seeds artificial defects (mutations) into programs; a non-detected mutation indicates a weakness in the test suite. We present an automated approach to generate unit tests that detect these mutations for object-oriented classes. This has two advantages: First, the resulting test suite is optimized toward finding defects modelled by mutation operators rather than covering code. Second, the state change caused by mutations induces oracles that precisely detect the mutants. Evaluated on 10 open source libraries, our _TEST prototype generates test suites that find significantly more seeded defects than the original manually written test suites. Improving test cases after mutation analysis usually means that the tester has to go back to the drawing-board and design new test cases, taking the feedback gained from the mutation analysis into account. This process requires a deep understanding of the source code and is a nontrivial task. Automated test generation can help in covering code (and thus hopefully detecting mutants), but even then, the tester still needs to assess the results of the generated executions—and has to write hundreds or even thousands of oracles. The test cases of an existing test suite are executed on a program version (mutant) containing one such fault at a time in order to see if any of the test cases can detect that there is a fault. A mutant that is detected as such is considered dead and of no further use. A live mutant, however, shows a case where the test suite potentially fails to detect an error and therefore needs improvement. There are two main problems of mutation analysis: First, the sheer number of mutants and the effort of checking all tests against all mutants can cause significant computational costs. Second, equivalent mutants do not observably change the program behaviour or are even semantically identical, and so there is no way to possibly detect them by testing.

Keywords: Mutation analysis, Testing, Object-Oriented Classes, Test Generation.

I. INTRODUCTION

Since the introduction of mutation analysis, a number of optimizations have been proposed to overcome possible performance problems, and heuristics can identify a small fraction of equivalent mutants at the end of the day, however, detecting equivalent mutants is still a job to be done manually.
If the mutant method/constructor is executed but the mutated statement it is not, then the fitness specifies the distance of the test case to executing the mutation; again, we want to minimize this distance. This basically is the prevailing approach level and branch distance measurement applied in a search-based test data generation. The approach level describes how far a test case was from the target in the control flow graph when it deviated course.

This is usually measured as the number of unsatisfied control dependencies between the point of deviation and the target, and is 0 if all control dependent branches are reached. The branch distance estimates how far the branch at which execution diverged from reaching the mutation is from evaluating to the necessary outcome. In addition, one can use the necessity conditions definable for different mutation operators to estimate the distance to an execution of the mutation that infects the state (necessity distance). To determine these values, the test case has to be executed once on the unmodified software. Data theft attacks are increase the volume of the attacker is aintended to do harm insider. This is considered as one of the top effective threats to cloud computing by the Cloud privacy Alliance.
While most Cloud computing users are well-aware of this effective threat, they are left only with If the mutation is executed, then we want the test case to propagate any changes induced by the mutation such that they can be observed. Traditionally, this consists of two conditions: First, the mutation needs to infect the state (necessity condition). This condition can be formalized and integrated into the fitness function (see above). In addition, however, the mutation needs to propagate to an observable state (sufficiency condition)—it is difficult to formalize this condition. Therefore, we measure the impact of the mutation on the execution; we want this impact to be as large as possible. Quantification

IARJSET

ISSN (Online) 2393-8021
ISSN (Print) 2394-1588

International Advanced Research Journal in Science, Engineering and Technology

ICRAESIT - 2015

B V Raju Institute of Technology, Narsapur

Vol. 2, Special Issue 2, December 2015

of the impact of mutations was initially proposed using dynamic invariants. The more invariants of the original program a mutant program violates, the less likely it is to be equivalent. A variation of the impact measurement uses the number of methods with changed coverage or return values instead of invariants.

2. LITERATURE REVIEW

2.1Effectiveness of Debugging When Random Test Cases Are Used

Automatically generated test cases are usually evaluated in terms of their fault revealing or coverage capability. Beside these two aspects, test cases are also the major source of information for fault localization and fixing. The impact of automatically generated test cases on the debugging activity, compared to the use of manually written test cases, has never been studied before. In this paper we report the results obtained from two controlled experiments with human subjects performing debugging tasks using automatically generated or manually written test cases. We investigate whether the features of the former type of test cases, which make them less readable and understandable (e.g., unclear test scenarios, meaningless identifiers), have an impact on accuracy and efficiency of debugging. The empirical study is aimed at investigating whether, despite the lack of readability in automatically generated test cases, subjects can still take advantage of them during debugging.We focused on modeling user search behavior to reveal an attacker's malicious intent. Automatic test case generation is an important area of software testing. The scientific community has reserved a great attention to test generation techniques, which are now available for many popular frameworks and programming languages

2.2 Test Sequence Length in Software Testing for Structural Coverage

In the presence of an internal state, often a sequence of function calls is required to test software. In fact, to cover a particular branch of the code, a sequence of previous function calls might be required to put the internal state in the appropriate configuration. Internal states are not only present in object-oriented software, but also in procedural software (e.g., static variables in C programs). In the literature, there are many techniques to test this type of software. However, to the best of our knowledge, the properties related to the choice of the length of these sequences have received only little attention in the literature. In this paper, we analyses the role that the length plays in software testing, in particular branch coverage. We show that, on “difficult” software testing benchmarks, longer test sequences make their testing trivial. Hence, we argue that the choice of the length of the test sequences is very important in software testing. Theoretical analyses and empirical studies on widely used benchmarks and onan industrial software are carried out to support our claims.

Software testing for white box criteria (e.g., branch coverage) consists of finding a set of test cases (i.e., the test suite) that maximizes those criteria [30]. Often, a test case is a driver that calls the function under test with a particular set of input values. The driver then compares the obtained output against the expected one. Using all possible inputs is infeasible because their number is innate in general.
Hence, the automation of software testing in this context consists of automatically finding the smallest set of these inputs such that the testing criterion is maximized. More problems arise if the software has an internal state.

In this paper, we analyze the role that the length of test sequences plays in testing software with internal state. In particular, we concentrate on the branch coverage criterion, because it is one of the most common criteria in the literature. To obtain high coverage, a common practice is to apply a first phase of random testing, followed by more sophisticated techniques aimed to cover the remaining branches (control flow analysis can be used to reduce their number by considering their dependences, e.g. nested branches). In fact, many branches are easy, and using expensive techniques to cover them would not be cost-effective. Some examples are formally proven. At any rate, this approach has two main issues, because there can be branches that are infeasible. First, we need to decide how long to apply random testing. Second, when we target specific branches, we need to decide as well how much effort we want to spend in trying to cover them. In fact, some of them could be infeasible. Only if we obtain 100% coverage we can be sure that we can stop the search effort. At any rate, when we target one branch at the time, it is very important to keep track of the other branches that can be covered by chance.

This document play a vital role in the development of life cycle (SDLC) as it describes the complete requirement of the system. It means for use by developers and will be the basic during testing phase. Any changes made to the requirements in the future will have to go through formal change approval process.
SPIRAL MODEL was defined by Barry Boehm in his 1988 article, “A spiral Model of Software Development and Enhancement. This model was not the first model to discuss iterative development, but it was the first model to explain why the iteration models. As originally envisioned, the iterations were typically 6 months to 2 years long. Each phase starts with a design goal and ends with a client reviewing the progress thus far. Analysis and engineering efforts are applied at each phase of the project, with an eye toward the end goal of the project.

IARJSET

ISSN (Online) 2393-8021
ISSN (Print) 2394-1588

International Advanced Research Journal in Science, Engineering and Technology

ICRAESIT - 2015

B V Raju Institute of Technology, Narsapur

Vol. 2, Special Issue 2, December 2015

Since the introduction of mutation analysis, a number of optimizations have been proposed to overcome possible performance problems, and heuristics can identify a small fraction of equivalent mutants at the end of the day, however, detecting equivalent mutants is still a job to be done manually. A recent survey paper concisely summarizes all applications and optimizations that have been proposed over the years. Javalanche is a tool that incorporates many of the proposed optimizations and made it possible to apply mutation analysis to software of previously unthinkable size.

2.1 Architecture

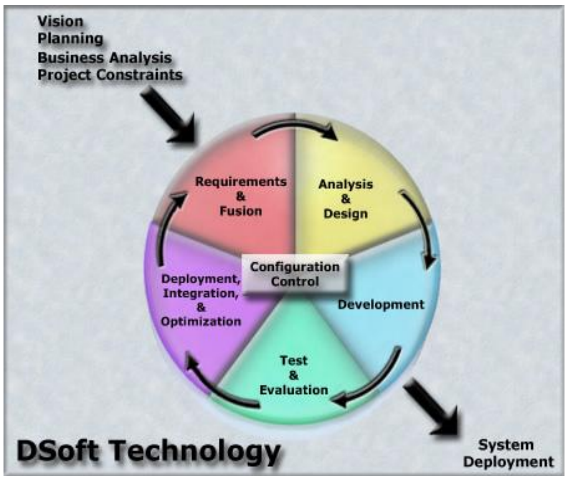

Figure1: DSoft Development Process Life Cycle

There are two main problems of mutation analysis: First, the sheer number of mutants and the effort of checking all tests against all mutants can cause significant computational costs. Second, equivalent mutants do not observably change the program behavior or are even semantically identical, and so there is no way to possibly detect them by testing. Since the introduction of mutation analysis, a number of optimizations have been proposed to overcome possible performance problems, and heuristics can identify a small fraction of equivalent mutants—at the end of the day, however, detecting equivalent mutants is still a job to be done manually.

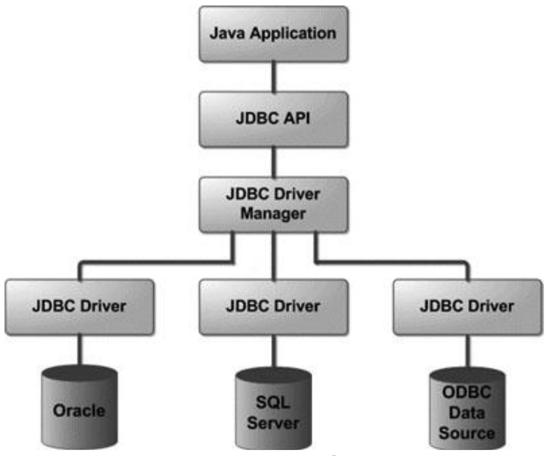

Figure2: JDBC Architecture

Software testing for white box criteria (e.g., branch coverage) consists of finding a set of test cases (i.e., the test suite) that maximizes those criteria [30]. Often, a test case is a driver that calls the function under test with a particular set of input values. The driver then compares the obtained output against the expected one.
Using all possible inputs is infeasible because their number is innate in general. Hence, the automation of software testing in this context consists of automatically finding the smallest set of these inputs such that the testing criterion is maximized. More problems arise if the software has an internal state. An internal state can be for example static variables inside functions in C programs. In object-oriented (OO) software, most programs have internal states. Internal states are problematic because the coverage of some code structures can depend on the current status of the internal state. To put the internal state in the right configuration, a sequence of function calls is often required.

In this paper, we analyse the role that the length of test sequences plays in testing software with internal state. In particular, we concentrate on the branch coverage criterion, because it is one of the most common criteria in the literature. To obtain high coverage, a common practice is to apply a first phase of random testing, followed by more sophisticated techniques aimed to cover the remaining branches (control flow analysis can be used to reduce their number by considering their dependences, e.g. nested branches).

In fact, many branches are easy, and using expensive techniques to cover them would not be cost-effective. Some examples are formally proven. At any rate, this approach has two main issues, because there can be branches that are infeasible. First, we need to decide how long to apply random testing. Second, when we target specific branches, we need to decide as well how much effort we want to spend in trying to cover them. In fact, some of them could be infeasible. Only if we obtain 100% coverage we can be sure that we can stop the search effort. At any rate, when we target one branch at the time, it is very important to keep track of the other branches that can be covered by chance.
For each user, which checks the previous history of incidental accesses of the user to the lure file documents on their file system. Based on this previous historical behavior pattern, we select a threshold limit, beyond which access to lure file documents is considered excessive or showing a cautious distrust of someone or something, in other words indicative of a false show activity. These models are also developed for each individual user by using 80% of the lure access data. We used the rest of the user data, as well as the showing a careful to avoid potential problems distrust of someone or something a false data for testing the user models.

Test cases can be generated effectively applying mutation analysis: Artificial defects (mutants) are injected into software and test cases are executed on these fault-injected versions.
A mutant that is not detected shows a deficiency in the test suite and indicates in most cases that either a new test case should be added, or that an existing test case needs a better test oracle. Improving test cases after mutation analysis usually means that the tester has to go back to the drawing-board and design new test cases, taking the feedback gained from the mutation analysis into account.

3. EXISTING SYSTEM

To assess the quality of test suites, mutation analysis seeds artificial defects (mutations) into programs; a non-detected mutation indicates a weakness in the test suite. We present an automated approach to generate unit tests that detect these mutations for object-oriented classes.

This has two advantages: First, the resulting test suite is optimized towards finding defects rather than covering code. Second, the state change caused by mutations induces oracles that precisely detect the mutants. Evaluated on two open source libraries, our µtest prototype generates test suites that find significantly more seeded defects than the original manually written test suites.

4. PROPOSED SYSTEM

Since the introduction of mutation analysis, a number of optimizations have been proposed to overcome possible performance problems, and heuristics can identify a small fraction of equivalent mutants at the end of the day, however, detecting equivalent mutants is still a job to be done manually. A recent survey paper concisely summarizes all applications and optimizations that have been proposed over the years. Javalanche is a tool that incorporates many of the proposed optimizations and made it possible to apply mutation analysis to software of previously unthinkable size. In addition, Javalanche alleviates the equivalent mutant problem by ranking mutants by their impact: A mutant with high impact is less likely to be equivalent, which allows the tester to focus on those mutants that can really help to improve a test suite. The impact can be measured, for example, in terms of violated invariants or effects on code coverage.

5. SCOPE OF DEVELOPMENT

Mutation analysis was introduced in the 1970s as a method to evaluate a test suite in how good it is at detecting faults, and to give insight into how and where a test suite needs improvement. The idea of mutation analysis is to seed artificial faults based on what is thought to be real errors commonly made by programmers. The test cases of an existing test suite are executed on a program version (mutant) containing one such fault at a time in order to see if any of the test cases can detect that there is a fault. There are two main problems of mutation analysis: First, the sheer number of mutants and the effort of checking all tests against all mutants can cause significant computational costs. Second, equivalent mutants do not observably change the program behaviour or are even semantically identical, and so there is no way to possibly detect them by testing.

6. METHODOLOGY USED FOR THE PROPOSED SYSTEM

6.1Genetic algorithm:-

To demonstrate the absence of the fault in the mutant (to kill the mutant) it is necessary that the test case cause execution to reach the mutated statement (reachability condition), the value of the mutated expression must be such that it is possible to have a different data state in the mutant compared to the subject (necessity condition) and this state difference must be propagated to the output (sufficiency condition). By incorporating these three conditions into the fitness function of a genetic algorithm a search may be made for test data to kill a given mutant. This paper describes such a fitness function. The function described has been implemented and preliminary results should be available in time for the workshop.

6. 2Genetic algorithm fitness function:-

The genetic algorithm searches the input domain of the subject program for suitable test cases to kill a given mutant. Guidance is provided by the fitness function which assigns a non-negative cost to each candidate input value. An input with a zero cost is a test case that kills the mutant.

Reachability condition:-

Consider the control flow graph of a subject program. The nodes are the basic blocks of the subject and the edges are the possible transitions between basic blocks. The conditional transitions are associated with a branch predicate. There is a distinguished start node and a distinguished exit node.To kill a mutant, the execution path of a test must reach the mutated statement, the particular path is unimportant

6.3 Algorithms used for solving the problem

Algorithm 1 Random generation used for the initial population.

```
Require: C: Set of all methods and constructors directly/
indirectly calling mutation
t←  s ← random generator for unit under test
forcallee and all parameters of s do
GenObject(class of parameter, {}, t)
end for
t← t.s
while|t| < desired length do
s← random method using an object from t as either
callee or parameter
forcallee and all parameters of s not existing in t do
GenObject(class of parameter, {}, t)
end for
t← t.s
end while
return t
```

Algorithm 2 GenObject: Recursive generation of objects.

```
Require: c: Class of desired object
Require: G: Set of classes already attempting to generate
Require: t: Test case
if t has object of class c then
return existing object with certain probability
end if
s← random element from all generators
if s = non-static method or field then
G ← G∪{c} Set callee of s to GenObject(Class of callee, G, t)
end if
for all parameters of s do
Set parameter to GenObject(Class of parameter, G,t)
end for
t← t.s
return t
```

Mutation analysis was introduced in the 1970s as a method to evaluate a test suite in how good it is at detecting faults, and to give insight into how and where a test suite needs improvement. The idea of mutation analysis is to seed artificial faults based on what is thought to be real errors commonly made by programmers.
The test cases of an existing test suite are executed on a program version (mutant) containing one such fault at a time in order to see if any of the test cases can detect that there is a fault. There are two main problems of mutation analysis: First, the sheer number of mutants and the effort of checking all tests against all mutants can cause significant computational costs. Second, equivalent mutants do not observably change the program behaviour or are even semantically identical, and so there is no way to possibly detect them by testing.
Javalanche is a tool that incorporates many of the proposed optimizations and made it possible to apply mutation analysis to software of previously unthinkable size. In addition, Javalanche alleviates the equivalent mutant problem by ranking mutants by their impact: A mutant with high impact is less likely to be equivalent, which allows the tester to focus on those mutants that can really help to improve a test suite.
At the customer option, the entire project can be aborted if the risk is deemed too great. Risk factors might involve development cost overruns, operating-cost miscalculation, or any other factor that could, in the customer's judgment, result in a less-than-satisfactory final product.
The existing prototype is evaluated in the same manner as was the previous prototype, and if necessary, another prototype is developed from it according to the fourfold procedure outlined above.
The preceding steps are iterated until the customer is satisfied that the refined prototype represents the final product desired. The final system is constructed, based on the refined prototype. The final system is thoroughly evaluated and tested. Routine maintenance is carried on a continuing basis to prevent large scale failures and to minimize down time.
At the customer option, the entire project can be aborted if the risk is deemed too great. Risk factors might involve development cost overruns, operating-cost miscalculation, or any other factor that could, in the customer's judgment, result in a less-than-satisfactory final product.
The existing prototype is evaluated in the same manner as was the previous prototype, and if necessary, another prototype is developed from it according to the fourfold procedure outlined above. The preceding steps are iterated until the customer is satisfied that the refined prototype represents the final product desired. The final system is constructed, based on the refined prototype. The final system is thoroughly evaluated and tested. Routine maintenance is carried on a continuing basis to prevent large scale failures and to minimize down time.

7. IMPLEMENTATION

7.1System architecture

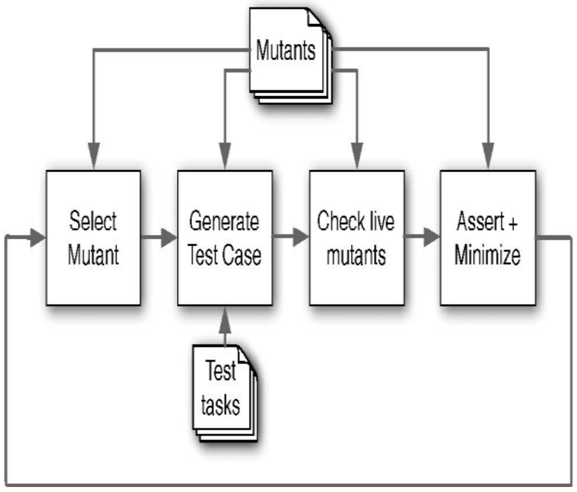

Figure3: Procedural Model for Implementation of Mutation Analysis

7.2 System Modules

Distance to Mutation

If the mutant method/constructor is executed but the mutated statement itself is not, then the fitness specifies the distance of the test case to executing the mutation; again, we want to minimize this distance. This basically is the prevailing approach level and branch distance measurement applied in a search-based test data generation. The approach level describes how far a test case was from the target in the control flow graph when it deviated course. This is usually measured as the number of unsatisfied control dependencies between the point of deviation and the target, and is 0 if all control dependent branches are reached. The branch distance estimates how far the branch at which execution diverged from reaching the mutation is from evaluating to the necessary outcome. In addition, one can use the necessity conditions definable for different mutation operators to estimate the distance to an execution of the mutation that infects the state (necessity distance). To determine these values, the test case has to be executed once on the unmodified software.

Mutation Impact

If the mutation is executed, then we want the test case to propagate any changes induced by the mutation such that they can be observed. Traditionally, this consists of two conditions: First, the mutation needs to infect the state (necessity condition). This condition can be formalized and integrated into the fitness function (see above). In addition, however, the mutation needs to propagate to an observable state (sufficiency condition)—it is difficult to formalize this condition. Therefore, we measure the impact of the mutation on the execution; we want this impact to be as large as possible. Quantification of the impact of mutations was initially proposed using dynamic invariants. The more invariants of the original program a mutant program violates, the less likely it is to be equivalent. A variation of the impact measurement uses the number of methods with changed coverage or return values instead of invariants.

Mutation Analysis

The first step of a mutation-based approach is to perform classic mutation analysis, which Javalanche does efficiently. Javalanche instruments Java bytecode with mutation code and runs JUnit test cases against the instrumented (mutated) version. The result of this process is 1) a classification of mutants into dead or live with respect to the JUnit test suite, as well as 2) an impact analysis on live mutants that have been executed but not detected.

8. CONCLUSION AND FUTURE WORK

Mutation analysis is known to be effective in evaluating existing test suites. In this paper, we have shown that mutation analysis can also drive automated test generation.
The main difference between using structural coverage and mutation analysis to guide test generation is that a mutation does not only show where to test, but also helps in identifying what should be checked for.
In our experiments, this results in test suites that are significantly better in finding defects than the (already high quality) manually written ones.
The advent of automatic generation of effective test suites has an impact on the entire unit testing process: Instead of laboriously thinking of sequences that lead to observable features and creating oracles to check these observations, the tester lets a tool create unit tests automatically and receives two test sets: one revealing general faults detectable by random testing, the other one consisting of regular unit tests. In the long run, finding bugs could thus be reduced to the task of checking whether the generated assertions match the intended behavior.

REFFERENCES

1. S. Ali, L.C. Briand, H. Hemmati, and R.K. Panesar-Walawege, “A Systematic Review of the Application and Empirical Investigation of Search-Based Test-Case Generation,” IEEE Trans. Software Eng., vol. 36, no. 6, pp. 742-762, Nov./Dec. 2010.
2. J.H. Andrews, L.C. Briand, and Y. Labiche, “Is Mutation an Appropriate Tool for Testing Experiments?” Proc. 27th Int’l Conf. Software Eng., pp. 402-411, 2005.
3. J.H. Andrews, A. Groce, M. Weston, and R.G. Xu, “Random Test Run Length and Effectiveness,” Proc. IEEE/ACM 23rd Int’l Conf. Automated Software Eng., pp. 19-28, 2008.
4. J.H. Andrews, S. Haldar, Y. Lei, and F.C.H. Li, “Tool Support for Randomized Unit Testing,” Proc. First Int’l Workshop Random Testing, pp. 36-45, 2006.
5. A. Arcuri, “It Does Matter How You Normalise the Branch Distance in Search Based Software Testing,” Proc. Third Int’l Conf. Software Testing, Verification and Validation, pp. 205-214, 2010.
6. A. Arcuri, “Longer Is Better: On the Role of Test Sequence Length in Software Testing,” Proc. Third Int’l Conf. Software Testing, Verification and Validation, pp. 469-478, 2010.
7. A. Arcuri and L. Briand, “A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering,” Proc. IEEE Int’l Conf. Software Eng., pp. 1-10, 2011.
8. A. Arcuri and X. Yao, “Search Based Software Testing of Object- Oriented Containers,” Information Sciences, vol. 178, no. 15, pp. 3075-3095, 2008.
9. K. Ayari, S. Bouktif, and G. Antoniol, “Automatic Mutation Test Input Data Generation via Ant Colony,” Proc. Ninth Ann. Conf. Genetic and Evolutionary Computation, pp. 1074-1081, 2007.
10. M. Boshernitsan, R. Doong, and A. Savoia, “From Daikon to Agitator: Lessons and Challenges in Building a Commercial Tool for Developer Testing,” Proc. Int’l Symp. Software Testing and Analysis, pp. 169-180, 2006.
11. L. Bottaci, “A Genetic Algorithm Fitness Function for Mutation Testing,” Proc. Int’l Workshop Software Eng. Using MetaheuristicInovative Algorithms, a Workshop at 23rd Int’l Conf. Software Eng., pp. 3-7, 2001.
12. V. Chvatal, “A Greedy Heuristic for the Set-Covering Problem,” Math.Operations Research, vol. 4, no. 3, pp. 233-235, 1979.