Do Redundant Mutants Affect the Effectiveness and Efficiency of Mutation Analysis?

René Just
Dept. of Applied Information Processing
Ulm University
rene.just@uni-ulm.de

Gregory M. Kapfhammer
Dept. of Computer Science
Allegheny College
gkapfham@allegheny.edu

Franz Schweiggert
Dept. of Applied Information Processing
Ulm University
franz.schweiggert@uni-ulm.de

***Abstract*—Mutation analysis is an unbiased and powerful method for assessing input values and test oracles. However, in comparison to other techniques, such as those that rely on code coverage, it is a computationally-expensive and time-consuming method, especially for large software systems. This high cost is due, in part, to the fact that many mutation operators generate redundant mutants that may both misrepresent the mutation score and increase the runtime of the mutation analysis process. After showing how the conditional operator replacement (COR) mutation operator can be defined in a redundant-free manner, this paper uses four real-world programs, ranging in size from 3,000 to nearly 40,000 lines of code, to show the prevalence of redundant mutants. Focusing on the conditional operator replacement (COR) and relational operator replacement (ROR) mutation operators that create 41% of all mutants in the chosen programs, the case study reveals that the removal of redundant mutants reduces the runtime of mutation analysis by up to 34%. Additional empirical results show that redundant mutants can lead to a mutation score that is misleadingly overestimated by as much as 10%. Overall, this paper convincingly demonstrates that it is possible to improve the effectiveness and efficiency of a mutation analysis system by identifying and removing redundant mutants.**

I. Introduction

Mutation analysis is a well-known test adequacy criterion that can assess input values and test oracles through the seeding of artificial faults into a system under test. Even though many prior studies have shown it to be a powerful metric [5], mutation analysis may be prohibitively time consuming and computationally expensive in comparison to other methods, such as those that employ coverage criteria.

Previous studies revealed that a subset of all applicable mutation operators is sufficient to achieve a meaningful result [14], [16]. Regarding the operators to be atomic, these studies focused on reducing the number of mutation operators without incurring a major loss in the accuracy of the mutation score. This paper considers these operators at a fine-grained level and shows that their original definition implies redundancy in the resulting set of mutants. Additionally, the paper demonstrates, by means of a case study, how prevalent those redundancies are in real-world applications and how the inclusion of redundant mutants leads to an inaccurate mutation score, thus making this metric less meaningful. In addition to focusing on effectiveness, the paper empirically demonstrates how reducing the set of mutants decreases the runtime of the mutation analysis process. In consideration of the question raised about the effect of redundant mutants on efficiency and effectiveness of mutation analysis, this paper makes the following contributions:

- A demonstration that the COR operator for replacing conditional binary operators with all valid alternatives should only apply a subset of replacements to avoid the creation of redundant or trivial mutants.
- A determination of the actual number of mutants generated by applying the COR and ROR operators. Using a well-known subset of mutation operators, the case study computes the ratio of the number of COR and ROR mutants to the size of the entire set of mutants.
- A case study that investigates how redundant mutants affect the effectiveness and efficiency of mutation analysis for four real-world programs that range in size from 3,000 to nearly 40,000 lines of code.

Since the contribution of this paper concerns the effectiveness and efficiency of mutation analysis, Section II discusses the basics of this technique and examines the definitions of the mutation operators. Next, Section III furnishes a more detailed view of the mutation operators and proposes a sufficient and minimal set of replacements for the COR operator. Section IV describes the case study, reports on the empirical results, and addresses the potential threats to validity. Section V describes related work and finally, Section VI concludes the paper and outlines future work.

II. Background on Mutation Analysis

Originally introduced by Budd and DeMillo [1], [2], mutation analysis is a fault-based technique for assessing the quality of input values or testing strategies. After methodically seeding faults into an application, a mutation analysis technique runs the test suite to assess its ability to find the injected faults. In contrast to the traditional method of error seeding (cf. [11]), the mutation analyzer systematically injects the faults, thus ensuring that the process is reproducible and unbiased. The application of a mutation operator produces faulty versions of the program under test, referred to as mutants. That is, a mutation operator is a formal description of a program transformation that produces one or more mutated versions of the program.

Table I
SUFFICIENT REPLACEMENTS FOR THE LOGICAL CONNECTOR AND

Literals		**Original clause**	**Sufficient mutations**				**Subsumed mutations**			**Subsumed operator UOI**		
a	b	a && b	false	lhs	rhs	a==b	a ‖ b	a!=b	true	!(a && b)	!a && b	a && !b
0	0	0	0	0	0	(1)	0	0	[1]	[1]	0	0
0	1	0	0	0	(1)	0	[1]	[1]	[1]	1	[1]	0
1	0	0	0	(1)	0	0	[1]	1	[1]	1	0	[1]
1	1	1	(0)	1	1	1	1	[0]	1	[0]	[0]	[0]

Table II
SUFFICIENT REPLACEMENTS FOR THE LOGICAL CONNECTOR OR.

Literals		**Original clause**	**Sufficient mutations**				**Subsumed mutations**			**Subsumed operator UOI**		
a	b	a ‖ b	a != b	rhs	lhs	true	a && b	a==b	false	!(a ‖ b)	!a ‖ b	a ‖ !b
0	0	0	0	0	0	(1)	0	[1]	0	[1]	[1]	[1]
0	1	1	1	1	(0)	1	[0]	[0]	[0]	[0]	1	[0]
1	0	1	1	(0)	1	1	[0]	[0]	0	[0]	[0]	1
1	1	1	(0)	1	1	1	1	1	[0]	[0]	1	1

A wide variety of mutation operators have been proposed for different purposes and programming languages (cf. [5], [9], [10]). However, applying all mutation operators results in a substantial number of mutants, especially for large software systems, and thus executing and analyzing all of the mutants can be very expensive. Responding to this challenge, Offutt et al. studied the effectiveness of a subset of mutation operators, revealing that this smaller group of sufficient mutation operators could be applied without a substantial loss of information [14]. While the actual subset that can be employed depends on the programming language, this paper considers the following set of mutation operators which are supported by an established mutation testing tool [6], [7] and commonly used in previous experiments [15], [16]:

- Operator Replacement Binary (ORB): Replace all occurrences of binary operators with all valid alternatives. Since ORB replaces arithmetic, logical, shift, conditional, and relational operators, the ROR and COR operators studied by this paper belong to this class.
- Operator Replacement Unary (ORU): Replace all occurrences of unary operators with all valid alternatives.
- Literal Value Replacement (LVR): Replace all literal values with a positive value, a negative value, and zero. Additionally, all variables with a reference type are replaced by a reference to `null`.

III. A DETAILED VIEW OF THE COR OPERATOR

Previous studies on the reduction of mutation operators (e.g., [14], [16]) did not take their definition into account and considered the operators to be atomic. This means that, for instance, a replacement operator was either applied with all valid replacements or it was excluded. More recently, Kaminski et al. investigated the relational operator replacement and showed that only four replacements are necessary to subsume all the others [8]. In this section, we focus on the conditional operator replacement for the logical connectors `&&` and `||`. Generally, valid mutations for a conditional expression such as `a <op> b`, where `<op>` denotes one of the logical connectors, include the following:

- `a && b`: Apply the logical connector AND
- `a || b`: Apply the logical connector OR
- `a == b`: Apply the relational operator `a == b`
- `a != b`: Apply the relational operator `a != b`
- `lhs`: Return the value of the left hand side operand
- `rhs`: Return the value of the right hand side operand
- `true`: Always evaluate to the boolean value `true`
- `false`: Always evaluate to the boolean value `false`

To ensure that mutation analysis is effective, it is often important for a mutant to have only a small impact on the output, thus making it hard to detect. Trivial and redundant mutants also should be avoided to reduce the runtime of the mutation analysis process and to not misrepresent the mutation score. This paper refers to mutants that result in a wrong output for all possible input values as trivial mutants.

With respect to the COR operator and the given valid replacements, Tables I and II show all possible mutations for the logical connectors and their effect on the boolean result of the corresponding conditional expression. Besides the original expression, the tables show four sufficient mutations for each expression where all mutants have the least possible impact (i.e., they only change the result of one out of four combinations). The actual changes are highlighted with circles to show that the mutations are disjoint and that their union forms a sufficient set. Since the depicted sufficient mutations collectively cover all possible combinations, they subsume all of the other mutations that have a higher impact, as highlighted by the rectangles. For example, the `lhs` operator produces the wrong output for a certain combination

Table III
INVESTIGATED CASE STUDY APPLICATIONS

	Files	LOC*	Relational Operators	Conditional Operators	Tests
com-math	408	39,991	3,577	428	2,169
com-lang	99	19,495	2,739	695	2,039
com-io	100	7,908	687	139	309
num4j	73	3,647	326	124	218

*Lines of code as reported by sloccount (non-comment and non-blank lines)

of the literals `a` and `b` and all subsumed mutations produce the same wrong output for this combination. Thus, as shown in Tables I and II, a test that can detect the `lhs` mutant also detects all of the other subsumed mutations.

Moreover, the sufficient set of mutations not only subsumes all of the other mutations but also another entire mutation operator, namely the unary operator insertion (UOI). The impact of the UOI operator is again highlighted with rectangles in Tables I and II. It has to be pointed out that the depicted subsumption hierarchy only holds for conditional expressions with one logical connector. This paper does not further investigate composed conditional expressions and leaves this matter open for future research.

IV. CASE STUDY

To examine both the frequency and the effect of redundant mutants, we conducted a case study with four open-source applications. Table III summarizes the investigated applications, showing how they differ in size, complexity, and operation purpose. Since the study focuses on the reduction of mutants associated with applying the COR and ROR mutation operators, the table also gives the counts for the occurrences of relational and conditional operators, in addition to the number of files, tests, and lines of code. The number of tests depicted in the last column of Table III represents the quantity of existing unit tests that are provided and released with the corresponding application.

Throughout the case study, we employed MAJOR, a compiler-integrated tool for the mutation analysis of Java programs, to mutate the applications and to perform the mutation analysis process [6], [7]. MAJOR also provides all relevant data about the number of generated mutants and the necessary runtime for the mutation analysis, thus enabling an investigation of the following four research questions:

- Q1: What is the ratio of the number of mutants generated by the COR and ROR operators to the number of mutants generated by applying all operators?
- Q2: Are conditional expressions with only one logical connector, like those studied in Section III, predominant in real-world applications?
- Q3: What is the actual savings in the runtime of mutation analysis due to the use of the reduced set of mutants?
- Q4: How does the elimination of redundant mutants affect the overall mutation score?

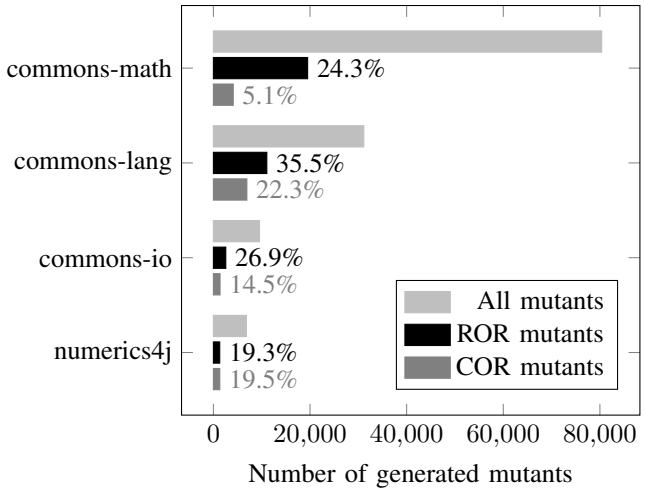

Figure 1. Ratio of the number of COR and ROR mutants to the number of all generated mutants for the investigated case study applications

A. The frequency of the COR and ROR mutants

To answer the first research question, we determined the number of mutants generated by applying the COR and ROR operators with all of the possible replacements defined by Namin et al. [16]. We also ascertained the number of mutants that can be generated by using all of the operators, including COR and ROR. Figure 1 visualizes the ratio of mutants associated with the COR and ROR operators (the dark gray and black bars) to the number of mutants generated by applying all mutation operators (light gray bar). Ranging from 29.4% for commons-math to 57.8% for commons-lang, the number of mutants generated by only applying the COR and ROR operators is a substantial portion of all the induced mutants. With a mean value of 41.8%, this range suggests that there is a notable potential for effectiveness and efficiency improvements through the removal of the redundant mutants associated with COR and ROR.

B. The number of connectors in conditional expressions

As stated in Section III, we can only guarantee that the reduced set of mutants generated by the COR operator is sufficient and redundancy-free for conditional expressions with one logical connector. Therefore, in order to ascertain the benefit of this partial solution, we calculated the ratio of conditional expressions with one connector to the remaining number of conditional expressions. For each case study application, Figure 2 illustrates the distribution of the number of logical connections in the conditional expressions. With a mean value of 78.2% across the four studied programs and a range between 63.3% for numerics4j and 85.9% for commons-math, the number of conditional expressions with only one connector is predominant for all applications. Thus, for almost 80% of the conditional expressions, the suggested subset of replacements, as given in Section III, provides a sufficient and redundancy-free set of mutants.

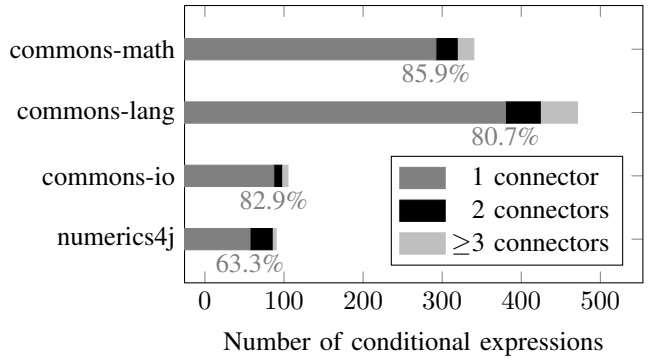

Figure 2. Distribution of the number of logical connectors in conditional expressions for the investigated case study applications

C. Decreasing the runtime of the mutation analysis

The smaller subset of replacements for the COR and ROR operators means that fewer mutants have to be generated and hence the number of necessary executions during the mutation analysis process is also reduced. With regard to the reduction of mutants and the decrease in runtime, we distinguish between generated and covered mutants, with covered meaning that a mutant is reached and executed but not necessarily killed. Hence, the mutation coverage is a necessary but not sufficient condition to kill a mutant. Tables IV and V show the decrease in the number of generated and covered mutants according to the following sets of mutations:

- $\boldsymbol{S_{M1}}$: All mutants generated by all mutation operators with all valid replacements
- $\boldsymbol{S_{M2}}$: Reduced set of mutants generated by all available mutation operators but with sufficient replacements for the COR and ROR operators
- $\boldsymbol{S_{M3}}$: All mutants generated by using the COR and ROR mutation operators with all valid replacements
- $\boldsymbol{S_{M4}}$: Reduced set of mutants generated by only employing the COR and ROR mutation operators but with sufficient replacements

The reduction of the mutations associated with the COR and ROR operators significantly affects the number of generated mutants, even when applying all mutation operators. Depending on the ratio of the COR and ROR mutants to all other mutants, the decrease ranges between 16.9% for commons-math and 32.3% for commons-lang, as shown in Table IV. With respect to the covered mutants, the decrease depicted in Table V is comparable to the decrease of the generated mutants for all applications except commons-io. The is due to the low mutation coverage rate of only 51.7% achieved by the test suite for this application. Many COR and ROR mutants are not covered and hence, a reduction of these mutants does not affect the number of covered mutants.

In addition to calculating the reduction in the number of generated and covered mutants, we also determined the actual improvement in the runtime while exploiting two runtime optimizations. On the one hand, we did not analyze

Table IV
DECREASE IN THE NUMBER OF GENERATED MUTANTS

	S_{M1}	S_{M2}	S_{M3}	S_{M4}
com-math	80,372	66,787 (-16.9%)	23,620	10,035 (-57.5%)
com-lang	31,130	21,074 (-32.3%)	17,998	7,942 (-55.9%)
com-io	9,547	7,319 (-23.3%)	3,954	1,726 (-56.3%)
num4j	6,835	5,437 (-20.5%)	2,647	1,249 (-52.8%)

Table V
DECREASE IN THE NUMBER OF COVERED MUTANTS

	S_{M1}	S_{M2}	S_{M3}	S_{M4}
com-math	72,203	59,195 (-18.0%)	22,620	9,806 (-56.6%)
com-lang	29,069	19,112 (-34.3%)	17,810	7,890 (-55.7%)
com-io	4,935	4,168 (-15.5%)	1,349	558 (-58.6%)
num4j	6,547	5,149 (-21.4%)	2,642	1,225 (-53.6%)

Table VI
DECREASE IN THE RUNTIME OF THE MUTATION ANALYSIS

	S_{M1}	S_{M2}	S_{M3}	S_{M4}
com-math	300.77	271.10 (-09.9%)	51.52	39.27 (-23.8%)
com-lang	28.25	18.70 (-33.8%)	12.63	6.82 (-46.0%)
com-io	6.95	4.58 (-34.1%)	4.15	2.00 (-51.8%)
num4j	2.85	2.08 (-26.9%)	0.92	0.50 (-45.5%)

*Runtimes reported in minutes

mutants that are not covered since they cannot be killed and on the other hand we did not further investigate a mutant once it has been killed. Table VI furnishes the execution time of a mutation analysis process that uses MAJOR to calculate a mutation score for each application's test suite [6], [7]. With a reduction in runtime of up to 34% for commons-io and a minimum of 10% for commons-math, the results demonstrate a significant speed-up for all of the applications. Yet, the observed improvement in the runtime depends on the distribution of the COR and ROR within the application and the runtimes of the tests that do not cover these mutants. For instance, the test suite for commons-math contains a few long-running tests that cover many mutants but only a few COR and ROR mutations. Since the runtime of these tests is a considerable proportion of the total runtime, the removal of the redundant COR and ROR mutants only yields a modest decrease in the cost of mutation analysis for this application.

D. Increasing the precision of the mutation score

Since redundant mutants lead to an imprecision in the mutation score, we used both the complete and reduced set of mutants to calculate this value for all of the case study applications. Table VII gives the mutation score for the generated mutants, with S_{M1}, S_{M2}, S_{M3}, and S_{M4} again denoting the mutation sets described in Section IV-C. Considering only the COR and ROR mutants that are represented by the sets S_{M3} and S_{M4}, the mutation score decreases by 19% on average. When applying all of the

Table VII
DIVERGENCE OF THE MUTATION SCORE WITH REGARD TO THE GENERATED MUTANTS

	S_{M1}	S_{M2}	S_{M3}	S_{M4}
com-math	0.77	0.73 (- 4.5%)	0.75	0.59 (-20.5%)
com-lang	0.76	0.67 (-10.7%)	0.81	0.67 (-17.4%)
com-io	0.41	0.44 (8.3%)	0.26	0.21 (-19.2%)
num4j	0.69	0.65 (- 5.9%)	0.74	0.59 (-19.3%)

Table VIII
DIVERGENCE OF THE MUTATION SCORE WITH REGARD TO THE COVERED MUTANTS

	S_{M1}	S_{M2}	S_{M3}	S_{M4}
com-math	0.85	0.83 (-3.2%)	0.78	0.61 (-22.1%)
com-lang	0.81	0.74 (-8.1%)	0.82	0.67 (-17.7%)
com-io	0.79	0.78 (-1.7%)	0.75	0.64 (-14.7%)
num4j	0.72	0.68 (-4.9%)	0.74	0.61 (-17.8%)

mutation operators, the reduced sets still result in a decrease of up to 10% for programs like commons-lang. Unless the redundant mutants are removed, the mutation score is overestimated for all applications except commons-io. For this application, the corresponding test suite only covers 33% of the generated COR and ROR mutants. Thus, removing the redundant mutants affects the number of generated mutants more significantly than the number of killed mutants, leading to an 8% increase in the mutation score.

Table VIII shows how the removal of the redundant mutants affects the mutation score that is calculated for the number of covered mutants. Once again, there is a notable 18% average decrease in the mutation score when applying only the COR and ROR mutations. For the sets S_{M1} and S_{M2}, the mutation score decreases between 2% for commons-io and 8% for commons-lang. Overall, redundant mutants tend to overestimate the mutation score for the case study applications, thus causing this metric to become a less accurate assessment of test suite quality.

E. Threats to Validity

It is important to examine the threats to the validity of this paper's case study. The chosen subset of sufficient mutation operators could be a threat to internal validity. Different or additional operators may affect both the number and the ratio of the generated mutants. However, the operators employed in the study are frequently used in the literature and therefore provide comparable results [15], [16].

The representativeness of the selected case study applications might be a potential threat to external validity. Thus, the presented results may be different for other programs. The analyzed applications, nevertheless, vary in terms of their size, complexity, and operation purpose. Therefore, we judge that the reported results are meaningful.

Defects in the compiler-integrated mutation tool could be a threat to construct validity. We controlled this threat by employing several small example programs and manually analyzing the resulting mutants and data. Moreover, we used the same tool to conduct two previous empirical studies (i.e., [6], [7]) without encountering any problems. Overall, we judge that the implementation worked correctly.

V. RELATED WORK

As previously mentioned in Section II, employing all available mutation operators with all valid replacements results in a significant number of mutants, especially for large software systems. In addition to the runtime overhead caused by the many mutants, there are also redundant and trivial mutants that may misrepresent the mutation score.

In an attempt to reduce the computational costs, different selective and sampling-based approaches have been proposed in the literature (e.g., [5], [15]). These techniques reduce the quantity of mutants either by decreasing the number of operators or by selecting only a subset of the generated mutants. Yet, all of these approaches view the mutation operators, in their originally defined form, as atomic. Hence, there are still redundancies within the selected subset that affect both the runtime and the mutation score.

Kaminski et al. investigated the ROR operator in detail and showed that a subset of four out of eight valid replacements was sufficient for this operator [8]. They additionally claimed, without further investigation or evidence, that this reduction would improve efficiency. Similar to our focus on conditional and relational operators, Tai developed a theory for testing the predicates in conditional logic statements [17].

In connection with our method that avoids redundant mutants and minimizes the impact of mutations, higher order mutation aims at generating fewer, but more subtle mutants [4]. Mutants created by means of the combination of two first order mutants are referred to as second order mutants. Accordingly, higher order mutation generally denotes the combination of two or more first order mutants. Jia and Harman showed the existence of higher order mutants that are harder to kill than the first order mutants out of which they were created. Nevertheless, the computational costs for higher order mutation are significantly greater because of the combinatorial explosion. However, search-based techniques seem to be an appropriate solution to this problem [4].

Apart from redundant mutants, the equivalent mutant problem is another crucial consideration in mutation testing. Equivalent mutants are harmful to the runtime of the mutation analysis process since they cannot be detected by any test. Additionally, employing a set of mutants that includes equivalent mutants results in an underestimation of the mutation score. Approaches that try to alleviate the equivalent mutant problem can be divided into two categories. On the one hand, there are techniques focusing on the detection of equivalent mutants (e.g., [3], [13]). On the other hand, approaches exist for reducing the number of equivalent mutants during the mutant generation process (e.g., [12]).

VI. Conclusion and Future Work

This paper investigates how redundant mutants affect the effectiveness and efficiency of mutation analysis. Focusing on two well-known mutation operators, namely the relational operator replacement (ROR) and conditional operator replacement (COR), the paper makes several contributions. First, it develops a subsumption hierarchy and reveals a sufficient set of replacements for the COR operator. Using this sufficient set, in conjunction with the ROR operator's reduced set that was shown to be sufficient by Kaminski et al. [8], this paper reports on a case study that empirically demonstrates how redundant mutants affect both the mutation score and the runtime of the mutation analysis process.

After determining how prevalent relational and conditional operators are in real-world applications, this paper examines the ratio of the number of mutants generated by the COR and ROR mutation operators to the total number of mutants. With a mean value of 41.8% and a range from 29.4% to 57.8%, the high percentage of COR and ROR mutants clearly reveals the potential for improving mutation analysis by focusing on the mutants produced by these operators. The experiments also show that employing the sufficient replacements for the COR and ROR operators leads to a commensurate drop in the number of generated mutants that ranges from 16.9 to 32.3%.

Moreover, reducing the number of generated mutants leads to a decrease in the runtime of the mutation analysis process that is between 9.9 and 34.1%. Finally, the empirical results show that, depending on the application, improving the precision of the mutation score can lead to a value that is greater than or less than the score resulting from the use of the original set of mutants. For three applications, using the reduced set of mutants yields a reduction in the mutation score ranging from 4.5 and 10.7%. Yet, when one program's test suite only covers a small percentage of the generated mutants, the mutation score increases by 8.3%.

Because of the promising results of the case study reported on in this paper, we plan as part of our future work to determine a sufficient set of replacements for other mutation operators, such as the arithmetic operator replacement (AOR). Leveraging the theory of Tai [17], the generalization of the results for the COR operator and the establishment of a subsumption hierarchy for conditional expressions with more than one logical connector are other areas for future research. After completing these steps, we intend to replicate the empirical study in this paper to better understand how redundant mutants affect the effectiveness and efficiency of mutation analysis. Our future empirical studies will also incorporate additional case study applications, thus better ensuring that our results are generalizable. Finally, to address the equivalent mutant problem, another critical concern in mutation testing, we will investigate whether mutants with a minimized impact, like the ones described in this paper, are more or less likely to be equivalent.

References

[1] T. A. Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, 1980.

[2] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, 1978.

[3] R. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, 9:233–262, 1999.

[4] Y. Jia and M. Harman. Higher order mutation testing. *Information and Software Technology*, 51:1379–1393, 2009.

[5] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[6] R. Just, G. M. Kapfhammer, and F. Schweiggert. Using conditional mutation to increase the efficiency of mutation analysis. In *Proceedings of the 6th Workshop on Automation of Software Test*, AST '11, pages 50–56, 2011.

[7] R. Just, F. Schweiggert, and G. M. Kapfhammer. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 612–615, 2011.

[8] G. Kaminski, P. Ammann, and J. Offutt. Better predicate testing. In *Proceedings of the 6th Workshop on Automation of Software Test*, AST '11, pages 57–63, 2011.

[9] K. N. King and A. J. Offutt. A Fortran language system for mutation-based software testing. *Software: Practice and Experience*, 21(7):685–718, 1991.

[10] Y.-S. Ma, J. Offutt, and Y.-R. Kwon. MuJava: A mutation system for Java. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 827–830, 2006.

[11] H. Mills. On the Statistical Validation of Computer Programs. Technical report, IBM FSD Report, 1970.

[12] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology*, 1(1):5–20, 1992.

[13] A. J. Offutt and W. M. Craft. Using compiler optimization techniques to detect equivalent mutants. *The Journal of Software Testing, Verification, and Reliability*, 4:131–154, 1994.

[14] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):99–118, 1996.

[15] J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. In *Proceedings of Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, pages 45–55, 2000.

[16] A. Siami Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 351–360, 2008.

[17] K.-C. Tai. Theory of fault-based predicate testing for computer programs. *IEEE Transactions on Software Engineering*, 22(8), 1996.