Inferring Mutant Utility from Program Context

René Just
University of Massachusetts
Amherst, MA, USA
rjust@cs.umass.edu

Bob Kurtz
George Mason University
Fairfax, VA, USA
rkurtz2@gmu.edu

Paul Ammann
George Mason University
Fairfax, VA, USA
pammann@gmu.edu

ABSTRACT

Existing mutation techniques produce vast numbers of equivalent, trivial, and redundant mutants. Selective mutation strategies aim to reduce the inherent redundancy of full mutation analysis to obtain most of its benefit for a fraction of the cost. Unfortunately, recent research has shown that there is no fixed selective mutation strategy that is effective across a broad range of programs; the utility (i.e., usefulness) of a mutant produced by a given mutation operator varies greatly across programs.

This paper hypothesizes that mutant utility, in terms of equivalence, triviality, and dominance, can be predicted by incorporating context information from the program in which the mutant is embedded. Specifically, this paper (1) explains the intuition behind this hypothesis with a motivational example, (2) proposes an approach for modeling program context using a program's abstract syntax tree, and (3) proposes and evaluates a series of program-context models for predicting mutant utility. The results for 129 mutation operators show that program context information greatly increases the ability to predict mutant utility. The results further show that it is important to consider program context for individual mutation operators rather than mutation operator groups.

CCS CONCEPTS

•Software and its engineering →Software testing and debugging;

KEYWORDS

Mutation analysis, program mutation, program context, mutant utility, equivalent mutants, trivial mutants

ACM Reference format:
René Just, Bob Kurtz, and Paul Ammann. 2017. Inferring Mutant Utility from Program Context. In *Proceedings of 26th ACM SIGSOFT International Symposium on Software Testing and Analysis , Santa Barbara, CA, USA, July 2017 (ISSTA'17),* 11 pages.
DOI: 10.1145/3092703.3092732

1 INTRODUCTION

Consider an engineer attempting to evaluate her test suite with mutation analysis [7]. A full-blown mutation analysis system produces far more mutants than necessary or useful. Until recently, the advice to the engineer would have been to choose a *do fewer* approach in which only a carefully chosen subset of the mutants was generated. Unfortunately, recent research has identified fundamental weaknesses in existing selective mutation approaches: all of them are very likely to miss effective mutants on some programs [24], and none of them is likely to greatly outperform random mutant selection, a strategy that simply chooses a random subsample of the generated mutants [9, 11]. This same research has shown that redundancy in mutants can be precisely characterized with *dominator sets*. In an empirical study, Ammann et al. showed that the dominator sets averaged just 1.2% of the non-equivalent mutants—that is, nearly 99% of the non-equivalent mutants were redundant [4]. In other words, a small set of dominator mutants captures the power of the full set of mutants generated by a typical mutation system.

The goal of selective mutation—finding a small set of mutants that retain the utility of the original set—is legitimate. The bad news is that finding dominator sets directly is undecidable and we do not know how to choose a useful proxy while avoiding mutants that are equivalent, trivial, or redundant. This paper is a first step in addressing this problem.

1.1 Rationale of Our Approach

Why do existing selective mutation approaches fail? We conjecture that a root problem, and perhaps *the* root problem, is that existing approaches to selective mutation take no account of program context. For example, existing mutation approaches treat mutating a relational operator in a `for` loop test the same as mutating a relational operator in an `if` statement. But many mutations of relational operators in `for` loop tests are killed by every test case, and hence useless. Others are equivalent, and hence worse than useless. As another example, mutating an arithmetic operator in the context of array indices is less likely to result in a dominator mutant than mutating an arithmetic operator in other contexts. As a third example, appending a post-increment operator (e.g., ++) to a variable is less likely to generate a non-equivalent mutant if it appears late in a computation, for the simple reason that there is less likely to be a data flow from the variable to the output.

Just et al. [20] showed that a strong coupling exists between mutants and, through the test cases that detect them, 73% of real faults. Allamanis et al. [3] proposed additional mutation operators that further increase the ratio of coupled real faults. The increased real-fault coupling, however, comes at the cost of significantly more mutants, most of which are redundant—reinforcing the need for effective mutant selection. Hence, if the goal is to generate a small set of mutants that are useful and highly coupled to likely real faults, then context-based program mutation is the way to go. In other words, to avoid equivalent, trivial, and redundant mutants, program context must be taken into account.

ISSTA'17, Santa Barbara, CA, USA

DOI: 10.1145/3092703.3092732

1.2 Contributions

The hypothesis of this paper is that the selection of a set of effective mutants must take into account the program context in which the mutants are generated. The relationship between mutant utility and program context may be quite complex, and hence we do not propose to derive it theoretically. Rather, this paper shows that program context information is useful for predicting mutant utility.

The specific contributions of this paper are:

- Insight into why program context is critical for assessing mutant utility.
- A model of mutant utility in terms of equivalence, triviality, and dominance.
- A model of program context based on neighboring nodes in the abstract syntax tree.
- An empirical study that shows that program context is a strong predictor for mutant utility.

2 BACKGROUND ON MUTATION ANALYSIS

Mutation testing [7] is a test criterion that generates a set of program variants, called *mutants*, and then challenges the tester to design tests that detect these mutants. A test that can distinguish between a mutant and the original program is said to detect, or *kill*, that mutant. In *strong* mutation testing, killing a mutant means that the mutant and the original program generate different outputs. In *weak* mutation testing, killing a mutant means that the internal program state of the mutant differs from the internal program state of the original program at some point during execution.

A mutant is generated by a *mutation operator*, which is a program transformation rule that generates a program variant of a given program based on the occurrence of a particular syntactic element. One example of a mutation operator is the replacement of an instance of the arithmetic operator `+` with `-`. Specifically, if a program contains an expression `a + b`, this mutation operator creates a mutant where `a - b` replaces this expression. The *mutation* is the syntactic change that a mutation operator introduces. This paper considers *first-order mutants*, which means that each program variant contains exactly one mutation.

A mutation operator is applied everywhere it is possible to do so. In the example above, if the arithmetic operator `+` occurs multiple times in a program, the mutation operator will create a separate mutant for each occurrence. A *mutation operator group* is a group of related mutation operators. For example, the AOR mutation operator group, which includes the mutation operator above, is the group of all mutation operators that replace an arithmetic operator. Similarly, the ROR mutation operator group is the group of all mutation operators that replace a relational operator.

2.1 Equivalent Mutants

A mutant may behave exactly as the original program on all inputs. Such a mutant is called an *equivalent mutant* and cannot be killed. As an example of an equivalent mutant, consider the following comparison of two integers, which returns the smaller value of the two: `return (a < b) ? a : b`. Replacing the relational operator `<` with `<=` (`return (a <= b) ? a : b`) results in an equivalent mutant—if `a` and `b` are equal, returning either value is correct, and hence both implementations are semantically equivalent.

Given a set of mutants, M, a test set T is *mutation-adequate* with respect to M iff for every non-equivalent mutant m in M, there is some test t in T such that t kills m.

2.2 Trivial Mutants

Mutants vary widely in how difficult it is to find a test case that kills the mutant. A *trivial mutant* is one that is killed due to an exception by every test case that covers and executes the mutated code location. As an example, consider a `for` loop with a boundary check for an array index (`for (int i=0; i<numbers.length; ++i){...}`). If the index variable `i` is used to access the array `numbers` then a mutation `i<=numbers.length` is trivial, as any test that reaches the loop will terminate with an `IndexOutOfBoundsException`—the last value for `i` is guaranteed to index `numbers` out of bounds.

2.3 Dominator Mutants

Mutation operators generate far more mutants than are necessary. This redundancy was formally captured in the notion of *minimal mutation* [4]. Give any set of mutants, M, a *dominator set* of mutants, D, is a minimal subset of M such that any test set that is mutation-adequate for D is also mutation-adequate for M.

Computing a dominator set is an undecidable problem, but it is possible to approximate it with respect to a test set [23]—the more comprehensive the test set, the better the approximation. This approximation is not useful for the practicing engineer, who needs to know the set of dominator mutants a-priori to develop a test set. However, from a research and evaluation perspective, a dominator set provides a precise way for identifying redundancy in a set of mutants, and hence the dynamic approximation approach is an important research tool for analyzing mutation testing techniques.

Given a finite set of mutants M and a finite set of tests T, mutant m_i is said to *dynamically subsume* mutant m_j if some test in T kills m_i and every test in T that kills m_i also kills m_j. If two mutants m_i and m_j in M are killed by exactly the same tests in T, we say that m_i and m_j are *indistinguished*.

We capture the subsumption relationship among mutants with the *Dynamic Mutant Subsumption Graph* or DMSG [23]. Each node in a DMSG represents a maximal set of indistinguished mutants and each edge represents the dynamic subsumption relationship between two sets of mutants. More specifically, if m_i dynamically subsumes m_j, then there is an edge from the node containing m_i to the node containing m_j. Further, if m_i dynamically subsumes m_j but the converse is not true, we say that the subsumption is *strict*. If a test kills any arbitrary mutant in the DMSG, it is guaranteed to kill all the subsumed mutants [4], i.e., all mutants below it in the graph.

Table 1 shows an example kill matrix that indicates which test kills which mutants. In this example, the set M consists of 14 mutants and the set T consists of 4 tests. Every test that kills m_{12} also kills m_3, m_6, and m_{11}. Hence, m_{12} dynamically subsumes these mutants. In the case of the first two mutants, the dynamic subsumption is strict. However, m_{11} and m_{12} are killed by exactly the same tests, so the subsumption is not strict; these mutants are indistinguished.

We use the subsumption relationships to construct the DMSG shown in Figure 1. Mutants m_1 and m_{10} are not killed by any of the tests in T—shown in the unconnected node with a dashed border.

Table 1: Example kill matrix.
A check mark indicates that a test t_i kills a mutant m_j. Context refers to the lexically enclosing statement of the relational operators in Figure 3.

	Mutant			Test			
	Group	Operator	Context	t_1	t_2	t_3	t_4
m_1:	ROR	< ⟼ !=	for				
m_2:	ROR	< ⟼ ==	for		✓		✓
m_3:	ROR	< ⟼ <=	for	✓	✓	✓	✓
m_4:	ROR	< ⟼ >	for		✓		
m_5:	ROR	< ⟼ >=	for		✓		✓
m_6:	ROR	< ⟼ true	for	✓	✓	✓	✓
m_7:	ROR	< ⟼ false	for		✓		
m_8:	ROR	< ⟼ !=	if	✓			
m_9:	ROR	< ⟼ ==	if		✓		
m_{10}:	ROR	< ⟼ <=	if				
m_{11}:	ROR	< ⟼ >	if	✓	✓		
m_{12}:	ROR	< ⟼ >=	if	✓	✓		
m_{13}:	ROR	< ⟼ true	if	✓			
m_{14}:	ROR	< ⟼ false	if		✓		

These mutants are equivalent with respect to T but they may be killable by a test that is not an element of T. The DMSG is based on a finite test set, so it can only make claims about test equivalence.

Dominator mutants are those not strictly subsumed by any other mutant and are shown in the graph in dominator nodes with double borders. Figure 1 has two dominator nodes and any combination of one mutant from each dominator node forms a *dominator mutant set*. Hence, Figure 1 has $4 * 2 = 8$ distinct dominator mutant sets; $\{m_4, m_8\}$ is an example. Because each dominator set contains one mutant from each dominator node, all dominator sets are equally useful and a dominator set can be selected arbitrarily from all possible sets. Consequently, only two of the 14 mutants matter—if a test set kills the mutants in a dominator mutant set, it is guaranteed to kill all non-equivalent mutants, which are redundant.

For completeness, Figure 2 shows the static subsumption relation for the mutants of the relational operator <, analyzed in isolation. The top row in Figure 2 shows the three dominator mutants—the mutants that replace < with `false`, <=, and != subsume all the others. Note that this dominance relation only holds for weak mutation testing [25], and that it assumes that none of the dominators is equivalent. Section 4 revisits the examples in Figure 1 and 2, and shows that one of the mutation operators (< ⟼ !=) indeed produces an equivalent mutant in the context of an example program.

3 MUTANT UTILITY

Informally, dominator mutants and mutants high in the mutant subsumption graph have high utility, and equivalent and trivial mutants have low utility. This section makes the notion of mutant utility precise along each of three dimensions of equivalence, triviality, and dominance. Section 5.8 shows how these three dimensions might be combined into an overall notion of mutant utility.

3.1 Equivalence

Ideally, a mutation system would not generate any equivalent mutants, but since mutant equivalence, or program equivalence, is

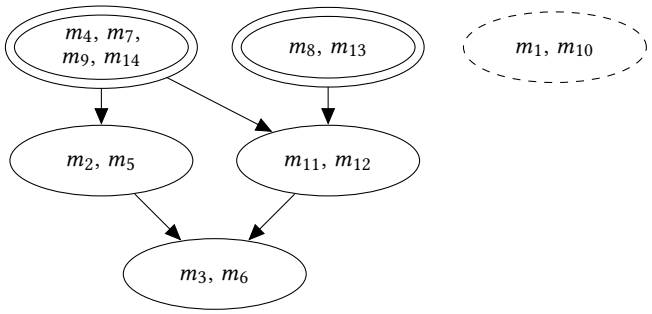


Figure 1: Dynamic mutant subsumption graph (DMSG) for the kill matrix shown in Table 1.

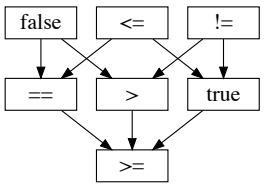


Figure 2: Static subsumption for the relational operator < [22].

an undecidable problem, this goal cannot be achieved in general. Instead, our approach is to rank mutants by estimating how likely they are to be equivalent. Specifically, mutant utility with respect to equivalence is an estimate of the likelihood that the mutant is equivalent. Program context is useful if it enables us to refine our estimate of equivalence for a given mutant. The farther the refined estimate is from the base (in either direction) and the closer it is to the ground truth, the more useful the context.

Formally, equivalence utility for a mutant m with respect to a mutation operator group G is the likelihood that an arbitrary mutant produced by some Op in G is equivalent. Similarly, equivalence utility for a mutant m with respect to a mutation operator Op is the likelihood that an arbitrary mutant produced by Op is equivalent. Finally, equivalence utility for a mutant m with respect to a mutant operator Op and context c is the likelihood that an arbitrary mutant produced by Op in context c is equivalent.

3.2 Triviality

Since mutation analysis requires more human and compute time than coverage criteria such as branch coverage, mutants that are always killed by branch-adequate test suites are not of practical value. One of our goals is to identify non-trivial mutants. Again, program context is useful if it enables us to refine our estimate of triviality for a given mutant.

The utility definitions for triviality are an exact parallel of the utility definitions for equivalence.

3.3 Dominance

Mutants that are not dominators, but are "close" to being dominators, are still valuable. For this paper, we need a metric that captures this observation. To this end, we propose the *dominator strength* metric that satisfies three important properties: it is monotonically increasing along any path in the DMSG, it is fairly evenly distributed between 0 and 1, and it is insensitive to redundant mutants.

```
1 /*
2  * Compute the minimum value of a non-null,
3  * non-empty array of integers.
4  */
5 public int getMin(int[] numbers) {
6   int min = numbers[0];
7   for (int i=1; i < numbers.length; ++i) {
8     if (numbers[i] < min) {
9       min = numbers[i];
10    }
11  }
12  return min;
13 }
```

Figure 3: Relational operator in two different program contexts.

We define the dominator strength $s_D(M)$ for any mutant M as the number of nodes in the graph that are subsumed by M, divided by the number of nodes in the graph subsumed by M plus the number of nodes in the graph that subsume M:

$$s_D(M) = \frac{\#nodes\ M\ subsumes}{\#nodes\ M\ subsumes + \#nodes\ that\ subsume\ M}$$

$s_D = 1$ identifies a dominator mutant and $s_D = 0$ identifies a mutant that does not strictly subsume any other mutants. As an example, in Figure 1, $s_D(m_{12}) = 1/(1+2) = 0.33$.

The utility definitions for dominance are an exact parallel of the utility definitions for equivalence. Note that in contrast to equivalence and triviality, higher dominance is better.

4 PROGRAM CONTEXT

This section first informally describes the notion of program context using a motivational example (Section 4.1) and then details our proposed approach to modeling program context (Section 4.2).

4.1 Motivational Example

Consider the program listing in Figure 3. The following seven mutation operators are applicable to each of the highlighted program locations (lines 7 and 8), where *lhs* and *rhs* are meta-variables:

Op_1: *lhs* < *rhs* ⟼ *lhs* != *rhs*
Op_2: *lhs* < *rhs* ⟼ *lhs* == *rhs*
Op_3: *lhs* < *rhs* ⟼ *lhs* <= *rhs*
Op_4: *lhs* < *rhs* ⟼ *lhs* > *rhs*
Op_5: *lhs* < *rhs* ⟼ *lhs* >= *rhs*
Op_6: *lhs* < *rhs* ⟼ `true`
Op_7: *lhs* < *rhs* ⟼ `false`

The utility of the mutants that these mutation operators generate depends on the program context, as Table 2 shows.

Table 2 corroborates that any approach that universally selects and applies a subset of mutation operators—even within a single program—is doomed to failure. For example, the mutation operator Op_1 generates an equivalent mutant in line 7 but a dominator mutant in line 8. This means that the inclusion of Op_1 is crucial but at the same time that this operator should never be applied in a context similar to the one in line 7. This example motivates our goal of capturing the notion of program context more precisely with the ultimate goal of learning what mutation operator is most likely to generate an equivalent, trivial, or dominator mutant in what program context. Figure 1 (dynamic subsumption) and Figure 2 (static subsumption) illustrate exactly this same point: the subsumption relations change when the mutations are considered in context.

Table 2: Mutant utility depending on program context.
For each of the seven mutation operators, applied to each of the highlighted program locations in Figure 3, is the generated mutant a dominator (*dom.*), subsumed (*sub.*), trivial (*triv.*), or equivalent (*equi.*) mutant?

Location	Op_1	Op_2	Op_3	Op_4	Op_5	Op_6	Op_7
Line 7	**equi.**	sub.	triv.	**dom.**	sub.	**triv.**	**dom.**
Line 8	**dom.**	sub.	equi.	**sub.**	sub.	**dom.**	**sub.**

An immediate follow-up question to this motivational example is whether this occurs in practice. To this end, we conducted an exploratory study, using the Lang-1 subject from the Defects4J benchmark. The test suite of Lang-1 achieves 98.2% statement coverage and kills 216 out of 508 ROR mutants that are generated by applying Op_1. The remaining 292 mutants are test-equivalent and, given the strength of the test suite, a large fraction of them is very likely to be equivalent. Hence, absent context information, the estimate that an Op_1 mutant is equivalent is 292/508 or 57%.

Incorporating context information and considering only mutations generated by Op_1 in the condition of a `for` loop, reduces the number of mutants from 508 to 165. Of these, 11 mutants are killed and 154 mutants are equivalent. Therefore, considering just the enclosing statement changes the estimate that an Op_1 mutation in a `for` loop context is equivalent from 57% to 93%. This enclosing statement context, all by itself, already provides a very strong signal for the equivalence of mutants generated by Op_1. Subsequent manual analysis revealed that for each of the 11 killed Op_1 mutants, there is additional context information that could be exploited to predict whether or not that mutant is equivalent. For example, some of these `for` loops start at a variable index instead of 0 or 1.

4.2 Modeling Program Context

Rather than predefining a very small set of exclusion patterns that provably generate equivalent or trivial mutants, we generalize this notion of patterns to program context. In particular, we model program context using the program's abstract syntax tree (AST). In contrast to the purely syntactic level, the AST provides a higher level of abstraction and semantic information. This allows us to abstract over potentially irrelevant details, such as identifier names, and to exploit information about data types and scopes. Figure 4 shows the partial AST for the `getMin` method from Figure 3.

Our ultimate goal is to apply machine learning on labeled ASTs to train a classifier that can predict mutant utility, given a mutation operator and a program's AST. As a first step, this paper explores different dimensions of program context to guide future research on developing more complex program context models.

4.2.1 Parent Context. Our approach traverses the AST and computes the sequence of AST nodes, from the target node to the root node. Such a traversal can generate AST node sequences at different levels of abstraction, as shown in Figure 5. The sequence of AST nodes allows us to predict mutant utility using only the parent statement node, the entire sequence, or n-grams of that sequence. In Figure 4, the parent statement of the target node is an `if` statement.

As an example, consider the expression *lhs* < *rhs* and the mutation operator Op_1 from the motivational example, which changes the relational operator from < to `!=`. This mutation is much more likely to be equivalent in a `for` loop than in an `if` statement context.

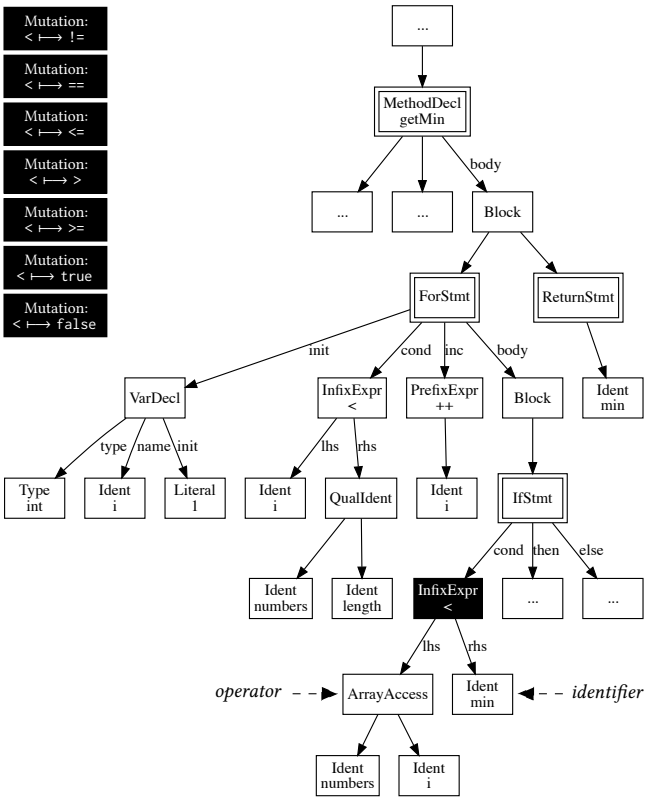


Figure 4: Partial AST for the `getMin` method (Figure 3) and seven possible mutations for the highlighted target AST node.

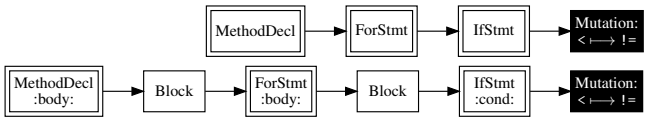


Figure 5: Two parent node sequences for the mutation operator Op_1 applied to the highlighted target AST node in Figure 4.

The two sequences represent the same path in the AST at different levels of abstraction: the first sequence includes only nodes that represent a top-level statement, whereas the second includes all nodes plus edge labels.

4.2.2 Children Context. Our approach considers the node types of the target AST node's children. In particular, it determines whether any of the child nodes represents a *literal* value, an *identifier*, or an *operator*. In Figure 4, the target node has two child nodes—an operator node (`ArrayAccess`) and an identifier node (`min`).

As an example, consider the statement `if (`*lhs* `&` *rhs*`)`, where *lhs* and *rhs* are expressions, and a mutation operator that changes the operator `&` to the short-circuit operator `&&`. If *rhs* is a literal value or an identifier, then the mutant is equivalent. However, if *rhs* is a side-effecting operator or method call, then the mutant may not be equivalent—if *lhs* evaluates to false, the missing side effect of the mutant may lead to its detection.

4.2.3 Data Type Context. Our approach considers, in addition to the structural parent and children contexts, the data type context of the target AST node. Depending on the target node, this includes the data types of operands or formal parameters and the return type. In Figure 4, the data type of the target node is `(int,int)boolean`.

As an example, consider the expression *lhs* < *rhs* and a mutation operator that changes the relational operator from < to <=. Knowing whether this comparison is integer vs. floating point leads to different expectations about mutant utility: mutating a floating point comparison is highly unlikely to lead to an `ArrayIndexOutOfBoundsException`—neither *lhs* nor *rhs* is likely to be used as an array index. On the other hand, the same mutation on integers is likely to lead to this exception if *lhs* or *rhs* is used as an array index. Moreover, differentiating < from <= is straight forward for integers but quite complex for floating point numbers. Hence, such a mutation is more likely to be equivalent for floating point numbers than for integers.

4.2.4 Other Types of Context. The AST provides additional program context information, not studied in this paper. This includes, amongst others, information about the scope and visibility of a variable, or the program's control and data flow.

As an example, consider mutating an assignment to a global vs. local variable. The persistence of a global variable means that mutations to its value are far less likely to be equivalent.

5 EVALUATION

The goal of this empirical evaluation is to study whether program context affects mutant utility and what dimensions of program context are strong predictors for what dimension of mutant utility.

5.1 Subjects

We employed the Defects4J benchmark [19] (v1.0.1), which provides a set of 357 subjects, each accompanied by a thorough, developer-written test suite. For each subject, Defects4J provides a buggy and a fixed program version. The difference between the two versions is a set of classes that a developer fixed—the set of *modified classes*.

We selected 163 subjects from the Defects4J benchmark where the test suite achieved at least 95% statement coverage on all modified classes. Since this study approximates mutant utility using test execution information (see Section 5.3), we selected these subjects because of their thorough test suites.

We employed the Major mutation framework [16] (v.1.3.2) to generate mutants for the modified classes for these subjects, perform a mutation analysis using the subjects' test suites, and compute the kill matrix (i.e., execute all tests against all mutants) for each subject. Major could not generate the kill matrix for 56 out of 163 subjects due to a computational timeout of 48 CPU hours. Note that the expected runtime to compute the kill matrix for some of the Defects4J subjects is beyond 100 CPU days [33].

An automated step then filtered the remaining 107 subjects to remove subjects containing duplicate classes (and thus removed the possibility of duplicate mutants) using the following procedure:

(1) When two or more subjects contain the same classes, retain only the subject with the highest statement coverage.
(2) When two or more subjects contain the same classes and have the same statement coverage, retain only the subject with the largest number of test cases.
(3) When two or more subjects contain the same classes, have the same statement coverage, and have the same number of test cases, retain only the subject with the newest version of the subject source code.

Table 3: Summary of investigated mutation operator groups and mutants.

Dominator strength gives the average dominator strength for all killed mutants, *Covering tests* gives the average number of tests that cover each covered mutant, and *Killing tests* gives the average number of tests that kill each covered mutant. The highlighted rows indicate discarded mutation operator groups, which do not contribute enough mutants for the empirical study and which are not included in the *Retained* row.

Group	Total mutants	Covered mutants		Killed mutants		Dominator mutants		Trivial mutants		Dominator stength	Covering tests	Killing tests
AOR	14868	14804	(99.6%)	13483	(91.1%)	4099	(27.7%)	1760	(11.9%)	0.666	44.1	20.1
COR	12097	12043	(99.6%)	9665	(80.3%)	4035	(33.5%)	1246	(10.3%)	0.701	76.7	26.3
EVR	4321	4214	(97.5%)	4024	(95.5%)	1201	(28.5%)	1241	(29.4%)	0.573	76.0	44.0
LOR	404	362	(89.6%)	311	(85.9%)	120	(33.1%)	9	(2.5%)	0.691	275.6	21.9
LVR	14321	13992	(97.7%)	11298	(80.7%)	4979	(35.6%)	2881	(20.6%)	0.695	105.3	16.2
ORU	601	591	(98.3%)	538	(91.0%)	172	(29.1%)	18	(3.0%)	0.675	442.2	20.3
ROR	27900	27668	(99.2%)	22822	(82.5%)	9977	(36.1%)	3874	(14.0%)	0.676	46.1	18.8
SOR	196	190	(96.9%)	170	(89.5%)	69	(36.3%)	1	(0.5%)	0.785	520.7	7.2
STD	7065	6858	(97.1%)	5840	(85.2%)	2568	(37.4%)	1025	(14.9%)	0.742	62.2	22.6
Retained	80572	79579	(98.8%)	67132	(84.4%)	26859	(33.8%)	12027	(15.1%)	0.680	63.8	25.7

This filtering process resulted in 98[1] subjects for which Major generated a total of 81,773 mutants. Table 3 provides a breakdown of these generated mutants across mutation operator groups. In contrast to the other mutation operator groups, LOR, ORU, and SOR yielded very few mutants. To avoid spurious results due to insufficient sample sizes, we discarded the 1,201 mutants from these three mutation operator groups (1.5% of all mutants). The mutants from these mutation operator groups, shown in gray in Table 3, are not included in the "Retained" row of the table. Of the 80,572 retained mutants, 993 were not covered by any test and from the evaluation, leaving 79,579 mutants for analysis. These remaining mutants were covered by an average of 63.8 tests per mutant, with 67,132 mutants (84.4%) killed by an average of 25.7 tests per mutant.

In addition to the breakdown of killed mutants, dominator mutants, and trivial mutants, Table 3 gives, for each mutation operator group, the average dominator strength and the average number of tests that cover and kill a mutant of that group.

5.2 Mutation Operator Groups and Operators

This study considers the following six mutation operator groups:

(1) AOR: Arithmetic operator replacement
(2) COR: Conditional operator replacement
(3) EVR: Expression value replacement
(4) LVR: Literal value replacement
(5) ROR: Relational operator replacement
(6) STD: Statement deletion

In addition to the mutation operator groups, this study analyzes the 129 individual mutation operators to determine the effect of program context on each mutation operator.

5.3 Mutant Utility

This study explores three dimensions of mutant utility (Section 3):

(1) Equivalence
(2) Triviality
(3) Dominance

[1]Chart-24, Closure-{1, 5, 8, 9, 12, 13, 14, 15, 28, 36, 46, 49, 55, 58, 67, 72, 88, 89, 91, 92, 98, 102, 103, 108, 111, 116, 124, 130, 132}, Lang-{1, 2, 4, 11, 14, 21, 22, 25, 28, 31, 33, 37, 40, 44, 45, 49, 51, 53, 54, 55, 58, 59, 62}, Math-{1, 2, 4, 5, 7, 10, 15, 21, 22, 25, 26, 28, 33, 35, 39, 40, 42, 44, 51, 52, 57, 68, 69, 71, 72, 76, 79, 82, 84, 86, 89, 91, 95, 96, 100, 103, 105}, Time-{3, 4, 5, 7, 10, 17, 21, 22}

Since ground truth about each of the three dimensions of mutant utility is, in general, undecidable, we resort to approximations in this paper, using the thorough test suites that accompany the study subjects. Specifically, we use unkilled mutants as a proxy for equivalent mutants, exceptional behavior as a proxy for trivial mutants, and the DMSGs computed from the test suites as a proxy for the true mutant subsumption relationships. Section 5.9 discusses threats to validity and the implications of these approximations.

5.4 Program Context

Recall that our overall goal is to identify what dimensions of program context are most likely to predict whether a mutation operator generates an equivalent, trivial, or dominator mutant. This study investigates the following three dimensions of program context:

(1) *Parent statement context*:
The type of the nearest ancestor AST node that corresponds to a top-level statement, annotated (where applicable) with the relationship between the mutated node and that parent statement node. Figures 4 and 5 give an example for such an annotated relationship, where the mutated node is the child node of an `if` statement condition (`IfStmt:cond:`).

(2) *Children context*:
- *Has literal child*: indicates whether any immediate child node of the mutated AST node is a literal.
- *Has identifier child*: indicates whether any immediate child node of the mutated AST node is an identifier.
- *Has operator child*: indicates whether any immediate child node of the mutated AST node is an operator.

It is possible for a mutated AST node that none or more than one of the above applies. Figure 4 gives an example, where the mutated AST node has both, an operator (`ArrayAccess`) and an identifier (`min`) child node.

(3) *Data type context*:
The data type of the mutated AST node. Figure 4 gives an example for a relational operator, where the data type includes the types of the operands and the return type (`(int,int)boolean`).

5.5 Results

This section discusses the results for parent statement context, which showed the strongest signal for mutant utility in isolation. Section 5.7 quantifies the results for each dimension of program context in isolation and in combination with the other two.

Each of Figures 6a, 6b, and 6c displays the expected mutant utility at three levels of granularity. The first level is the mutation operator group level. For example, in each figure, the first row in the first column shows the expected mutant utility for all AOR mutants. The second level is the mutation operator level. For example, in each figure, the second row in the first column shows the expected mutant utility for all mutants generated by each of the AOR mutation operators. The third level puts each mutation operator into context—that is, it associates each mutation operator with parent statement context. For example, in each figure, the third row in the first column shows the expected mutant utility for all mutants generated by each of the AOR mutation operators in a particular parent statement context. Note that the third row in each figure only shows a mutation operator in a parent statement context if that operator yielded at least ten mutants in that context; this excludes 1.7% of all mutants.

Since very low and very high values are of most interest, the x-axis for each graph in each column shows data sorted in non-descending order. Informally, "flat" graphs are "bad", in that they do not convey much predictive power as to mutant utility, but graphs with "low" and/or "high" values are "good", in that these regions identify mutants that are either desirable or should be avoided.

5.5.1 Equivalence. Figure 6a shows the impact of the program context on the expected mutant utility considering only equivalence. The top row of the figure shows that there is not much variance between the six mutation operator groups that we studied. In particular, trying to predict likely equivalent mutants from mutation operator groups is hopeless. The second row in the figure shows that there is some variance between mutation operators, both within and between mutation operator groups. For example, while the expected mutant utility shows almost no variation for AOR and STD mutation operators, some COR and ROR mutation operators are much more likely to generate equivalent mutants than others. Adding parent statement context information, as is shown in the last row of the figure further increases this variance. In particular, the context information allows one to identify some COR, EVR and ROR mutation operators—the ones at the right hand edge of the graph—that are highly likely to generate equivalent mutants in a particular context. Conversely, the left edge of each graph shows mutation operators in contexts where they are much less likely to generate equivalent mutants.

Figure 6a shows a strong signal that certain mutation operators are very likely to generate equivalent mutants in certain parent statement contexts, and also that others are very unlikely to generate equivalent mutants in other contexts. In terms of our motivational example, the combination of the Op_1 ROR mutation operator, which replaces the relational operator < with != and the `for` loop parent statement context appears near the right hand side of the third row, fifth column graph, precisely because this combination of mutation operator and parent statement context is highly likely to generate an equivalent mutant. Hence, a context-aware mutation system should not apply this mutation operator in that program context.

5.5.2 Triviality. Figure 6b shows the impact of the program context on the expected mutant utility considering only triviality. The top row of Figure 6b shows that for all mutation operator groups, roughly 20% of the generated mutants are trivial. The top row, however, does not give any guidance as to which of these mutation operator groups are more likely to generate trivial mutants than others. Breaking the analysis down by mutation operator significantly improves matters. For example, the third graph in the second row shows that some EVR operators are highly unlikely to generate a trivial mutant, while other EVR operators are highly likely to do so. Adding parent statement context information, as shown in the third row improves the prediction of triviality even more.

5.5.3 Dominance. Figure 6c shows the impact of the program context on the expected mutant utility considering only dominance. Again, knowing just the mutation operator group is not enough to make a meaningful prediction of dominator strength. Adding in the specific mutation operator adds some predictive power, but not nearly as much as also including the parent statement context, as shown in the last row. However, the results for dominance show less differentiation compared to equivalence and triviality.

5.6 Comparison with Random Selection

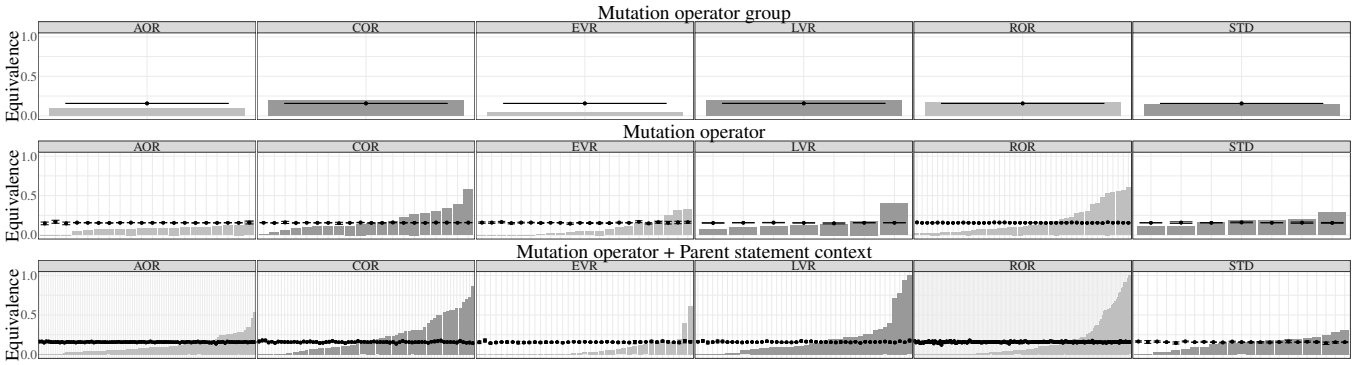
To ensure that our results were not simply an artifact of grouping smaller numbers of mutants together, we performed a control experiment using random mutant selection, duplicating our context-based selection process as closely as possible. For each bar in Figures 6a–6c, which represents the expected mutant utility for a set of grouped mutants, we substituted an identical number of mutants randomly chosen from the entire set of mutants. We repeated this randomized process 100 times. The error bars in Figures 6a–6c show the results in terms of mean expected mutant utility and 90% confidence interval. The random selection results show very little differentiation in expected mutant utility compared to our new selection process.

Additionally, we computed the 90% confidence interval for the expected mutant utility for each mutation operator in a given parent statement context using bootstrapping [8] with 1000 iterations; Figure 7 shows the results. While the confidence intervals show some variance for smaller groups of mutants, program context still provides a clear signal for mutant utility. Overall, we conclude that the differentiation of results is indeed due to program context and not a sampling artifact.

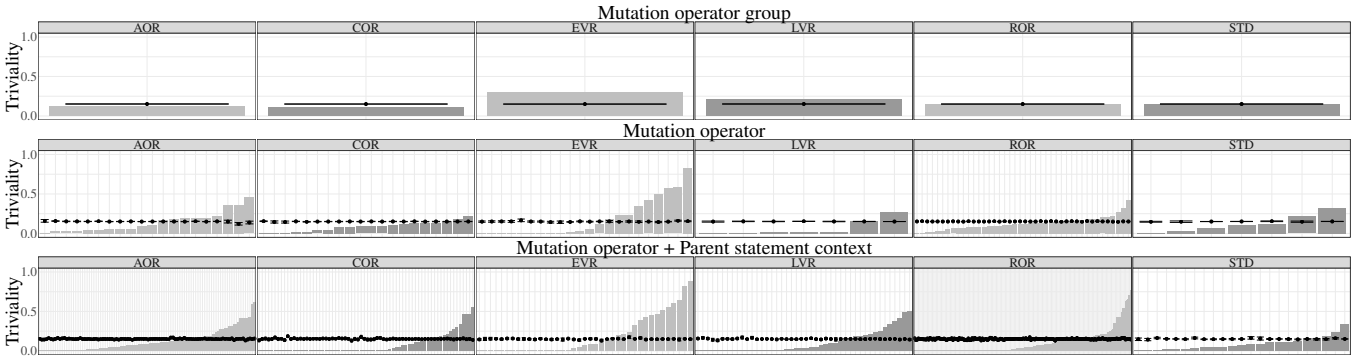
5.7 Predicting Mutant Utility

We assessed the predictive power of program context in a cross validation experiment, studying which dimension of program context best predicts mutant utility in isolation and whether the combination of multiple dimensions improves over the strongest predictor.

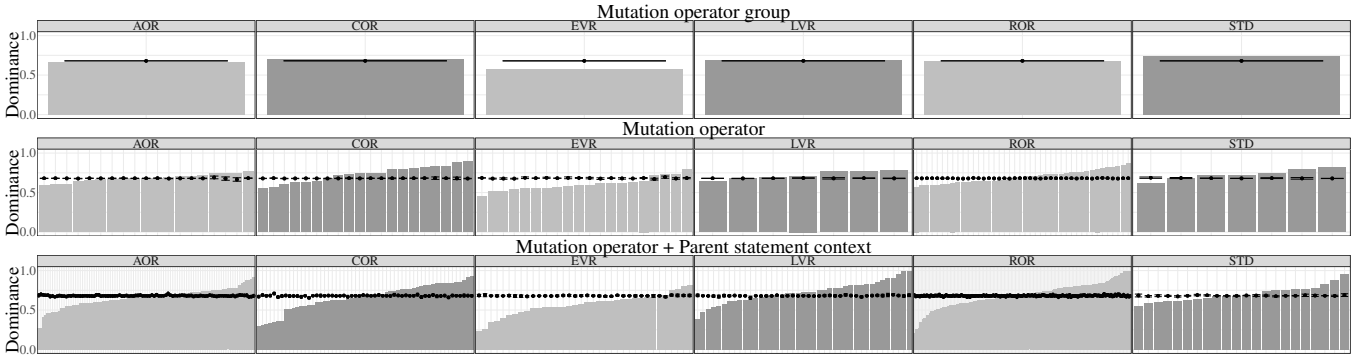
Specifically, we performed repeated random subsampling (100 runs), randomly splitting the set of all mutants into a training set (80% of the mutants) and a test set (the remaining 20% of the mutants). For each dimension of mutant utility (equivalence, triviality, and dominance), we computed the expected mutant utility for each mutation operator group and individual mutation operator, with and without incorporating program context information, on the training set. Using the expected mutant utility as a predictor, we



(a) Equivalence



(b) Triviality



(c) Dominance

Figure 6: Expected mutant utility for a) equivalence, b) triviality, and c) dominance, with and without considering program context. For each of equivalence, triviality, and dominance, each gray bar gives the expected mutant utility for a set of mutants grouped together by mutation operator group (top row), mutation operator (middle row), or mutation operator in a given parent statement context (bottom row). The corresponding error bar shows the expected mutant utility for a set of randomly selected mutants that has equal cardinality.

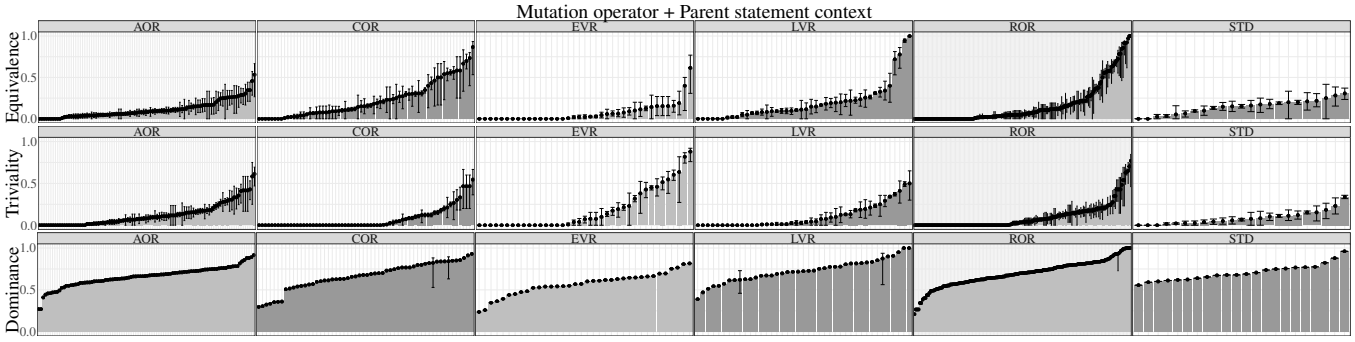


Figure 7: Bootstrap confidence intervals for the expected mutant utility for each mutation operator in a given parent statement context.

Table 4: Cross-validation results: Improved prediction of mutant utility when incorporating program context.

Improvements shown as the reduction of the cross-validation test error. A higher reduction is better, 0% indicates a prediction that is no better than random, and 100% indicates a perfect prediction. *Grp* denotes mutation operator group, *Op* denotes mutation operator, *PC* denotes parent statement context, *CC* denotes children context, and *TC* denotes data type context.

Utility	Grp	Grp +					Grp + Op +			
		Op	PC	CC	TC	PC+CC+TC	PC	CC	TC	PC+CC+TC
Equi.	2%	13%	5%	2%	3%	6%	20%	15%	14%	23%
Triv.	2%	9%	5%	3%	8%	10%	14%	12%	13%	19%
Dom.	1%	3%	3%	1%	1%	3%	6%	4%	4%	8%

then computed the prediction error for each mutant in the test set. For approximately 1.5% of all tested mutants across all 100 runs, all mutants associated with a particular mutation operator and program context ended up in the test set, precluding the computation of the expected mutant utility for those mutants on the training set. In these cases, we resorted to the expected mutant utility of the mutation operator that generated the mutants. Finally, we computed the mean squared error over all runs and all mutants.

Table 4 quantifies the results for all three dimensions of mutant utility in terms of reduction of the mean squared error in comparison to a baseline, which simply computes the expected mutant utility over all mutants in the training set. The table shows how each dimension of program context, in isolation and in combination with the other two, improves the prediction of mutant utility. For comparison, the table also shows the reduction of the mean squared error when predicting mutant utility based on mutation operator groups (Grp) and individual mutation operators (Op). Note that this experiment aims to compare the relative performance of individual predictors and that training an actual machine learning classifier is likely to yield larger error reductions.

Parent statement context is the strongest predictor of mutant utility in isolation, but adding children and data type context shows additional improvements. This suggests that more complex program context models should consider and refine all three dimensions. The results also show that program context information should be considered for individual mutation operators rather than mutation operator groups.

5.8 Context-Based Mutant Selection

Sections 5.5–5.7 show the results for each of the three dimensions of mutant utility (equivalence, triviality, and dominance) in isolation. These results suggest that program context is indeed a strong predictor of mutant utility, but individually they do not provide enough information to select mutation operators in particular program contexts that yield predominantly dominator mutants with few equivalent and trivial mutants. To do that effectively, we must consider the expected mutant utility of mutation operators and program contexts in multiple dimensions.

Figure 8 shows the distribution of the expected mutant utility and explores whether mutant utility can be optimized along all three dimensions. Each data point represents the expected mutant utility of a particular mutation operator in a particular program context. The green color coded operator/context combinations close to $(1, 1, 1)$ generate highly desirable mutants—high expected dominance with very low expected equivalence and triviality. In

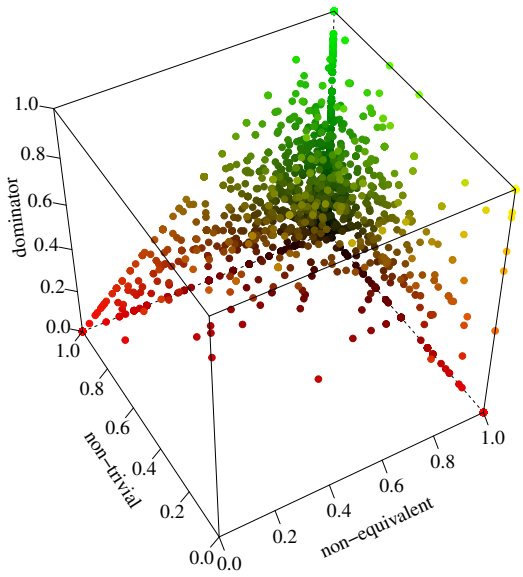


Figure 8: Distribution of the expected mutant utility.

Each data point represents the expected utility of a mutant generated by a particular mutation operator in a particular program context.

contrast, the red color coded operator/context combinations close to $(0, 1, 0)$ and $(1, 0, 0)$ generate highly undesirable mutants—low expected dominance with very high equivalence or triviality.

The overall goal is to select those operator/context combinations that are close to $(1, 1, 1)$, but specific thresholds and weights for each dimension of mutant utility depend on the use case and are subject to future research. For example, equivalent mutants are a major concern in test generation, as they cause too much work for a tester. Trivial and subsumed mutants do not cause as much work but inflate the mutation score [21, 24].

5.9 Threats to Validity

Prior studies (e.g., [4]) have reported that less than 2% of non-equivalent mutants are dominators; in contrast, we observed 33% in our data set. One reason for this discrepancy is that former studies used Proteum, a mutation system with a very large set of mutation operators. This poses a threat to external validity, and future research should verify whether our findings generalize to this significantly richer set of mutation operators. However, we argue that this seemingly large discrepancy is likely not a major threat given that some Proteum mutation operators are specifically designed to implement basic coverage criteria via trivial mutants, which inevitably introduce redundancy and can be easily marked trivial—even in the absence of context information.

The approximation of mutant utility and the lack of ground truth introduces noise and is a threat to construct validity; it is possible that ground-truth knowledge would change our results. This threat is, however, unlikely to significantly impact our results for three reasons. First, we mitigated this threat by selecting only subjects with thorough test suites to ensure good approximations. Second, our ratio of equivalent mutants (15.6%) is on par with prior studies on Java programs that found 16% of mutants to be equivalent [34]. Third, if an exception is observed on all extant tests that cover a particular mutant, it is reasonable to deem the mutant undesirable even if an undiscovered test exists that would not result in an exception.

6 RELATED WORK

Mathur [26] determined that the complexity of mutation testing is $O(n^2)$, where n is the size of the program under test, and introduced the idea of *constrained mutation* to reduce that complexity to $O(n)$ by reducing the number of mutation operators to create fewer mutants. Offutt et al. [29, 31] took an empirical approach to defining an appropriate set of selective mutation operators, and proposed the *E-selective* set of five operators, while Wong et al. [36, 37] evaluated combinations of mutation operators for efficiency and effectiveness.

Other researchers have examined whether selective mutation is more effective than random sampling of similar numbers of mutants. Acree [1] and Budd [6] separately concluded that executing tests that kill a randomly-selected 10% of mutants could provide results close to executing tests that kill the full set of mutants. Wong and Mathur demonstrated similar results [37]. More recently, Zhang et al. [39] and Gopinath et al. [9, 11] also found no appreciable difference in performance between selective mutation and random selection. Kurtz et al. reassessed the performance of *E-Selective* mutation using dominator mutation score [24] and found it indistinguishable from both statement deletion and random mutant selection. Gopinath et al. [9, 11] expanded this investigation using a much larger body of open-source code and compared several different mutation selection strategies with random selection, again finding that random selection performs as well as any other strategy. In a later paper, Gopinath et al. [10] took a different approach to dealing with the large number of mutants, and showed that determining the mutation score based on as few as 1,000 randomly-selected mutants provides an estimate of quality of a test suite in terms of mutation score.

Several researchers have addressed the notion that some mutants are more valuable than others. Kurtz et al. rendered this notion more precisely using subsumption graphs [23]. Yao et al. [38] suggested the notion of a *stubborn* mutant, defined as one that is not killed by a branch-adequate test set. Namin et al. [27] proposed *MuRanker*, a tool that identifies hard-to-kill mutants based on the syntactic distance between the mutant and the original program. They postulate the existence of "super mutants" that are hard to kill and for which a killing test may also kill a number of other mutants.

Kaminski et al. [22] were the first to consider mutation operators at the next level of detail, recognizing that the mutation operator group that targets relational operators includes many redundant mutation operators. They showed that, for any given relational operator, three mutants will always weakly subsume the other four mutants, making them redundant. Just et al. [21] performed a similar analysis targeting conditional operators, and Yao et al. [38] targeted arithmetic operators.

Jia et al. surveyed mutation testing in general and provided a detailed review of mutation equivalence detection techniques [15]. Baldwin and Sayward described how compiler optimization techniques could detect equivalent mutants [5]. Offutt and Craft investigated this approach in the context of Mothra [28], and reported an equivalent mutant detection rate of 15%. An extension of this approach to include infeasible constraint detection enabled Offutt and Pan to improve the detection rate to 45% [30]. More recently, Papadakis et al. [32] proposed equivalent mutant detection via compiler optimizations, where program binaries are simply compared via diff. Surprisingly, this very simple approach yields a detection rate of 7% to 21%, with a reported potential improvement to 30%. Voas and McGraw suggested a program slicing approach to equivalent mutant detection [35], which Hierons et al. [14] formalized and Harman et al. extended with fine-grained dependence [12].

Schuler and Zeller [34] rank equivalent mutants based on impact in terms of *coverage difference*, the difference in statement coverage between the original artifact and the mutant. Just et al. [17, 18] took this approach a step further and identify test-equivalent and equivalent mutants using state infection and local propagation conditions. While effective, these methods require a pre-existing test suite.

Harman et al. argued that a Higher-Order Mutant (HOM) approach might introduce fewer equivalent mutants than first-order approaches, and that a co-evolutionary approach to mutant generation should "almost guarantee that no equivalent mutants will be created" [2, 13].

7 CONCLUSIONS

Existing selective mutation approaches, which ignore program context, do not outperform random mutant selection. As a consequence, selective mutation, as currently defined, fails to eliminate the many low-utility (equivalent, trivial, and redundant) mutants that existing mutation techniques produce. This paper forges a new path out of this dilemma, making four contributions.

First, it shows, via motivational examples, why any useful selective mutation approach must be context-sensitive and must consider individual mutation operators rather than operator groups.

Second, it proposes a concrete model of mutant utility along three dimensions of equivalence, triviality, and dominance.

Third, it proposes to model program context using information extracted from a program's abstract syntax tree—in particular, information about parent nodes, child nodes, and data types.

Fourth, it studies the connection between program context and mutant utility, and shows that context information helps to predict mutant utility: some program contexts are more likely than others to yield mutants that are equivalent, or trivial, or have high dominator strength. The empirical results show that program context can predict mutant utility across a wide range of mutation operators.

One promising next step is to explore a variety of more complex program context models and (non-linear) machine learning classifiers to evaluate their predictive capacity for mutant utility. We conjecture that more complex models of and interactions between different dimensions of program context are likely to yield further improvements. Another step is to investigate the trade-offs and interactions between equivalence, triviality, and dominance to determine which thresholds and combinations yield the strongest test set with the least effort.

Our broader vision is to customize program mutation to a target program, only generating mutants with high utility for that program. This paper is a first step towards realizing this vision. We hope that the promising results will inspire additional research that will ultimately yield a turn key, context-sensitive mutation technique that engineers will adopt in practice.

ACKNOWLEDGEMENTS

We would like to thank Huzefa Rangwala, Andrew McCallum, and the anonymous reviewers for helpful comments and suggestions. Bob Kurtz's research is supported by Raytheon.

REFERENCES

[1] Alan T. Acree. 1980. *On Mutation.* Ph.D. Dissertation. Georgia Institute of Technology, Atlanta, GA.

[2] Konstantinos Adamopoulos, Mark Harman, and Robert M. Hierons. 2004. How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-Evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference*. Springer LNCS 3103, 1338–1349.

[3] Miltiadis Allamanis, Earl T. Barr, René Just, and Charles Sutton. 2016. Tailored mutants fit bugs better. *arXiv preprint arXiv:1611.02516* (2016).

[4] Paul Ammann, Marcio E. Delamaro, and Jeff Offutt. 2014. Establishing Theoretical Minimal Sets of Mutants. In *7th IEEE International Conference on Software Testing, Verification and Validation (ICST 2014)*. Cleveland, Ohio, USA, 21–31.

[5] Douglas Baldwin and Fred G. Sayward. 1979. *Heuristics for Determining Equivalence of Program Mutations.* Research Report 276. Department of Computer Science, Yale University.

[6] Tim A. Budd. 1980. *Mutation Analysis of Program Test Data.* Ph.D. Dissertation. Yale University, New Haven, Connecticut, USA.

[7] Richard A. DeMillo, Richard J. Lipton, and Fred G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer* 11, 4 (April 1978), 34–41.

[8] Bradley Efron and Robert J Tibshirani. 1994. *An introduction to the bootstrap*. CRC press.

[9] Rahul Gopinath, Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. 2015. *Do Mutation Reduction Strategies Matter?* Technical Report. School of Electrical Engineering and Computer Science, Oregon State University, Corvallis, Oregon, USA.

[10] Rahul Gopinath, Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. 2015. How hard does mutation analysis have to be anyway?. In *IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*.

[11] Rahul Gopinath, Mohammad Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. 2016. On the Limits of Mutation Reduction Strategies. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 511–522.

[12] Mark Harman, Rob Hierons, and Sebastian Danicic. 2001. Mutation Testing for the New Century. Kluwer Academic Publishers, Chapter The Relationship Between Program Dependence and Mutation Analysis, 5–13.

[13] Mark Harman, Yue Jia, and William Langdon. 2010. A Manifesto for Higher Order Mutation Testing. In *Sixth IEEE Workshop on Mutation Analysis (Mutation 2010)*. Paris, France, 80–89.

[14] Rob Hierons, Mark Harman, and Sebastian Danicic. 1999. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification, and Reliability, Wiley* 9, 4 (December 1999), 233–262.

[15] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions of Software Engineering* 37, 5 (September 2011), 649–678.

[16] René Just. 2014. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 433–436.

[17] René Just, Michael D. Ernst, and Gordon Fraser. 2013. Using state infection conditions to detect equivalent mutants and speed up mutation analysis. In *Proceedings of the Dagstuhl Seminar 13021: Symbolic Methods in Testing*, Vol. abs/1303.2784. arXiv:1303.2784, preprint.

[18] René Just, Michael D. Ernst, and Gordon Fraser. 2014. Efficient mutation analysis by propagating and partitioning infected execution states. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. San Jose, CA, USA, 315–326.

[19] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. San Jose, CA, USA, 437–440.

[20] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *FSE 2014, Proceedings of the ACM SIGSOFT 22nd Symposium on the Foundations of Software Engineering*. Hong Kong, 654–665.

[21] René Just and Franz Schweiggert. 2015. Higher accuracy and lower run time: efficient mutation analysis using non-redundant mutation operators. *Software Testing, Verification, and Reliability, Wiley* 25, 5-7 (2015), 490–507.

[22] Garrett Kaminski, Paul Ammann, and Jeff Offutt. 2013. Improving Logic-Based Testing. *Journal of Systems and Software, Elsevier* 86 (August 2013), 2002–2012. Issue 8.

[23] Bob Kurtz, Paul Ammann, Marcio E. Delamaro, Jeff Offutt, and Lin Deng. 2014. Mutation Subsumption Graphs. In *Tenth IEEE Workshop on Mutation Analysis (Mutation 2014)*. Cleveland, Ohio, USA, 176–185.

[24] Robert Kurtz, Paul Ammann, Jeff Offutt, Márcio E. Delamaro, Mariet Kurtz, and Nida Gökçe. 2016. Analyzing the Validity of Selective Mutation with Dominator Mutants. In *FSE 2016, Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. Seattle, Washington, 571–582.

[25] Birgitta Lindström and András Márki. 2016. On Strong Mutation and Subsuming Mutants. In *Twelfth IEEE Workshop on Mutation Analysis (Mutation 2016)*. Chicago, Illinois, USA.

[26] Aditya Mathur. 1991. Performance, Effectiveness, and Reliability Issues in Software Testing. In *Proceedings of the Fifteenth Annual International Computer Software and Applications Conference*. Tokyo, Japan, 604–605.

[27] Akbar Namin, Xiaozhen Xue, Omar Rosas, and Pankaj Sharma. 2015. MuRanker: A mutant ranking tool. *Software Testing, Verification, and Reliability* 25, 5-7 (August 2015), 572–604.

[28] Jeff Offutt and William Michael Craft. 1994. Using Compiler Optimization Techniques to Detect Equivalent Mutants. *Software Testing, Verification, and Reliability, Wiley* 4, 3 (September 1994), 131–154.

[29] Jeff Offutt, Ammei Lee, Gregg Rothermel, Roland Untch, and Christian Zapf. 1996. An Experimental Determination of Sufficient Mutation Operators. *ACM Transactions on Software Engineering Methodology* 5, 2 (April 1996), 99–118.

[30] Jeff Offutt and Jie Pan. 1997. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification, and Reliability* 7, 3 (September 1997), 165–192.

[31] Jeff Offutt, Gregg Rothermel, and Christian Zapf. 1993. An Experimental Evaluation of Selective Mutation. In *Proceedings of the International Conference on Software Engineering (ICSE)*. Baltimore, MD, 100–107.

[32] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. 2015. Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique. In *Proceedings of the International Conference on Software Engineering (ICSE)*. Florence, Italy, 936–946.

[33] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *Proceedings of the International Conference on Software Engineering (ICSE)*. Buenos Aires, Argentina, 609–620.

[34] David Schuler and Andreas Zeller. 2013. Covering and uncovering equivalent mutants. *Software Testing, Verification, and Reliability, Wiley* 23, 5 (2013), 353–374.

[35] Jeffrey M. Voas and Gary McGraw. 1997. *Software Fault Injection: Inoculating Programs Against Errors.* John Wiley & Sons, Inc., New York, NY, USA.

[36] W. Eric Wong, Márcio E. Delamaro, José C. Maldonado, and Aditya P. Mathur. 1994. Constrained Mutation in C Programs. In *Proceedings of the 8th Brazilian Symposium on Software Engineering*. Curitiba, Brazil, 439–452.

[37] W. Eric Wong and Aditya P. Mathur. 1995. Reducing the Cost of Mutation Testing: An Empirical Study. *Journal of Systems and Software, Elsevier* 31, 3 (December 1995), 185–196.

[38] Xiangjuan Yao, Mark Harman, and Yue Jia. 2014. A Study of Equivalent and Stubborn Mutation Operators using Human Analysis of Equivalence. In *Proceedings of the International Conference on Software Engineering (ICSE)*. Hyderabad, India, 919–930.

[39] Lu Zhang, Shan-San Hou, Jun-Jue Hu, Tao Xie, and Hong Mei. 2010. Is operator-based mutant selection superior to random mutant selection?. In *Proceedings of the International Conference on Software Engineering (ICSE)*. Cape Town, South Africa, 435–444.