7-31-2015

An Empirical Study of Off-by-one Loop Mutation

M. S. Raunak
Loyola University Maryland, raunak@loyola.edu

Christian Murphy
University of Pennsylvania, cdmurphy@seas.upenn.edu

Bryan O'Haver
bryan.ohaver@gmail.com

MS-CIS-16-01

An Empirical Study of Off-by-one Loop Mutation

Abstract

Context: Developing test cases that are measurably effective in finding faults in programs is a very challenging research problem. Mutation testing, a prominent technique developed to address this challenge, often becomes com- putationally too expensive for practical use due to the very large number of mutants that need to be analyzed. Objective: This paper evaluates the impact of One-by-one (OBO) loop mutation in reducing the cost of mutation analysis and investigates this technique's effectiveness in measuring the strength or weakness of test suites. Method: A set of Java and C programs have been used to generate both OBO and traditional mutants. Mutation scores are computed and analyzed for both sets of mutants. An analysis of first order vs. higher order loop mutations have also been performed. Results: On average, 89.15% fewer mutants are generated by OBO op- erator in comparison to traditional operators while the two sets of muta- tion scores still remain highly positively correlated (correlation coefficient of .9228) indicating the usefulness of OBO operator in measuring test suite's ef- fectiveness of finding faults in programs. We also investigate the relationship between first order OBO mutation (FOM) and their corresponding higher order mutations (HOM). We have found that OBO HOMs do not subsume their corresponding FOMs. Conclusion: We conclude that One-by-one (OBO) loop mutant operator, which targets specific program elements for mutation, can greatly reduce the number of mutants generated, and thus make the mutation analysis relatively inexpensive and practical while still being capable of providing useful measurement of the strength or weakness of a test suite. Our investigation into the relationship between higher order OBO mutants (HOM) and first order OBO mutants (FOM) has revealed that OBO HOMs usually do not add any value to the mutation analysis over the corresponding FOMs.

Keywords

Mutation Testing, One-by-one, Loop Mutation

Disciplines

Computer Engineering | Computer Sciences

Comments

MS-CIS-16-01

An Empirical Study of Off-by-one Loop Mutation

Mohammad Raunak[a], Christian Murphy[b], Bryan O'Haver

[a]*Loyola University Maryland*
[b]*University of Pennsylvania*

Abstract

Context: Developing test cases that are measurably effective in finding faults in programs is a very challenging research problem. Mutation testing, a prominent technique developed to address this challenge, often becomes computationally too expensive for practical use due to the very large number of mutants that need to be analyzed.

Objective: This paper evaluates the impact of Off-by-one (OBO) loop mutation in reducing the cost of mutation analysis and investigates this technique's effectiveness in measuring the strength or weakness of test suites.

Method: A set of Java and C programs have been used to generate both OBO and traditional mutants. Mutation scores are computed and analyzed for both sets of mutants. An analysis of first order vs. higher order loop mutations have also been performed.

Results: On average, 89.15% fewer mutants are generated by OBO operator in comparison to traditional operators while the two sets of mutation scores still remain highly positively correlated (correlation coefficient of .9228) indicating the usefulness of OBO operator in measuring test suite's effectiveness of finding faults in programs. We also investigate the relationship between first order OBO mutation (FOM) and their corresponding higher order mutations (HOM). We have found that OBO HOMs do not subsume their corresponding FOMs.

Conclusion: We conclude that Off-by-one (OBO) loop mutant operator, which targets specific program elements for mutation, can greatly reduce the number of mutants generated, and thus make the mutation analysis rela-

Email addresses: `raunak@loyola.edu` (Mohammad Raunak), `cdmurphy@seas.upenn.edu` (Christian Murphy), `bryan.ohaver@gmail.com` (Bryan O'Haver)

tively inexpensive and practical while still being capable of providing useful measurement of the strength or weakness of a test suite. Our investigation into the relationship between higher order OBO mutants (HOM) and first order OBO mutants (FOM) has revealed that OBO HOMs usually do not add any value to the mutation analysis over the corresponding FOMs.

Keywords: Mutation Testing, Off-by-one, Loop Mutation

1. Introduction

An important research challenge in software testing is to measure the effectiveness of a test suite in revealing program errors or faults and to find ways to guide the development of a stronger test suite. Mutation testing emerged as a promising technique in this regard, and has received growing attention in recent years [1]. In this approach, numerous simple syntactic faults are systematically inserted into a program P. Each individual syntactic fault results in a separate version of the program, P_{m_i}, which is called a mutant of P. The rule used for creating P_{m_i} is called a mutation operator. For example, an arithmetic mutation operator can alter a '+' to a '−' to produce a mutant. Once P_{m_i}s are created using a set of mutation operators, test suites are run against each of the mutants. If the test suite fails over a mutant P_{m_i}, it is said to have been *killed*. Otherwise the mutant is considered to remain *live*.

An altered program P_{m_i} is called an equivalent mutant if it is semantically equivalent to the original program P. In this case, all test cases will produce the same result for both P_{m_i} and P, i.e., the mutant cannot be killed by any test case. The mutation testing process measures how strong a test suite is by computing its *mutations score* - the ratio of the number of mutants killed by the test suite with respect to the total number of non-equivalent mutants generated. The goal of mutation analysis is to have a test suite with a high mutation score, which is indicative of a stronger test suite. If the mutation score is low, however, it points to a relatively weak test suite, which is likely to fail in discovering many faults in the program. When the mutation score is low, test cases are added to kill the previously live mutants and thus to systematically strengthen the test suite. Thus mutation testing provides a systematic process for developing strong, effective test suites. The term *mutation coverage* is used to specify a test adequacy criterion, where the mutation score defines the level of coverage achieved by a set of test cases.

In mutation testing, the synthetically created mutants are hypothesized to be representative of real faults introduced by programmers in their software [2]. Here the idea is that programmers tend to make small syntactic errors instead of making gross mistakes in their programs. This is known as the "competent programmer hypothesis." [2]. Another hypothesis underlying mutation testing approach is that a test suite's ability to detect small simple faults is correlated to its capacity in revealing major, complex faults (the "coupling effect") [2][3]. Both of these hypotheses have been studied theoretically and validated to some degree using empirical studies [4][5].

There are two major limitations, however, of this potentially very useful mutation analysis technique: a) the high computational cost of executing a very large number of mutants against the test suite; and b) the detection of equivalent mutants, which often requires human effort. Researchers have made a number of different efforts in reducing the cost of using mutation testing. These efforts have focused primarily on reducing the number of generated mutants, and enabling faster and cheaper execution of test cases [1]. In recent years, researchers have also looked into creating mutants by inserting more than one fault in a mutant program, known as higher order mutation or HOM [6].

In this research, we present a novel mutant reduction technique, *Off-by-one (OBO) loop mutation*, in which instead of applying a traditional set of mutation operators at all potential program points, we only alter the for-loop headers. In a classic study of programmer errors, Edward A. Youngs showed that one of the more common programmer errors is related to unsuccessful iterations [7]. A common fault often found in loops is that it is iterated one less than or one more than the required number of times. This type of error or fault is popularly known as *off-by-one (OBO)* error. In this research, we have studied the usefulness of using OBO as a mutation operator.

The main contributions of our research presented here are as follows:

- Our study empirically evaluates the impact of OBO loop mutation in measuring the effectiveness of test suites.
- Our results provide some empirical validation of the coupling effect between first-order and second-order OBO loop mutations, which aligns with some parts of theoretical study presented by Wah[5].

2. Approach

There are seven possible mutations that can be made altering the for-loop header related to the index variable. These involve two errors in initialization (EI+1 and EI-1), two errors in comparing against the sentinel value (ES+1 and ES-1), two shift errors (SH+1 and SH-1), and a shrink error (SK). The following list shows the mutations introduced by our OBO mutation operator for an original loop declaration of the form:
for (int bar = 0; bar < N; bar++)

- Error in Initialization +1 (EI+1): *for (int bar = 0+1; bar < N; bar++)*
- Error in Initialization -1 (EI-1): *for (int bar = 0-1; bar < N; bar++)*
- Error in Sentinel Value +1 (ES+1): *for (int bar = 0; bar < N+1; bar++)*
- Error in Sentinel Value -1 (ES-1): *for (int bar = 0; bar < N-1; bar++)*
- Shift Error+1 (SH+1): *for (int bar = 0+1; bar < N+1; bar++)*
- Shift Error-1 (SH-1): *for (int bar = 0-1; bar < N-1; bar++)*
- Shrink Error (SK): *for (int bar = 0+1; bar < N-1; bar++)*

Ideally, each mutant program should be created with only one simple alteration of the code such as EI+1, EI-1, ES+1 and ES-1. When multiple mutations are done to create a mutant program, it is known as a *Higher Order Mutation* (HOM). Thus SH+1, SH-1 and SK are HOMs, as each of them involves two first order OBO mutations. In particular: SH+1 is a combination of EI+1 and ES+1; SH-1 is a combination of EI-1 and ES-1; and SK is a combination of EI+1 and ES-1.

3. Experimental Design

In this study, we investigate the following research questions:

- **RQ1**: Are OBO loop mutation operators effective in evaluating the strength or weakness of a test suite compared to other traditional mutation operators?
- **RQ2**: Do higher order OBO loop mutations subsume any of the first order OBO loop mutations?

3.1. Mutation Tools and Process

We developed an application to generate OBO mutants in programs written in Java and C. Our OBO mutant generator, which is implemented in Java, reads in the source code, identifies locations where for-loops exist, and then performs some syntax checking to determine whether to mutate the loop or not. For example, mutations are not made to *for-each* loops as there is no opportunity for programmers to insert a traditional OBO type error. Next, the tool determines if the index variable can be incremented or decremented by checking its type against a set of predefined types. The types that are considered for mutation include *int*, *double*, *long*, *short*, *byte*, and *float*, as well as the *Integer*, *Double*, *Long*, *Short*, *Byte*, and *Float* wrapper classes for Java.

While some other types could be incremented, such as nodes on a list or user defined data types, the method to iterate these must be determined on a case-by-case basis. Whenever our tool decides against an automatic mutation for a loop, it is saved in a log file for later manual inspection and creation of mutation through human intervention as necessary.

The tools generate OBO mutants judiciously by applying the OBO operator only where it is reasonable. For example, if the body of a for-loop does not use the index variable at all, the loop does not generate the higher-order Shift mutations, and it only generates Error in Initialization (EI+1 or EI-1), or Error in Sentinel Value (ES+1 and ES-1) mutations, but not both. In such loops, the index variable is only counting how many times the loop iterates. If the original loop iterates N times, shift mutations would create an *equivalent mutant* that would still iterate N times. Our mutation generation program thus tries to avoid generating these types of equivalent mutants.

For applying traditional mutant operators in our study, we used PIT [8], a fast mutation tool for Java programs. PIT creates the mutations at the byte code level. There are ten groups of mutation operators available in PIT including conditionals boundary, the negation of conditions, mathematical operators, increment operators, constant replacements, return value mutator, void and non-void method calls, and constructor calls. For the C programs, we wrote our own tool to generate mutant programs that provides the following mutation operators: arithmetic operators, comparison operators, boolean operators, assignment operators, and increment operators.

3.2. Subject Programs

We have used two sets of programs in this study: a) a set of four open source Java libraries with JUnit test suites and b) a set of five C programs. Table 1 shows the list of programs used in our study. In addition to listing the size of the programs in LOC and in number of classes or procedures, we also provide the number of test cases included in their test suite or the number of data sets we have used while testing the programs.

Table 1: Programs Used in Experiments

Program	Classes/functions	LOC	Test Cases/Test Data Sets
Jtopas	50	5400	101
HTML Parser	245	47000	805
XML Security	225	16800	678
JMeter	389	43400	92
Totinfo	7	565	944
Grep	146	10068	139
Space	136	6199	1349
C4.5	141	5285	16
MartiRank	19	804	20

Amongst our Java subject programs, we have *JTopas*, which is a Java tokenizer and parser tool. *HTML Parser* is a library used to parse HTML documents. *XML-Security* is a security and encryption tool for XML. Finally *JMeter* is an application designed to load-test functional behavior and measure performance of software.

For our C programs, we have studied *Totinfo*, a program originally created by Tom Ostrand and his colleagues at Siemens Corporate Research to study fault based software testing. It is part of the Siemens test suite. *Grep* is the popular UNIX shell utility. *Space* is an interpreter for an array definition language (ADL). *Space* has had by far the most comprehensive set of test cases associated with it. We will also see an impact of this in the mutation analysis of the program. *C4.5* is a statistical classifier used for generating decision trees from a set of training data [9]. *MartiRank* is a supervised machine learning ranking algorithm first presented by Gross et al. in their study of predicting electrical feeder failure [10].

Except for *HTML Parser*, all Java programs were collected from the software-artifact infrastructure repository (SIR)[11]. Four of the C programs

(Tot-Info, Grep, Space and MartiRank) were also collected form SIR. We collected C4.5 directly from its author Ross Quinlan's website. Each of these programs has been used in a number of different mutation analysis and other software testing related studies [1].

3.3. Methodology

As the first step of experimenting with our selected test programs, we made sure that the programs passed all their associated test cases before any mutation was applied. We then applied two types of mutations on each of the subject programs: a) the OBO loop mutation operators introduced by us in this study and b) a set of traditional mutation operators as described in section 3.1.

Once the mutants were generated, we ran each of the mutated programs against its set of test cases, collecting data on whether all tests passed or if there was any failed test case. If the program resulted in a runtime error, we excluded the mutant from our data. Although we could also include the runtime errors as cases where mutants are killed, excluding those data points gives us a more conservative score about the effectiveness of the mutation operators. Since our study investigates the relative effectiveness between OBO and traditional operators, the decision of excluding runtime errors from collected data does not impact the findings.

4. Results and Analysis

4.1. Impact of OBO Mutants

To answer RQ1, we looked into the effectiveness of OBO mutation analysis in identifying the strength or weakness of a test suite compared to other traditional mutant operators.

We begin by investigating OBO loop mutant operators' impact on mutant reduction. As shown in the rightmost column of Table 2, OBO mutation substantially reduces the number of mutants compared to traditional mutation, ranging from 63.95% for JTopas to 98.08% for Space. For the nine target programs, OBO mutation reduced the number of mutants by 89.15% overall.

To demonstrate the OBO mutations are effective in evaluating the strength or weakness of a test suite compared to traditional mutation operators, we computed the Mutation Score MS for each technique using the formula:

$$MS = \frac{MU_{killed}}{MU_{tot} - MU_{equiv}}$$

Table 2: Comparison of OBO and Traditional Mutation Operators

	OBO Loop Mutation Operators			Traditional Mutation Operators			
Program	# non-equiv Muts	Killed	Mut Score	# non-equiv Muts	Killed	Mut Score	Reduction
JTopas	84	65	.7738	233	121	.5193	63.95%
HTMLParser	1470	559	.3803	6507	2115	.3250	77.41%
XMLSec	1164	413	.3548	4833	1298	.2686	75.92%
JMeter	2760	488	.1768	23736	937	.0395	88.37%
TotInf	82	71	.8659	1894	1609	.8495	95.67%
Grep	839	105	.1251	12483	3176	.2544	93.28%
Space	101	101	1.00	5273	5273	1.00	98.08%
C4.5	406	324	.7980	8539	5203	.6093	95.25%
MartiRank	287	148	.5157	2802	796	.2841	89.76%

In this equation, MU_{tot} is the total number of mutants generated, MU_{killed} represents the number of mutants killed by a test suite or test data, and MU_{equiv} is the number of equivalent mutants that were generated. The mutation score MS is a measurement of how strong the test suite is in its ability to catch program errors or faults. A higher mutation score is indicative of the test suite being more effective in revealing faults. By contrast, a lower mutation score points toward a relative weakness of the test suite or test data in identifying potential faults in the programs.

Table 2 shows that mutation score from using OBO operator (MS_{obo}), in general, is higher than mutation score from using traditional mutation operators (MS_{trad}). This means that traditional mutant operators, which is a larger set compared to OBO operators, provide a more conservative, and often more accurate, measure of a test suite's fault revealing strength. However, this information comes at a cost of often extraordinarily large number of mutants getting generated as part of computing MS_{trad}, which can make the mutation analysis too computationally expensive for practical use. On the other hand, OBO loop mutant operators produce much fewer mutants, while still providing a useful mutation score MS_{obo} indicative of the strength or weakness of the test suite. The relatively higher MS_{obo} compared to the MS_{trad} means that OBO mutation operator gives a more optimistic score about the strength of a test suite than the traditional operators. We note, however, that there is a strong positive correlation between MS_{obo} and MS_{trad}. From the set of experiments we have performed, we have found a correlation coefficient of .9228 between OBO loop mutant operators and the much larger set of traditional mutant operators. Figure 4.1 depicts the correlation between the two sets of mutation scores.

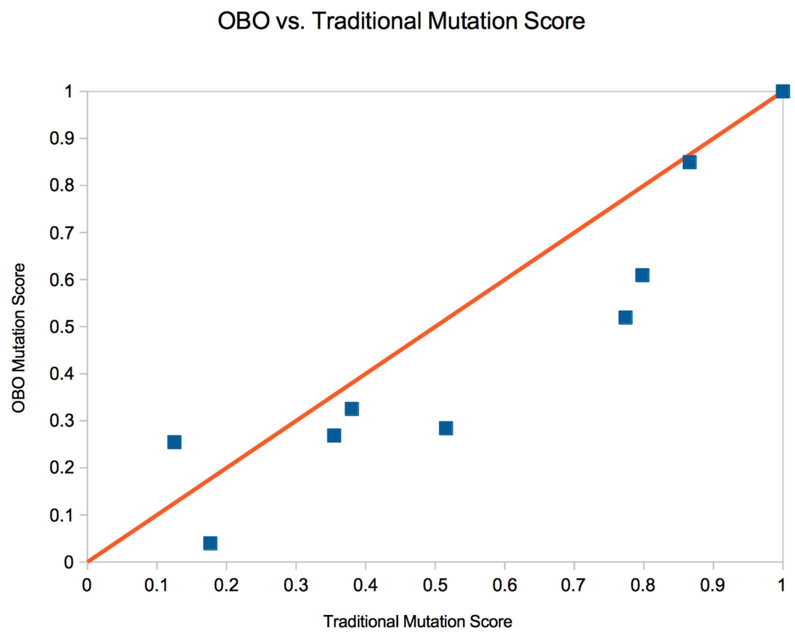

Figure 1: Comparison of mutation scores of OBO and traditional mutant operators. Although the OBO scores tend to be higher, they still show strong correlation to the traditional scores.

Unlike all other subject programs, in the case of 'grep', we found that MS_{obo} was lower than MS_{trad}. In other words, OBO operator is indicating a more conservative measure of test suite effectiveness than traditional operators for 'grep'. We conjecture that this is possibly a result of how the test cases were developed for this particular subject program. The software-artifact infrastructure repository (SIR) [11], from where this program was taken, notes that the test cases were generated by seeding errors in the code and then writing test cases specifically to reveal those errors. The seeded errors were probably more representative of traditional mutant operators than

OBO operators. Hence the test suite was capable of killing more traditional mutants compared to OBO mutants.

The subject program ‘Space’ shows another special case scenario. This subject program came with by far the most comprehensive test suite. We only ran a randomly selected subset of the test cases and still found that set of test cases to have a 100% mutation coverage, i.e., they killed all the mutants generated using the traditional mutation operators. Not surprisingly, the same set of test cases also killed all OBO loop mutants.

Our research question 1 (RQ1) aimed at investigating the effectiveness of OBO loop operators in evaluating the strength or weakness of a test suite as compared to the traditional operators. Using table 2 and from the above discussion, we can summarize our findings to address RQ1. The OBO loop mutant operator significantly reduces the number of mutants generated while still providing a useful indication of the strength or weakness of a test suite. The OBO mutations scores are highly correlated with the findings one would get from the much more expensive option of using all traditional mutation operators. Thus, although OBO mutant operators cannot directly replace all traditional mutants, they can be a practical, relatively inexpensive, and useful technique to measure the strength of test suites.

4.2. Higher order OBO Loop Mutation

In section 2, we introduced three Higher Order OBO Mutations (HOM): the SH+1 (shift forward), SH-1 (shift backward), and SK (shrink). Our second research question (RQ2) sought to investigate whether these HOMs provide any additional information about test suites’ strengths compared to the two First Order Mutations (FOM) that comprise them. For this particular study, we have used only the Java subject programs. We collected our data by running the test suite only against the HOMs vs. the FOMs that comprise them. We can describe the effectiveness of the HOMs as follows:

- More effective: The HOM is more effective than its corresponding FOMs if both FOMs are killed while the the HOM lives. In this case, the HOM provides additional information about the weakness of the test suite not captured by the FOMs.

- Less effective: The HOM is less effective than its corresponding FOMs if at least one of the two FOMs lives and the HOM is killed. This means that the HOM does not reveal a weakness of the test suite while the one of the two FOMs did.

- Equally effective: The HOM is equally effective compared to its corresponding FOMs if the HOM lives and at least one of the FOMs also lives. This means that both the HOM and at least one of the FOMs reveals a weakness of the test suite.

- No information: If both the HOM and each of its corresponding FOMs are killed, then no specific information about the relative effectiveness of HOM vs. FOM can be deduced. This is because all the mutants failed to reveal any weakness of the test suite in this scenario.

We analyzed our data for the results of running each HOM against the results of running the two corresponding FOMs. Table 3 presents the combined results from the four Java programs we tested.

Table 3: Higher Order Mutation Analysis

The HOM is..	EI+1,ES+1 vs.SH+1	EI-1,ES-1 vs.SH-1	EI+1,ES-1 vs. SK
More effective	0	1	0
Less effective	50	28	46
Equally effective	33	28	94
No information	42	32	126

We found only one instance where a HOM lived while both its corresponding FOM were killed by the test suite. This suggests that HOM OBO operators are rarely useful in providing additional information than what is already known from FOMs. Additionally, we have found that OBO HOMs do not subsume their corresponding FOMs. It is not uncommon where one of the FOM lives to indicate a weakness in the test suite while the same test suite kills the corresponding HOM. This empirical finding aligns with some of the theoretical findings presented by Wah in [5].

The above analysis also implies that we can generate even fewer OBO loop mutants without losing much effectiveness in judging the strength of a test suite. By eliminating three of the seven OBO mutations we introduced (SH+1, SH-1, and SK), we can reduce the total number of mutants being generated by 3/7 = 42%. The resulting mutants, as shown in table 4, still produces mutation scores that closely track the MS_{obo} with all the OBO operators. In fact, using only the FOM operators consistently provide a lower mutation score for each of the test suite. In other words, FOMs provide a more conservative and thus more accurate measure about the weakness of the test suite.

Table 4: Comparison of First-Order Mutants (FOMs) and Higher-Order Mutants (HOMs)

Program	Total OBO Mutation Score	FOM-only OBO Mutation Score
JTopas	0.7738	0.7500
HTMLParser	0.3802	0.3530
XMLSec	0.3548	0.3343
JMeter	0.1768	0.1675

5. Related Work

Since the introduction of the idea in the 1970s [12][2], mutation testing has seen a wide range of interest over the years [1]. Even though mutation testing technique provides one of the best known methodologies towards estimating the fault-finding ability of a test suite, it has not yet received widespread acceptance in practice due to its high cost. One primary focus of our research has been to tackle the challenge of this high computational cost of mutation analysis by reducing the number of mutants that are generated. Other researchers have also worked towards achieving this goal over the years. The techniques applied by researchers toward mutant reduction can be categorized into four groups: a) sampling, b) clustering, c) selective subset, and d) using higher order mutations.

In mutant sampling, a randomly selected subset of mutants is used for mutation analysis. DeMillo and Offutt developed the first widely used mutation analysis tool, Mothra, which applied 22 mutation operators on Fortran programs [13]. Wong and Mathur's study used random sampling to select mutants from the full set of mutants [14]. King, Offutt and Demillo have done similar studies later [13] [15] with various degrees of success.

Instead of selecting mutants randomly, another approach has been to use clustering techniques to divide mutants in different clusters and then select a small number from each cluster [16]. Another approach, known as selective mutation, selects a subset of all mutation operators and only applies these operators for creating mutants. The term *N-selective* mutation is used to represent the omission of N operators from the set of well established mutation operators, such as the ones defined in the Mothra system. Wong and Offutt have worked separately with 2, 4, and 6-selective mutations [17, 18], where they have looked at mutation analysis by removing 2, 4 and 6 operators from the original Mothra mutation operators.

In a study by Jia and Harman [6], higher order mutations have been shown to subsume some of the first order mutations that comprise them. That is, any test suite that would kill an HOM would also kill the corresponding FOMs. Our study reveals a different scenario for OBO mutants than what Jia and Harman found for some of their HOM mutant operators.

One important aspect of many of the above mentioned studies is that they were applied on very small programs (a few dozen lines of code), often written in languages like Fortran. In our research, we have worked with significantly larger programs written in Java and C. Although our approach is similar to selective mutation, instead of selecting a subset of traditional mutation operators, we have focused on a specific program unit to mutate, specifically for-loops. There is no other study that we are aware of which has looked at such program element based mutation.

6. Threats To Validity

The threat to the internal validity of our experiments is limited. The only independent variable in the experiments has been the types of mutation operators used, which has resulted in different numbers of mutants being generated with corresponding mutation score. The causal relationship between the mutant operators and the number of different mutants generated is easy to establish.

There is room for being cautious about the external validity of the findings. The sample size of our empirical study is not very large. The experiments, however, include different types of test programs. There is diversity in the size as well as in the language of the test programs (Java and C). Another limitation of the experiments is that there was a single set of test cases used with each program, namely the ones that came with them. It is conceivable that a different set of test cases may result in finding the mutation scores from OBO operators to be more or less effective as compared to mutation scores from traditional mutant operators. Since most of the test cases were developed along with the programs, they are likely to be representative of unit tests typically written by programmers during the development of these software. Let us also note that the experiment subjects are a well known set of programs that have been used in many mutation and other testing studies. Nevertheless, a larger experimental data set and multiple set of test cases would increase the confidence in the generalizability of the findings.

7. Concluding Remarks

Considering the coupling hypothesis (small syntactic bugs are coupled with larger, more complex bugs), mutation testing has the potential to emerge as one of most effective techniques to develop test cases that are potentially very useful in revealing errors in programs.

We have presented a novel mutant reduction technique, namely using the Off-by-one (OBO) loop mutant operator, which targets specific program elements of software for mutation. Through empirical studies, we have found that such selective mutation on specific program elements can greatly reduce the number of mutants generated, and thus making the mutation analysis relative inexpensive and practical while still being capable of providing useful measurement of the strength or weakness of a test suite.

We have further studied the relationship between the higher order (HOM) and first order OBO loop mutants (FOM) and our data suggests that HOMs add value to the mutation analysis only rarely. On the other hand, HOM OBO operators do not subsume their corresponding FOMs. This suggests that we can reduce the number of generated mutants even more using only first order OBO mutant operators without losing effectiveness in measuring the strength or weakness of a test suite.

In future, it would be useful to extend the HOM vs. FOM investigation using additional programs. We are also interested in studying the possible prioritization of all OBO loop mutant operators in terms of their capacity of revealing errors or faults. Finally we would like to perform additional studies in future using a wider number of subject programs and compare the results to other other mutant reduction approaches.

Acknowledgments

A number of people have directly helped us with this work. We would like to thank Silvia Moreno, Tim Stephani and Nandini Sundara-Raman for helping us with running the experiments and generating the data. We are grateful to Dr. Dave Binkley and Dr. David Broderick for their valuable comments on the initial draft of the paper. We are also grateful to Loyola University Maryland for partially supporting this research through its summer faculty research grant program.

[1] Y. Jia, M. Harman, An analysis and survey of the development of mutation testing, IEEE Transactions on Software Engineering TSE 37 (2011).

[2] R. A. DeMillo, R. J. Lipton, F. G. Sayward, Hints on test data selection: Help for the practicing prgorammer, Computer 11 (1978) 34–41.

[3] A. J. Offutt, Investigations of the software testing coupling effect, ACM Transactions on Software Engineering and Methodology 1 (1992).

[4] J. H. Andrews, . Briand, L. C, Y. Labiche, Is mutation an appropriate tool for testing experiments?, in: Proc. of the International Conference on Software Engineering (ICSE'05), St. Louis, MO.

[5] K. S. H. T. Wah, An analysis of the coupling effect i: Single test data, The Science of Computer Programming 48 (2003).

[6] Y. Jia, M. Harman, Constructing subtle faults using higher order mutation testing, pp. 249–258.

[7] E. A. Youngs, Human errors in programming, International Journal of Man-Machine Studies 6 (1974).

[8] H. Coles, Pit mutation testing, http://pitest.org, 2011.

[9] J. R. Quinlan, C4.5: Programs for Machine Learning, Morgan Kaufmann Publishers Inc., San Francisco, CA, 1993.

[10] P. Gross, A. Boulanger, M. Arias, D. Waltz, P. M. Long, C. Lawson, R. Anderson, M. Koenig, M. Mastrocinque, W. Fairechio, J. A. Johnson, S. Lee, F. Doherty, A. Kressner, Predicting electricity distribution feeder failures using machine learning susceptibility analysis, in: Proc. of the 18th conference on Innovative applications of artificial intelligence, IAAI'06, AAAI Press, 2006.

[11] H. Do, S. G. Elbaum, G. Rothermel, Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact., Empirical Software Engineering: An International Journal 10 (2005) 405–435.

[12] R. Lipton, Fault Diagnosis of Computer Programs, Student Report, Carnegie Melon University, 1971.

[13] R. DeMillo, D. Guindi, W. McCracken, A. Offutt, K. N. King, An extended overview of the mothra software testing environment, in: Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on, pp. 142–151.

[14] W. E. Wong, On Mutation and Data Flow, Ph.D. thesis, Purdue University, West Lafayette, IN, 1993.

[15] A. J. Offutt, A fortran language system for mutation-based software testing, Software: Practice and Experience 21 (1991).

[16] C. Ji, Z. Chen, B. Xu, Z. Zhao, A novel method of mutation clustering based on domain analysis, in: Proc. of the 21st International Conference on Software Engineering and Knowledge Engineering (SEKE'09), Boston, MA.

[17] A. J. Offutt, G. Rothermel, C. Zapf, An experimental evaluation of selective mutation, in: Proceedings of the 15th international conference on Software Engineering, ICSE '93.

[18] W. E. Wong, A. P. Mathur, Reducing the cost of mutation testing: An empirical study, Journal of Systems and Software 31 (1995) 185–196.