Scientific Research and Essays Vol. 7(45), pp. 3848 -3859, 19 November, 2012
Available online at http://www.academicjournals.org/SRE
DOI: 10.5897/SRE11.1487
ISSN 1992-2248 ©2012 Academic Journals

*Full Length Research Paper*

# Effectiveness of control flow test coverage criteria using mutation analysis: An experimental study

**Hossein Keramati* and Seyed-Hassan Mirian-Hosseinabadi**

Computer Engineering Department, Sharif University of Technology, Tehran, Iran.

Accepted 12 November, 2012

**Empirical studies play an important role in measuring the effectiveness of software testing methods and coverage criteria. This has led us to develop an experimental research to study four major test coverage criteria based on the Control Flow Graphs extracted from the source code of programs. In this study, different implementations of an industrial problem are selected as subject programs and the effectiveness of Edge, Edge-Pair, Prime-Path and All-Path coverage criteria are measured for them by means of mutation analysis. Generating and evaluating large number of mutants without random selection in one hand, and running the experiment against entire input domain on the other hand, increased accuracy of the results and removed effect of using random mutants and test case pools in similar experimental studies. Analyzing the results, we discuss the effectiveness of these four coverage criteria, the effect of employing Sidetrip and Detour touring, and the reliability and maximum power of graph-based coverage criteria.**

**Key words:** Software testing, control flow coverage criteria, mutation analysis, empirical study.

## INTRODUCTION

In unit testing, which is the primary method for testing software elements and components, test cases are designed to be executed against the system under test and to evaluate whether the software is working as it is expected or not. In this respect, test engineers need methods and techniques to design test suites as effective as possible. Trying to fulfill this requirement, graph coverage is a category of model-based software testing criteria that is based on Control Flow Graphs (CFG) extracted from the software artifacts (for example, source code). Based on source code graph coverage criteria, including control flow and data flow criteria, the source code of a program should be first modeled as a graph, and lots of paths can then be derived from that graph as test requirements. These test requirements could be called path-based requirements. Each test requirement should be satisfied by at least one test case. Coverage level of test suites indicates the number of satisfied test requirements via the test suite against the whole number of extracted requirements for a specific criterion. In other words, these criteria are to guide test engineers seeking for effective test cases and to help them when to stop augmenting test suites. Edge, Edge-Pair, Prime-Path and All-Path coverage criteria are four control flow graph coverage criteria (Ammann and Offutt, 2008) which are studied in this experimental research. Due to diversity in effectiveness, complexity and cost of employing these criteria, empirical studies are being employed to evaluate the effectiveness of coverage criteria. In this approach, some programs are selected as subject programs and are designed to measure the effectiveness of criteria-covering test suites on detecting faults in the programs. Faults could be real historical bugs found in subject programs or seeded faults.

Mutation analysis is an automatic fault seeding technique which uses a variety of mutation operators to change original source code of programs and evaluates the power of test suites on revealing the seeded faults. Mutation analysis assumes that the seeded faults, called mutants are representatives of real faults in subject programs; therefore, mutation score is taken as fault

*Corresponding author. E-mail: keramati@ce.sharif.edu. Tel: +98 21 66166679. Fax: +98 21 66019246.

detection rate in studies based on this evaluation technique. This assumption has been validated and confirmed empirically in some studies (Andrews et al., 2005, 2006) on programs which have history of real faults. They “suggest that mutants, when using carefully selected mutation operators and after removing equivalent mutants can provide a good indication of the fault detection ability of test suites” (Andrews et al., 2005).

## Motivation

Numerous studies have been done aiming at evaluating the effectiveness of control flow graph coverage criteria. Major works are reviewed in the “related works” sub-section. Most of these works use mutation analysis technique for evaluation. But, because of the wide range of the input domain of their subject programs, we cannot see any result discussing the maximum power of path-based coverage criteria which can be achieved through applying All-Path as selected coverage criterion. This evaluation requires detecting all infeasible paths in the extracted control flow graph model and generating test suites that cover all feasible test requirements. Another inaccuracy in such studies is using approximation in detecting equivalent mutants, where mutants have the same behavior as the original program and cannot be killed through any test case. Even though equivalent mutants should be detected and removed from the experiments, most studies use approximation or random techniques for detection. It is because detecting all equivalent mutants needs to run the program against the entire input domain which is not possible for their subject programs. Remaining equivalent mutants undetected injects a bias into the results of fault detection rate of test suites and decreases the accuracy of these experiments. Selecting subject programs with finite input domain lets us to perform such experiments more accurately.

In addition to measuring maximum effectiveness of path-based coverage criteria, which is achieved through evaluating mutant detection rate of All-Path covering test suites, it also lets us to compare the effectiveness of weaker criteria with respect to All-Path criterion empirically. In this study, 10 programs from the same problem domain are selected as subject programs. These are used in evaluating the effectiveness of the four graph coverage criteria: Edge, Edge-Pair, Prima Path and All-Path on revealing seeded faults in the programs. The experimental evaluation is based on mutation analysis. Selecting the entire input domain of the programs and detecting equivalent mutants completely and all infeasible test requirements as well, lets us to measure the effectiveness of these four criteria precisely in addition to comparative evaluation of their effectiveness. In addition, maximum power of path-based coverage criteria is reported for our subject programs and discussed. It is due to building test suites that traverse all feasible paths within control flow graph of the programs. Applying Prime-Path as a control flow graph coverage criterion in the experiment, which has been studied in a few number of previous works (Li et al., 2009), is another motivation for this empirical study. In addition, removing the negative effect of easy to find faults through detecting easy-to-kill mutants makes the comparative results more expressive and meaningful. According to regression testing challenges, effectiveness evaluation of cross test suites, which is studying the effectiveness of test suites generated for a program under test on detecting faults in another implementation of that program, makes the experiment and its results more motivated. Another contribution of this study is presenting results of applying indirect touring methods such as Sidetrip and Detour (Ammann and Offutt, 2008), on satisfying infeasible test requirements of control flow graph based coverage criteria.

This paper is constructed as follows: A brief review of works related to empirical analysis of software testing methods is first presented. Thereafter the design of the experiment is described. Results of the experiment and their analysis are discussed and lastly, the study is concluded and future works recommended.

## Related works

Using mutant analysis in software testing was initially introduced by Budd and Sayward (1977), Hamlet (1977), DeMillo et al. (1978) and continued by other researchers subsequently. Offutt et al. (1996) proposed a subset of mutant operators that are sufficient for mutation testing and made this process less expensive than before. Jia and Harman (2011) provide a recent comprehensive analysis and survey on mutation testing. In addition to using mutation analysis as a test adequacy criterion, some empirical studies employ mutation analysis as a tool for evaluating the effectiveness of test suites which are generated to satisfy other coverage criteria. Many researches such as works that have been done by Thévenod-Fosse et al. (2002), Kim et al. (2001), Andrews and Zhang (2003) and Andrews et al. (2006) use mutation analysis for this purpose. All of these empirical studies are based on the assumption that automatically seeded faults are representatives of real faults in programs. Andrews et al. (2005) validated this assumption and suggested to use mutation analysis in empirical studies on evaluating the effectiveness of test suites generated with other software testing methods.

Li et al. (2009) reported the results of their experimental study on comparison of Edge-Pair, All-Uses and Prime-Path coverage criteria in addition to mutation analysis over a set of small programs. To our knowledge, this is the only study that has directly examined the effectiveness of Prime-Path coverage criterion.

**Table 1.** Summary information on the subject programs.

| Program | LoC | # of methods | ∑ CCN | CFG nodes | CFG edges |
|---|---|---|---|---|---|
| P1 | 86 | 1 | 16 | 17 | 22 |
| P2 | 75 | 3 | 25 | 30 | 37 |
| P3 | 56 | 2 | 16 | 13 | 17 |
| P4 | 125 | 5 | 38 | 38 | 45 |
| P5j | 162 | 5 | 42 | 69 | 83 |
| P5p | 139 | 5 | 36 | 56 | 67 |
| P6 | 50 | 2 | 13 | 10 | 10 |
| P7 | 141 | 6 | 402 | 40 | 42 |
| P7a | 144 | 7 | 35 | 42 | 45 |
| P8 | 670 | 7 | 297 | 157 | 267 |
| Total | 1648 | 43 | 920 | 472 | 635 |

## EXPERIMENTAL DESIGN

### Subject programs

Ten programs are selected as subject programs in this study. They are different implementations of the same problem, converting Jalali calendar dates to Gregorian calendar dates. Calendar converter is an essential component in applications implemented for countries that use calendars other than built-in Gregorian calendar in computing machines. One of them is Jalali calendar. Since all transactions, business logics, and user interfaces of these applications, especially information systems are based on dates and times, reliability and accuracy of the date converter component is essential. Because of the complexity of date conversion for Jalali calendar (Borkowski, 1996), there are several implementations of this problem. Some use simple algorithms with a little code and some others implement more accurate and complex date-conversion algorithms. In this study, we have selected 8 different industrial implementations of this problem from the Internet and all are in Java. One of them (P5) has two versions that are different in only one method. We also duplicated one of these programs (P7) and changed a small part of it to see its consequence on the effectiveness of testing methods. Finally, we converted programming interface of all programs into a common interface, a class with a public method with three integer values as its input parameters and a return value indicating the converted Jalali date.

Table 1 shows the list of the programs, their lines of code measured by the CLOC (Northrop Grumman Corporation, 2010) tool, and cyclomatic complexity number (CCN) (McCabe, 1976) of them which are measured by both CyVis (Selvaraj and Iyer, 2006) and JavaNCSS (Lee, 2009) tools. In addition, the number of edges and nodes in extracted control flow graphs of the programs are listed in this table.

### Input domain

In general, date converter programs should convert all dates from the source calendar system to appropriate dates in the target calendar. But, this assumption is not true in practice. It is due to the complexity of conversion algorithms, lack of precise and reliable knowledge about entire history of the calendars and need for making decision on some future years according to whether or not it is a leap year. As a result, there are various implementations of the date conversion problem with different properties. Most Jalali to Gregorian date converters are implemented for years between 475 Jalali and 1468. The upper range is limited, because year 1469 has to be decided by governments to be a leap year or not. Accordingly, we restrict input domain to years in this range. The exception is program P8, which is implemented for years between 1 and 474 and due to this difference, we cannot use it in cross test suites effectiveness evaluation. All programs rely on a defined precondition which limits days between 1 and 31, months between 1 to 12, and years in the specified range. In addition, we added a few code to all programs to control this precondition.

According to this defined input domain, first nine programs have 369,768 and P8 has 176,328 different input values which make test inputs pool. We also added a lot of test inputs to have test cases for the injected precondition checking codes. This defined pool of inputs is large in size but it is finite and reasonable and lets us to employ the entire input domain in the experimental process with a plausible cost.

### Mutants

We used MuJava (Ma et al., 2005) to generate mutants for the subject programs. All 15 mutation operators supported by MuJava were applied on all subject programs. It generated 16,056 mutant programs totally. Table 2 shows detailed information on the number of generated mutants for each program. In mutation analysis process, a lot of generated mutant programs are equivalent to the original program and cannot be killed by any test case in problem domain. Considering equivalent mutants in evaluating the fault detection rate of test suites underestimates effectiveness of these suites and makes the results unreliable. Using random subsets from input domain on detecting equivalent mutants makes mutation analysis inaccurate and may incorrectly mark some mutants as equivalent. Some studies such as Andrews et al. (2006) use historical test inputs plus test inputs generated to satisfy some coverage criteria in addition to random-generated suites to detect equivalent mutants. It makes the experiment more accurate but does not completely solve the problem. Results of our study, as discussed in ‘reliability of path-based testing methods’, show that even if we select enough test inputs to traverse all feasible paths in the subject program, there are a lot of mutants that will remain alive. They will be considered as equivalent to the original program but are not actually equal; there are some test inputs that can kill them and these mutants are expected to be killed by effective test suites.

In this study, to detect and remove equivalent mutants, all test

**Table 2.** Summary of mutants in mutation analysis.

| Program | All mutants | Equivalent | Easy-to-kill | Final set |
|---|---|---|---|---|
| P1 | 456 | 53 | 261 | 142 |
| P2 | 442 | 65 | 157 | 220 |
| P3 | 770 | 168 | 449 | 153 |
| P4 | 756 | 211 | 366 | 179 |
| P5j | 666 | 187 | 374 | 105 |
| P5p | 659 | 185 | 379 | 95 |
| P6 | 323 | 79 | 198 | 46 |
| P7 | 5,890 | 1,386 | 419 | 4,085 |
| P7a | 751 | 113 | 398 | 240 |
| P8 | 5,343 | 707 | 171 | 4,465 |
| Total | 16,056 | 3,154 | 3,172 | 9,730 |

inputs from the pool have been executed against all generated mutants. In this process, after about 5 billion test case executions, 3154 out of 16056 mutants were marked as equivalent mutants. Since the whole input domain is applied, all remained mutants are expected to be killed by an effective test suite and makes results of the experiment more precise. Another issue is mutants which are being killed by nearly all of test inputs from the input domain. They increase mutation score of test suites but killing them should not be considered as a success for test suites generated for a coverage criterion because all other small test suites, even random suites, will kill them. In this experiment, easy-to-kill mutants have been removed from the mutation analysis by detecting mutants which are being killed by 80% of randomly selected test cases. Number of detected equivalent and easy-to-kill mutants of subject programs is listed in Table 2. All non-equivalent and non-easy-to-kill mutants make the final set of mutant programs to be used in the mutation analysis process.

## Model extraction and test requirements

Control flow graphs of subject programs have been extracted by hand from their source code. Each node represents a basic block and we have considered an edge for each decision. Also, to track the execution path of programs against input test cases, instrumented version of each program also has been created according to its extracted control flow graph. Considering summary information on programs in Table 1, one may point out the inconsistency between the numbers given and the definition of Cyclomatic Complexity Number:

$$CCN = E - N + 2P$$

where E is the number of edges of the graph, N is the number of nodes of the graph and P is the number of connected components (P is considered as zero here).

When we have multiple predicate decision statements, there are two approaches on extracting CFGs from the source codes of such programs. McCabe's Cyclomatic Complexity assumes that in each decision statements, each clause of the predicate should be considered as a separate decision and an edge is required. Because this approach is not feasible in practice and dramatically decreases the maintainability of the source codes (Ammann and Offutt, 2008) and when we have combined non-decisional logical statements, it does not solve the problem totally; this problem is classified in logic coverage class of software testing criteria. Multiple-condition coverage criterion (Myers et al., 2004) and all other logic-based coverage criteria (Ammann and Offutt, 2008) are trying to cover these situations effectively. We have chosen the practical and more recommended approach to leave multiple-condition decision statements unchanged in the source codes and draw a branch edge for each decision. This is the reason behind the aforementioned inconsistent numbers.

To prepare test requirements, we took advantage of Control Flow Graph Web Application (Ammann et al., 2010), supporting tool of Ammann and Offutt's textbook (Ammann and Offutt, 2008), in extracting requirements of Edge, Edge-Pair and Prime-Path coverage criteria. Totally, 635, 814 and 1340 test requirements were extracted for Edge, Edge-Pair and Prime-Path criteria.

## Execution paths and test requirements feasibility problem

In generating test suites to satisfy coverage criteria, a major problem is test requirements that cannot be satisfied by any test case. These are infeasible test requirements. "The detection of infeasible test requirements is formally undecidable for most coverage criteria, and even though some researchers have tried to find partial solutions, they have had only limited success" (Ammann and Offutt, 2008). Dealing with the problem of detecting infeasible test requirements, we have generated all feasible execution paths of the subject programs. Each program in the study generates a relatively small number of unique execution paths against the entire input domain. Each of the first nine programs does not generate more than 1,000 unique paths out of 369,773 execution paths. P8 generates 12,442 different execution paths.

## Test suites

Test suite is a set of test cases which are generated according to a test selection strategy. In this study, we have three types of test suites: coverage criteria suites, all paths covering suites and random suites. Coverage criteria suites are generated to cover test requirements of Edge, Edge-Pair and Prime-Path coverage criteria with a specified percentage as their goal. For each criterion, a lot of test suites are generated to cover the requirements of that criterion from 5% up to maximum feasible coverage with the step of five. In addition, five redundant test suites are generated for each coverage level of each criterion. For example, for program P8, 90 test suites are generated to cover 18 different coverage levels (5 to 90%) of Edge-Pair coverage criterion. Test cases are selected randomly from the inputs pool to incrementally augment a test suite up to its specified coverage goal. When all feasible test requirements are

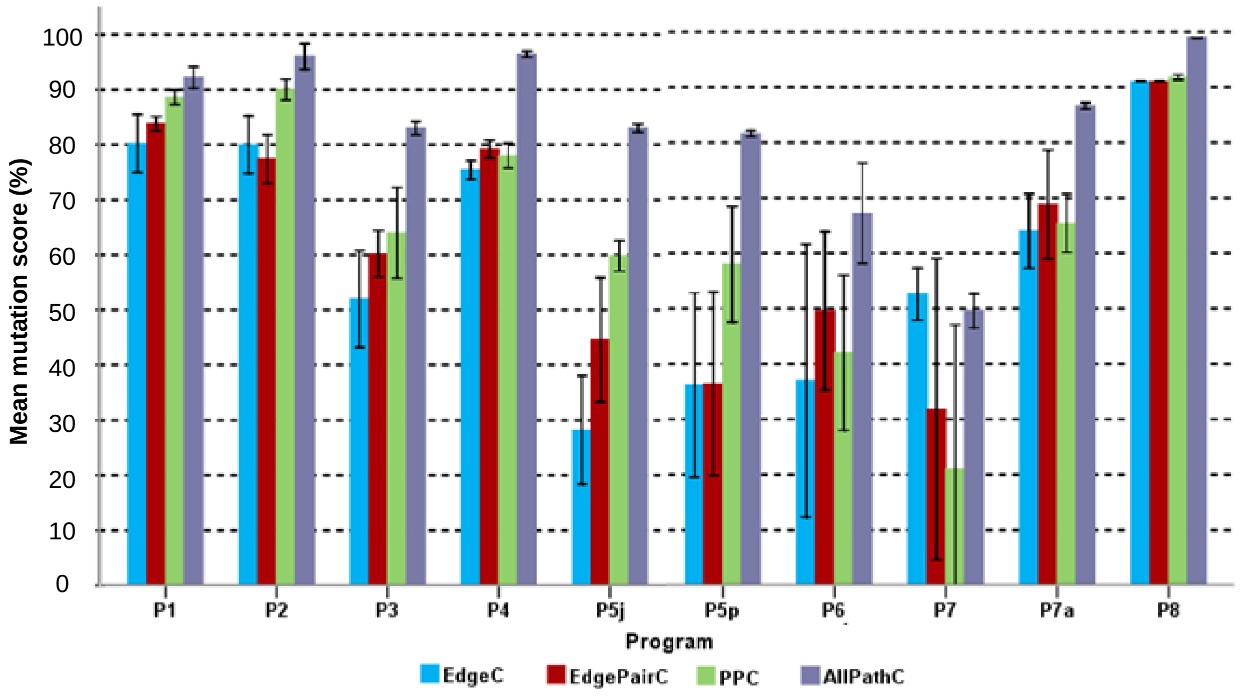


**Figure 1.** Mean of the mutation scores of high coverage test suites generated for each criterion.

covered but test suites do not reach their goal, Sidetrip and then Detour touring (Ammann and Offutt, 2008) are used to add more test cases to those suites to cover remained test requirement indirectly as much as possible. In addition to coverage criteria test suites, random suites are generated with suite size 1 up to 31 (maximum size of criteria suites) for programs P1 to P7a and up to 124 test cases for P8.

All-Path covering suites are generated to cover all feasible paths in the programs. Since each execution path may be covered by more than one test input, 5 redundant All-Path covering suites are generated for each program. Test cases to cover each feasible path are selected randomly for these suites too.

## RESULTS AND ANALYSIS

### Coverage criteria effectiveness

Here, we analyze the effectiveness of the three test coverage criteria on detecting seeded faults in the programs. To see the maximum power of each coverage criterion on detecting faults, we study results of mutation analysis on test suites which are generated to satisfy all feasible test requirements of the criterion under study. As mentioned in Test Suites, five test suites are generated for each specific coverage level of each criterion per subject program. Figure 1 shows the results as a bar chart indicating average mutation scores of five test suites with highest coverage level. Results for Edge, Edge-Pair and Prime- Path criteria are shown in order from left to right for each program. Also, the mean of mutation scores of test suites that are covering all feasible paths is presented as the most right bar of each column. Since all equivalent mutants have been removed from the mutation analysis process, all mutants are expected to be killed and effective test suites are expected to detect all faults and reach 100% mutation score. The first question is how many faults are detected by each coverage criteria? Results show that the effectiveness of coverage criteria is different in each program. For some programs (for example P1, P2 and P8), Prime-Path is relatively successful and detects almost 90% of the faults. But for other programs, all three coverage criteria leave out 20 to 80% of seeded faults. We will discuss the reasons of these low fault detection rates in ‘results and analysis’. Shortly, it highly depends on the programs under test.

On the other hand, in comparison between the effectiveness of All-Path and other three criteria, we see that the average fault detection rates of all of these three criteria are significantly below the detection rate of All-Path criterion covering test suites. In most of the programs, except P1, P2, P8, there is a large difference between mutation scores of All-Path coverage suites and the scores of Edge, Edge-Pair and Prime-Path criteria. Although, this is an expected result, it means that even a powerful criterion like Prime-Path cannot reach the fault detection rate that is achievable by path-based criteria. Even in P1, P2 and P8 that these three criteria are successful compared to the All-Path, they find that about 90% of seeded faults and 10% of detectable faults are remained undetected in the programs. This leads us to

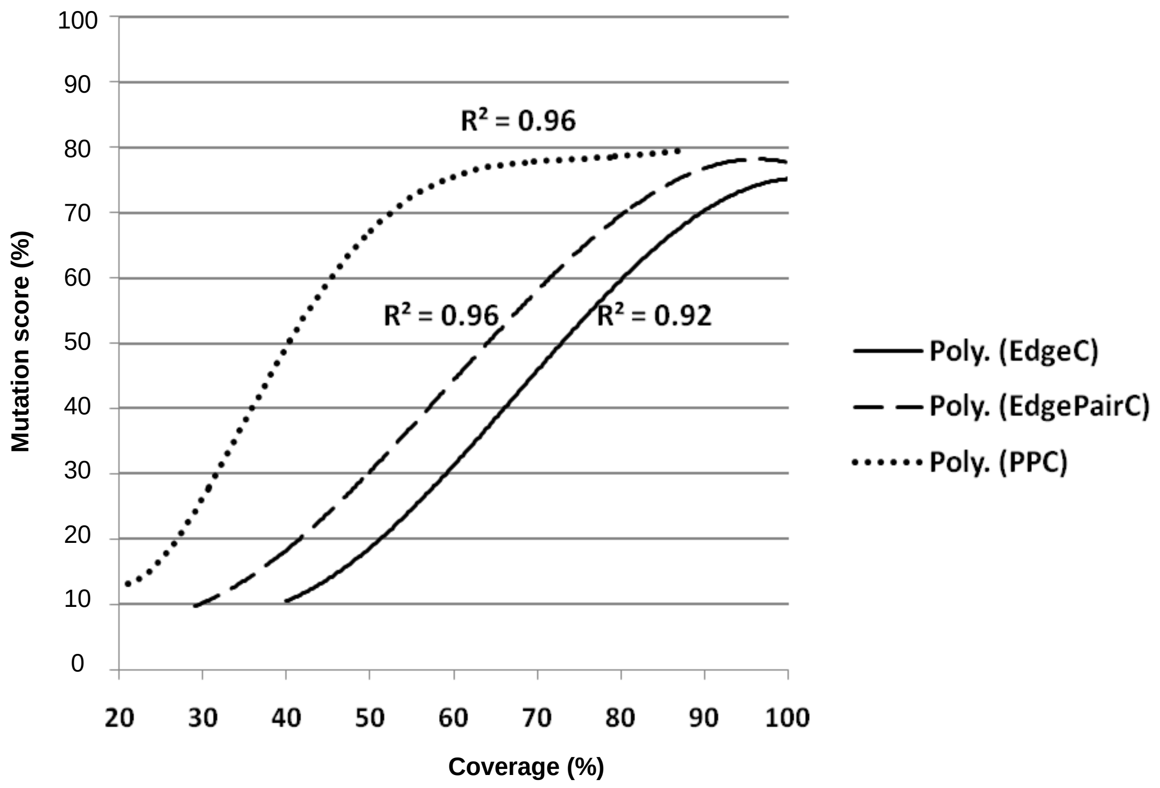


**Figure 2.** Mutation scores versus coverage levels for P4.

combine these criteria with other software testing methods such as logic-based criteria when we need higher levels of fault detection rates. Comparing maximum effectiveness of Edge, Edge-Pair and Prime-Path coverage criteria, they detect almost the same number of faults in the programs P4 and P8. But in general, it is observable that Prime-Path coverage criterion is more effective than the other two criteria. Edge-Pair reveals more faults than Edge coverage criteria. This result is consistent with what the subsumption relationships say about these three criteria (Ammann and Offutt, 2008).

Exceptions are because of the moderately or highly deviated values from the mean. This makes these results unreliable. As we will discuss in 'reliability of path-based testing methods', whereas we have high dispersion in the results of some programs, average of mutation scores of test suites for these programs is not representative of fault detection power of the evaluated criteria. In these cases, we have test suites with the same level of coverage but with significantly different mutation scores. Pearson correlation analysis shows weak positive correlation between mutation scores and criteria coverage levels for these test suites. In practice, generating test suites to cover all feasible test requirements is not possible because recognizing infeasible requirements is an undecidable problem (Ammann and Offutt, 2008) and test engineers do not know about most achievable coverage level. In addition, finding test inputs for all feasible requirements is a very expensive process. Thus, comparing the effectiveness of test suites with lower coverage levels is important too. Figure 2 demonstrates mutation scores of test suites generated to satisfy coverage criteria for the program P4. Polynomial regression lines estimate results well ($R^2 > 0.9$).

It shows that achieving the same level of coverage leads to significantly different fault detection rate for each criterion compared to the others. Prime-Path covering test suites are significantly more effective than Edge and Edge-Pair suites. It also shows that only high level coverage of Edge-Pair and Edge is effective for these two criteria but Prime-Path reaches its maximum power near the last one third of its feasible coverage level. Except P6 and P7 that are discussed in the following section, similar results have been attained for other subject programs.

## Reliability of path-based testing methods

Graph coverage criteria, such as those studied in this paper, guide test engineers to select a lot of paths as test requirement from extracted control flow graph and try to find test inputs to satisfy these test requirements.

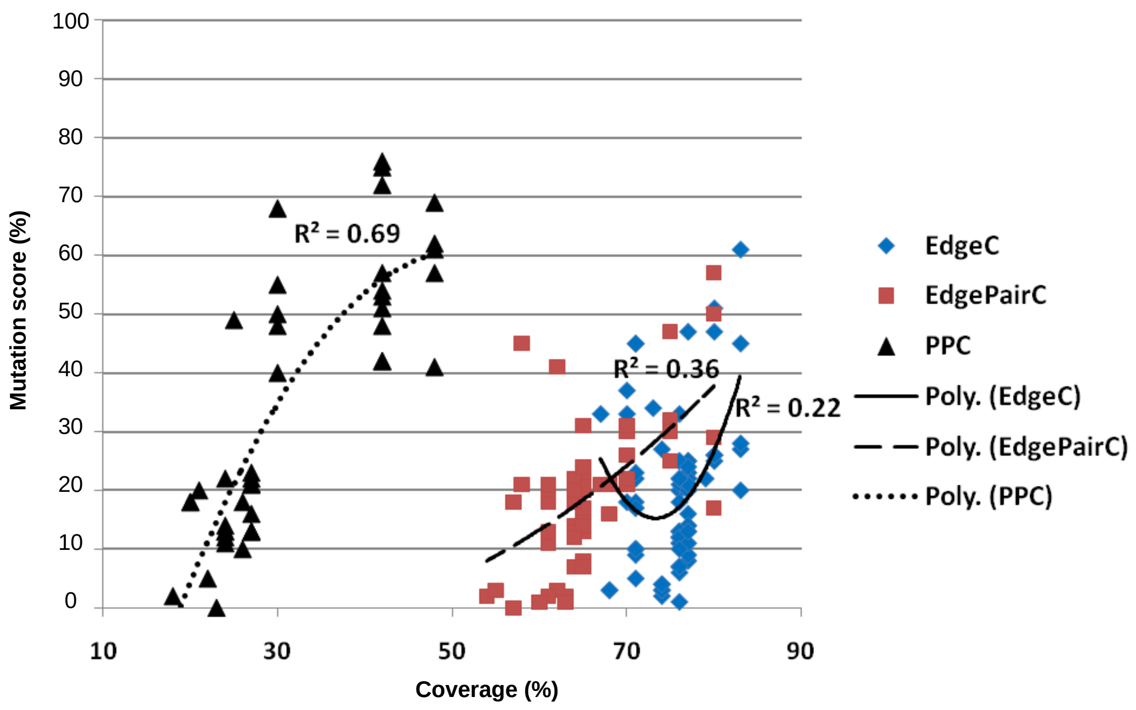


**Figure 3.** Mutation scores of coverage criteria on P5p.

"Formally, given a set *TR* of test requirements for a graph coverage criterion *C*, a test set *T* satisfies *C* on graph *G* if and only if for every test requirement *tr* in *TR*, there is at least one test path *p* in *path* (*T*) such that *p* meets *tr*" (Ammann and Offutt, 2008). This includes structural and data flow criteria in general. On the other hand, we know that All-Path coverage criterion subsumes all other graph coverage criteria. This statement implies that if our test suite covers all feasible paths in the CFG, such a test suite will reach maximum fault detection rate that is expected for a graph (path-based) coverage criterion. Results of the experiment here confirm this statement. Figure 1 shows that mutation scores of the All-Path covering test suites are higher than any other criteria for all programs. The result holds for all test suites with low standard deviations from the mean. However, the problem is where we observe that for most programs, All-Path covering test suites are unable to detect high rates of seeded faults. They leave about 10% of faults undetected for programs P1 and P7a, about 20% for P3, P5j, and P5p, and up to 50% for P7. Since there is no other pure graph coverage criterion to be more powerful than All-Path, this is the maximum effectiveness of such criteria for these programs.

Studying the results from another point of view will guide us to the underlying reason. For some programs in this experiment, we observe high levels of deviation from the mean of mutation scores for test suites with the same criterion coverage percentage. As illustrated in Figure 3, test suites generated for the program P5p not only have low mutation score, but also suffer from high dispersion and unreliability. Results cannot be correlated around a regression line with an acceptable confidence level. For example, test suites which are covering Edge-Pair test requirements have mutation scores between 17 and 57% and we cannot see any convergence in the results. This is also true for two other criteria and for some other programs such as P3, P5j and P6. P7 also demonstrates similar results but its test suites converge when they reach near the maximum feasible coverage level. These results are also because of the same reason: traversing the same execution paths does not mean killing the same mutants and revealing the same set of faults. Therefore, selecting only one test input to satisfy each feasible path in the graph will not detect all faults in the program.

In this paper, we cannot discuss all aspects of this non-deterministic behavior but we will show the reason with two simple examples. Suppose that the correct version for a part of our program is as follows:

*int remainder Five (int year) {return year % 5}*

*and the faulty code is:*

*int remainder Five (int year) {return + year % 5}*

If we select {year = 3} as test input, it will not reveal the fault in the faulty code, but test input {year = 4} detects the fault. These two test cases traverse the same path in the graph. One of them is effective on revealing seeded

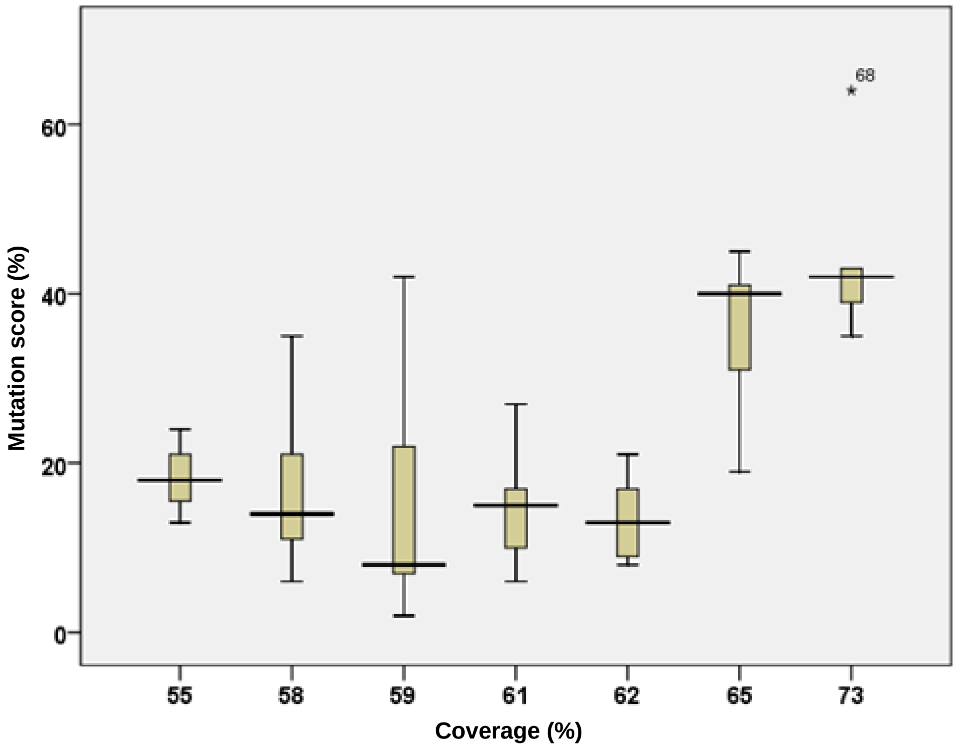


**Figure 4.** Mutation score dispersion of Edge-Pair covering suites generated for program P5j.

fault but the other one is not. As another example, assume that the correct version of another code should be:

*if (c1 & c2) {do something}*
*else {do another thing}*

where c1 and c2 are its boolean inputs. We also have an implemented code with a fault as follow:

*if (c1 || c2) {do something}*
*else {do another thing}*

A test case with {c1 = true, c2 = true} as its inputs will not detect the fault and the program acts as the correct version. However, the test input {c1 = true, c2 = false} will reveal the fault in the implemented code. These two test cases also cover the same execution path in the graph of the implemented code but they are different in fault detection. In this example, this is the cause for the existence of multiple-condition decision statements in the program. This issue was discussed in ‘input domain’. However, this type of problems is not limited to branches and decision-statements; all logical expressions may have this non-deterministic behavior on fault detection when we select test cases with graph coverage criteria. Similar parts of code exist in the source code of the programs in this study and lead to results mentioned earlier for some programs. Figure 4, as an example, demonstrates mutation scores dispersion of test suites generated for Edge-Pair coverage criteria for program P5j, using box plot diagram. It shows high dispersion in the results even for mutation scores that cover all feasible test requirements (73% coverage). In addition, running Pearson correlation analysis on these data shows significant (p-value $< 0.001$) but weak correlation between coverage level and mutation score variables (correlation coefficient $r = 0.41$). This result is also valid for Edge and relatively Prime-Path coverage criteria of this program and similar results have been observed for programs P3, P5p, P6 and P7. The exception is Prime-Path covering test suites in programs P5j and P5p which have higher correlation coefficients ($r > 0.8$).

In such cases that we have test inputs that traverse the same execution path but detect different set of faults in the program, using path-based criteria prevents test engineers from selecting test cases to reveal a lot of faults. Graph coverage criteria such as Edge and Prime-Path motivate test engineers to efficiently select only one test case to satisfy each test requirement which is a path in the extracted control flow graph. In these situations, relying on such criteria may have negative impact on the effectiveness of software testing process. Because of this issue, as discussed in ‘competitive random test suites’, even random test suites may be more effective than using graph coverage criteria in programs with such property. This fact raises the need for some metrics and criteria to check whether or not a coverage criterion is expected to be effective for a program under test?

## Cost-effectiveness of coverage criteria

Since achieving high levels of coverage criteria is expensive, cost-effectiveness analysis expresses which criterion is more effective according to the cost of building test suites. Because measuring real cost of building test suites to satisfy each criterion is a complex problem, like similar studies (Andrews et al., 2006), we take the size of test suites as the cost of building the suites. Using this assumption in comparing test suites with the same size (cost), which are generated for two coverage criteria, a criterion which gains higher mutation score is more cost-effective than the other one. Figure 5a shows relationship between mutation score and the size of test suites for the program P2 as an example. In this diagram, results are estimated well with logarithmic regression lines ($R^2 > 0.8$). We can see that Edge coverage is more cost-effective than Edge-Pair, and Edge-Pair is more cost-effective than Prime-Path coverage criterion. But, Prime-Path continues its way to achieve higher mutation score at the end.

Except where the results are not reliable and highly deviate from the mean, this result is consistent among all subject programs, especially for small size test suites. The difference between cost-effectiveness of these three criteria is statistically significant for all subject programs, but the difference is small in practice and cannot be considered practically significant. For program P8, the difference reaches zero in the middle of the way and continues vice versa. Prime-Path overcomes the two other criteria when the size of test suites is greater than 60 in this program (Figure 5b).

## Competitive random test suites

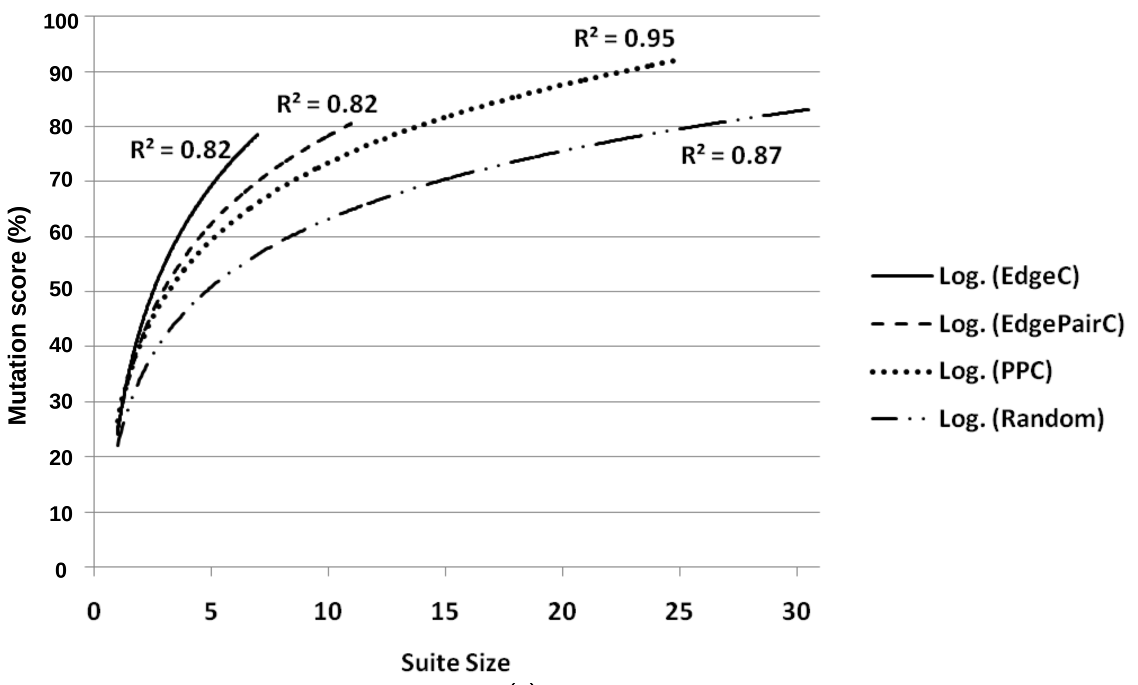
Studying mutation score of random-generated test suites shows that these test suites are not successful in detecting lots of seeded faults in the programs. In analyzing the cost-effectiveness of random test suites with respect to coverage criteria suites, it is observed that in all programs, random suites are less effective than the criteria-based generated suites (Figures 5a and b). In other words, when we select random test suites with the same size of suites generated to satisfy coverage criteria, their fault detection rates are significantly lower than the rates of the criteria-covering suites. But, when we increase the size of random suites (up to 31 test cases in this experiment for P1 to P7a), their mutation scores are comparable to the maximum achieved mutation score of some coverage criteria suites. One reason for this success of random test suites in this experiment is the nature of the subject programs and their input domain distribution. Their input domain is relatively symmetric and has about normal distribution on traversing execution paths in the programs. Thus, random-generated test suites will cover a lot of paths in the graph and can achieve a competitive mutation score compared to coverage criteria suites. However, random test suites cannot reach high levels of fault detection rate.

Using coverage criteria helps test engineer to select test cases such that they cover paths that would be traversed rarely by random inputs. For example, using coverage criteria in program P8 significantly raises their mutation score upon the mutation scores of random suites with similar size (Figure 5b). Another observation for a number of programs is where random-generated test suites attain higher mutation scores than the test suites generated to satisfy our three coverage criteria. For programs P3, P5j, P5p and P6, some random test suites reach higher mutation scores than criteria-based generated suites. Analyzing this case leads us to the same conclusion made in ‘reliability of path-based testing methods’; when the program under test has several paths and each path needs more than one test case or a special test case to detect its hidden faults, using graph-coverage criteria such as Edge, Edge-Pair or Prime-Path may show satisfaction of the test requirement, but faults still remain in the code. It is particularly likely for paths that traverse a wide subset of the input domain. In this case, random test suites may select more than one test case traversing this path and have more chance to detect faults than test suites with only one test case for the path.

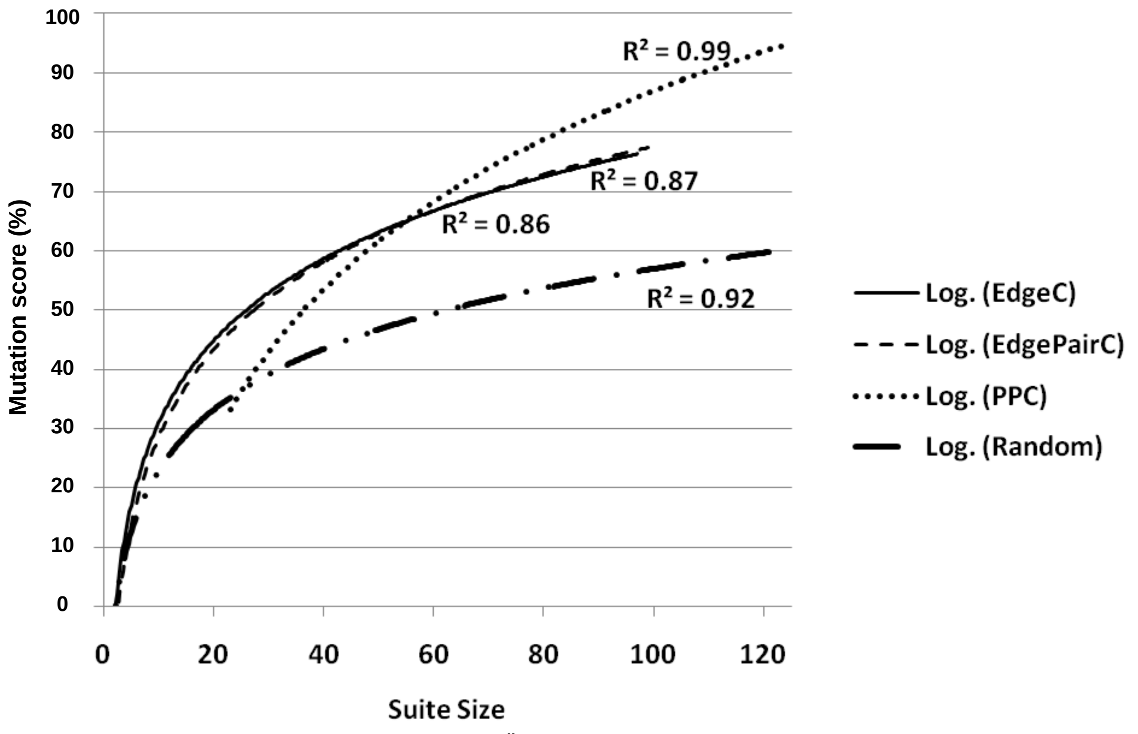
Although, random-generated test suites cannot reveal faults in exceptional and rarely traversing paths, selecting more test cases to traverse primary paths of the program may improve their fault detection rate. Their fault detection rate may be increased to a level even higher than that of the criteria-covering suites, which try to select only one test case for each path to be efficient and cost-effective.

## Applying Sidetrip and Detour

Whereas, there are so many infeasible test requirements for some coverage criteria, it is recommended to cover these requirements with other touring methods like Sidetrip and Detour (Ammann and Offutt, 2008). Analyzing the results in this study shows that for most programs, although adding more test cases to tour infeasible test requirements with Sidetrip and Detour increases criteria-coverage level of test suites, it does not increase mutation score of them significantly. For Edge and Edge-Pair criteria, Sidetrip and Detour were not applicable. It is true for all of the programs. For Prime-Path coverage criterion, Sidetrip and then Detour help the criterion to select more test cases to cover infeasible test requirements, but they are not significantly useful in



(a)



(b)

**Figure 5.** Mutation score versus test suite size for programs a) P2 and b) P8.

increasing the mutation score of the suites for most programs. For programs P3 and P5j, it seems that these two touring methods are effective for Prime-Path coverage criterion. However, the bad news is that adding random test cases also increases the test suites' effectiveness with relatively the same rate for the program P3. It means that indirect touring were not useful in this case too. Thus, in our experimental study, Sidetrip and Detour was significantly beneficial only for P5j, but due to its small difference with respect to random suites, it was not practically significant.

To conclude, we can report that these touring methods were not advantageous for subject programs in this experiment, but in general, they may be useful for other types of programs.

### Effectiveness of cross test suites

Since all subject programs in this experiment are implementing the same problem, we can measure the effectiveness of test suites generated for one implementation on detecting faults of another implementation. The goal of this analysis is to observe whether previously generated test suites are still effective on detecting fault when developers change or re-implement it again. This part of the experiment has been done for programs P1 to P7a due to their equivalent input domain. According to the results, test suites of programs P1 and P2 are effective on almost all programs (except P7) compared to the effectiveness of test suites of the programs themselves. Mutation scores of these test suites on other programs are almost equal or above the mutation scores of self-generated test suites. However, test suites of P3 to P7a are not significantly effective on other programs. In general, results show that we cannot rely on test suites designed for one implementation of a program to test another one.

When a development team decides to make a major change or implement a unit of a program again, test engineers also have to derive new test suites for the new implementation, or at least measure the effectiveness of the previously extracted test suites on the newly implemented unit.

### Threats to validity

Looking for validity issues is a critical analysis of every empirical research in software testing (Briand, 2007). Here, we discuss briefly the external, internal, construct and conclusion validity issues in this study. Although, the selected subject programs are industrial components, but they belong to the same domain, and this is an external threat to the validity of the study. So, it is important to repeat this experiment on other subject programs from other domains. Another threat is in measuring the cost of testing which is assumed to be related to the size of generated test suites. This is a construct validity issue similar to some other related studies (Andrews et al., 2005). In this study, it is tried to mitigate internal and construct validity issues by means of employing all mutation operators supported by MuJava (Ma et al., 2005), using all inputs as test inputs pool, detecting all equivalent mutants, removing easy-to-kill mutants and building several criteria-covering test suites through guided random selection of path-covering test cases from the pool.

## CONCLUSIONS AND FUTURE WORK

This paper reports results of an experimental study on evaluating the effectiveness of four graph-based coverage criteria which are Edge, Edge-Pair, Prime-Path and All-Path for 10 subject programs from a single domain. Employing the entire input domain of the programs in detecting equivalent mutants and infeasible test requirements makes the results more accurate and dependable in the study. Our results show that for all selected graph coverage criteria, the effectiveness of test suites generated to satisfy them highly depends on the program under test. These criteria are effective for some programs, but criteria generated test suites failed to achieve acceptable mutation score for some other programs. The fault detection rate of the Edge, Edge-Pair and Prime-Path coverage criteria are significantly lower than the fault detection rate of the All-Path criterion. Furthermore, employing Sidetrip and Detour methods for indirectly covering infeasible test requirements did not significantly improve the effectiveness of criteria-generated test suites in the experiment. It also shows that only high coverage levels are effective for the Edge and Edge-Pair criteria. However, Prime-Path reaches its maximum power near the last one third of its feasible coverage level. Even though All-Path coverage criterion, as the superior of other graph coverage criteria is expected to reveal significant number of faults, results show that it cannot reach the maximum feasible mutation score. It is due to the fact that test cases which traverse the same path in control flow graph of programs do not kill same set of mutants on that path. Graph coverage criteria, which are based on satisfying path-based test requirements, may stop test engineers to select more than one test case to cover each execution path. Therefore, in some cases, relying on these criteria may have negative impact on the effectiveness of criteria-generated test suites in a software testing process.

According to the results, the difference in the cost-effectiveness of Edge, Edge-Pair and Prime-Path criteria is not practically significant. All of them are more cost-effective than random-generated suites. However, coverage criteria will stop adding more test cases to test suites but random suites continue their way and rapidly reach to competitive mutation scores in some cases. Another experience in this study was selecting different implementations of the same problem as subject programs. It made several advantages. First, the results were more comparable. Secondly, it let us to evaluate cross test suite effectiveness, which is studying whether test suites generated to satisfy a coverage criterion for a program are effective in revealing fault of another implementation of the same problem. Thirdly, test oracle problem could be solved by means of selecting one of the programs as oracle function and applying the majority selection approach to use it as the expected output, or detect variances among their outputs and search for the correct output only for only a small set of test inputs. Finally, ease of experimental automation was another benefit in this manner. Results show that test suites which are effective on detecting faults in one implementation of a problem are not necessarily effective on revealing faults in another implementation of the same problem. Thus, when developers do vast changes in a program or re-implement a unit of the program, based on the results, it is recommended not to rely on test cases generated for the previous implementation.

Further studies are required to confirm the results in this study with subject programs from other problem domains. Since the effectiveness of test suite generated for coverage criteria highly depends on the program under test, we are to find some metrics to recommend how to choose a coverage criterion and also which level of fault detection rate is expected to achieve for a criterion in different situations.

## ACKNOWLEDGEMENTS

The authors acknowledge Jeff Offutt, Paul Ammann and Nan Li for sharing their resources and tools for this study with us. Also thanks to Mostafa Hashemi for sharing the source code of his proprietary calendar-converter component.

## REFERENCES

Ammann P, Offutt J (2008). Introduction to software testing. Cambridge University Press.

Ammann P, Offutt J, Xu W, Li N (2010). Graph Coverage Web Application. Internet: http://cs.gmu.edu:8080/offutt/coverage/GraphCoverage. version Feb. 2010.

Andrews J, Briand L, Labiche Y (2005). Is mutation an appropriate tool for testing experiments? In Proceedings of the 27th International Conference on Software Engineering (ICSE-27), pp. 402-411.

Andrews J, Briand L, Labiche Y, Namin A (2006). Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. IEEE Trans. Softw. Eng. 32:608-624.

Andrews J, Zhang Y (2003). General test result checking with log file analysis. IEEE Trans. Softw. Eng. 29(7):634-648.

Borkowski K (1996). The Persian calendar for 3000 years. Earth Moon Planets 74(3):223-230.

Briand L (2007). A critical analysis of empirical research in software testing. First International Symposium on Empirical Software Engineering and Measurement, pp. 1-8.

Budd T, Sayward F (1977). Users guide to the Pilot mutation system. Department of Computer Science, Yale University, Technical Report 114.

DeMillo R, Lipton R, Sayward F (1978). Hints on test data selection: Help for the practicing programmer. Computer 11(4):34-41.

Hamlet R (1977). Testing programs with the aid of a compiler. IEEE Trans. Softw. Eng. 3(4):279-290.

Jia Y, Harman M (2011). An Analysis and Survey of the Development of Mutation Testing. IEEE Trans. Softw. Eng. 37(5):649-678.

Kim S, Clark J, McDermid J (2001). Investigating the effectiveness of object-oriented testing strategies using the mutation method. Softw. Test. Verification Reliab. 11(4):207-225.

Lee CC (2009). JavaNCSS: A Source Measurement Suite for Java. Internet: http://javancss.codehaus.org.

Li N, Praphamontripong U, Offutt J (2009). An Experimental Comparison of Four Unit Test Criteria: Mutation, Edge-Pair, All-Uses and Prime-Path Coverage. In Proceeding of International Conference on Software Testing, Verification, and Validation (ICST) Workshops pp. 220-229.

Ma Y, Offutt J, Kwon YR (2005). MuJava: An automated class mutation system. Softw. Test. Verification Reliab. 15(2):97-133.

McCabe T (1976). A Complexity Measure. IEEE Trans. Softw. Eng. 2(4):308-320.

Myers GJ, Badgett T, Sandler C, Thomas TM (2004). The art of software testing. John Wiley and Sons.

Northrop Grumman Corporation (2010). Count Lines of Code Tool (CLOC). Internet: http://cloc.sourceforge.net.

Offutt A, Lee A, Rothermel G, Untch R, Zapf C (1996). An experimental determination of sufficient mutant operators. ACM Trans. Softw. Eng. Methodol. 5(2):99-118.

Selvaraj P, Iyer V (2006). Software Complexity Visualiser (CyVis). Internet: http://cyvis.sourceforge.net.

Thévenod-Fosse P, Waeselynck H, Crouzet Y (2002). An experimental study on software structural testing: Deterministic versus random input generation. In Proceedings of 21st International Symposium on Fault-Tolerant Computing (FTCS-21) pp. 410-417.