Higher Accuracy and Lower Run Time: Efficient Mutation Analysis Using Non-Redundant Mutation Operators

René Just[1]*and Franz Schweiggert[2]

[1]*Department of Computer Science, University of Washington, USA*
[2]*Department of Applied Information Processing, Ulm University, Germany*

Mutation analysis is a powerful but computationally expensive method to measure the effectiveness of a testing or debugging technique. The high cost is due, in part, to redundant mutants generated by commonly used mutation operators. A mutant is said to be *redundant* if its outcome can be predicted based on the outcome of other mutants. The execution of those redundant mutants is unnecessary and wastes CPU resources. Moreover, the inclusion of redundant mutants may lead to a skewed mutant detection rate and therefore misrepresent the effectiveness of the assessed testing or debugging technique.

This paper extends previous work and makes the following contributions. First, it defines and provides non-redundant versions of the conditional operator replacement (COR), unary operator insertion (UOI), and relational operator replacement (ROR) mutation operators. Second, it reports on a conducted empirical study using ten real-world programs that comprise a total of 410,000 lines of code. The empirical study used developer-written and generated test suites. The results show how prevalent redundant mutants are and how their elimination improves the efficiency and accuracy of mutation analysis. In summary, the total mutation analysis run time decreased by more than 20% by removing redundant mutants, and the inclusion of redundant mutants led to an overestimated mutation score for all analyzed test suites.

1. INTRODUCTION

Originally introduced by Budd [1] and DeMillo [2], mutation analysis measures the effectiveness of a testing or debugging technique based on artificial faults that are systematically seeded into a program under test (the *original program*). Each of those seeded faults leads to a small syntactic variation of the original program, called *mutant*. The syntactic change within such a mutant is referred to as *mutation*, which is produced by a *mutation operator*. Effectiveness is quantified in the *mutation score*, which is the percentage of mutants that a test can distinguish from the original program. A test that can distinguish a mutant from the original program is said to *detect* that mutant.

*Correspondence to: René Just, rjust@cs.washington.edu

Accepted for publication in the journal of Software Testing, Verification and Reliability

Many test executions are necessary to calculate the mutation score — in the worst case one execution for every single mutant. Since even small programs can lead to large numbers of mutants, mutation analysis may be prohibitively time-consuming and computationally expensive, especially in comparison to structural code coverage criteria [3].

However, not every mutant adds value to the analysis. Previous studies empirically investigated redundancies between mutation operators and showed that a certain subset of all applicable mutation operators is sufficient to measure test effectiveness [4, 5]. The obtained subset of mutation operators, referred to as *sufficient mutation operators*, significantly improves efficiency but incurs a minor loss of information in terms of the mutation score. Both previous studies considered the mutation operators to be atomic, meaning that the set of mutants derived from a specific mutation operator was either included as a whole or not at all. For instance, a replacement operator was either applied with all valid replacements or entirely excluded.

This paper considers the set of sufficient mutation operators at a fine-grained level and shows that their original definition implies redundancy in the resulting set of mutants. It formally describes the requirements for a sufficient set of non-redundant mutations and provides a non-redundant version of the conditional operator replacement (COR), unary operator insertion (UOI), and relational operator replacement (ROR) mutation operators. Additionally, the empirical study demonstrates how prevalent redundant mutants are for real-world programs and how their elimination improves the efficiency of mutation analysis. The empirical study also shows how the inclusion of redundant mutants leads to an inaccurate mutation score.

In summary, this paper makes the following contributions with regard to the effect of redundant mutants on the efficiency and accuracy of mutation analysis:

- A definition of a sufficient set of non-redundant mutations that collectively subsume all other mutations of a given mutation operator.
- A determination of a subsumption hierarchy for the COR, UOI, and ROR mutation operators. The paper shows that only 4 out of 10 COR and UOI mutations and 3 out of 7 ROR mutations are necessary to form a sufficient set of non-redundant mutations.
- An empirical study that investigates how redundant mutants derived from the COR, UOI, and ROR mutation operators affect the efficiency and accuracy of mutation analysis for ten real-world programs totaling 410,000 lines of code. The results show that eliminating redundant mutants significantly decreases the mutation analysis run time and improves the accuracy of the mutation score.
- A comparison of mutation coverage (i.e., ratio of reached and executed mutations) with statement coverage, branch coverage, and the mutation score. The results show a very strong correlation between mutation coverage and statement and branch coverage, but only a moderate correlation between mutation coverage and mutation score.

The remainder of the paper is structured as follows: Section 2 furnishes a detailed view of mutation operators and defines a sufficient set of non-redundant mutations for the COR, UOI, and ROR mutation operators. Section 3 describes the empirical study and reports on the corresponding results. Section 4 describes related work, and Section 5 concludes the paper.

2. A DETAILED VIEW OF MUTATION OPERATORS

A wide variety of mutation operators has been proposed for different purposes and programming languages (e.g., [6, 7, 8, 9]). The actual set of mutation operators that can be employed depends on the programming language, and this paper considers the following set of mutation operators for Java programs that are supported by the Major mutation framework [10]:

- Operator Replacement Binary: Replace all occurrences of binary operators (e.g., arithmetic, logical, shift, conditional, or relational operators) with all valid alternatives. Note that the COR and ROR mutation operators studied in this paper belong to this class.
- Operator Replacement Unary: Replace all occurrences of unary operators with all valid alternatives.
- Unary Operator Insertion (UOI): Insert a unary boolean operator to negate boolean expressions. This operator is also applied to subexpressions and boolean literals.
- Literal Value Replacement: Replace all numerical literals with a positive value, a negative value, and zero. Replace reference type variables with a reference to `null`.

The remainder of this section considers the COR, UOI, and ROR mutation operators at a fine-grained level. It defines non-redundant and subsumed mutations, and provides a sufficient set of non-redundant mutations for those three mutation operators. Section 2.1 formally defines a sufficient set of non-redundant mutations. Section 2.2 employs those definitions and provides a sufficient set of non-redundant mutations for the COR and UOI mutation operators. Section 2.3 discusses the definitions in the context of composed conditional expressions. Section 2.4 provides a sufficient set of non-redundant mutations for the ROR mutation operator.

2.1. Definition of a sufficient set of non-redundant mutations

Recall that *mutation* refers to a syntactic change (e.g., `a && b` $\mapsto$ `a || b`), whereas *mutant* refers to the program that includes a single mutation.

Definition 1
Semantically equivalent mutant

Let p be the original program and p_m be a mutant of p. Let further Ω be the input domain of p and $\omega \in \Omega$ be an input tuple — that is, ω denotes a tuple of input values and configuration parameters, necessary to execute p. A mutant p_m is semantically equivalent to p, written as $p_m \equiv p$, if and only if p_m yields the same output as p for all possible input tuples:

$$p_m \equiv p :\Leftrightarrow p_m(\omega) = p(\omega), \forall \omega \in \Omega$$

Definition 2
Semantically equivalent mutation

Let p_m be a mutant of program p and $op \in p$ be an n-nary operator, referred to as *original version*. Let further m be a mutation of op and $\pi \in \Pi$ be an n-nary input tuple. A mutation m is semantically equivalent to op if and only if m computes the same output as op for all possible input tuples:

$$m \equiv op :\Leftrightarrow m(\pi) = op(\pi), \forall \pi \in \Pi$$

In the context of first-order mutants (i.e., mutants that contain only one mutation at a time), equivalence of a mutation implies equivalence of the corresponding mutant $m \equiv op \Rightarrow p_m \equiv p$. The converse of this implication does not necessarily hold as a mutant p_m can be equivalent to the original program p even if the output of the corresponding mutation m differs from op — an example is a missing propagation of that difference to an observable program output [11].

Definition 3
Trivial mutation

Let m be a mutation of an original version op and $\pi \in \Pi$ be an input tuple. A mutation m is a trivial mutation if and only if its output differs from op for every possible input tuple:

$$m \text{ is trivial mutation} :\Leftrightarrow m(\pi) \neq op(\pi), \forall \pi \in \Pi$$

Trivial mutations are not desirable since they are likely to be detected independently of the actual input tuple. A mutation should rather be a subtle change, making it hard to detect — that is, a mutation should be similar to its original version. Such a similarity between a mutation and its original version can be described as the distance between the output values.

Definition 4
Distance between mutation and original version $d(m, op)$

Let $I = \{1, \ldots, |\Pi|\}$ be an index set enumerating all possible input tuples $\pi_i \in \Pi$. Let further m be a mutation of an original version op. The distance between m and op, written as $d(m, op)$, is defined as the L_1 norm:

$$d(m, op) := \|m - op\|_1 = \sum_{i \in I} |m(\pi_i) - op(\pi_i)|$$

Considering COR, UOI, and ROR mutations for which the outputs of m and op are boolean, the distance $d(m, op)$ is equal to the number of input tuples for which the outputs of m and op differ.

Definition 5
Minimal-distance mutation

Let $I = \{1, \ldots, |\Pi|\}$ be an index set enumerating all possible input tuples $\pi_i \in \Pi$. Let m further be a mutation of an original version op. A mutation m is a minimal-distance mutation if the distance between m and op is 1:

$$m \text{ is minimal-distance mutation} :\Leftrightarrow d(m, op) = 1 \Leftrightarrow \exists! \pi_i \in \Pi : m(\pi_i) \neq op(\pi_i)$$

Definition 5 indicates that there exists exactly one input tuple π_i for which the output of a minimal-distance mutation differs from op. Note that a distance of 0 implies equivalence.

Definition 6
Subsumed mutation

Let $I = \{1, \ldots, |\Pi|\}$ be an index set enumerating all possible input tuples $\pi_i \in \Pi$. Let m_1 and m_2 further be two mutations for an original version op. A mutation m_1 subsumes a mutation m_2 if detecting m_1 implies detecting m_2:

$$m_1 \text{ subsumes } m_2 :\Leftrightarrow \exists i \in I_\Delta(m_2) : m_1(\pi_i) = m_2(\pi_i) \wedge I_\Delta(m_1) \subseteq I_\Delta(m_2),$$

$$I_\Delta(m) = \{i \in I \mid m(\pi_i) \neq op(\pi_i)\}$$

The index set $I_\Delta(m)$ represents all input tuples for which the output of a mutation m differs from the output of its original version op.

Definition 7
Sufficient set of non-redundant mutations M_{suf}

Let $I = \{1, \ldots, |\Pi|\}$ be an index set enumerating all possible input tuples $\pi_i \in \Pi$ and M be the set of all possible mutations for an original version op. Let further $f : I \mapsto M$ be an injection that maps an index $i \in I$ to a minimal-distance mutation m_i (i.e., $m_i = f(i)$) for the input tuple π_i. The sufficient set of non-redundant mutations $M_{\text{suf}} \subset M$ is defined as follows:

$$M_{\text{suf}} := \{m_i \mid i \in I\}$$

The set M_{suf} is minimal and subsumes all other possible mutations:

- M_{suf} is minimal because it contains exactly one minimal-distance mutation for each input tuple, and hence the index sets $I_\Delta(m_i)$ are disjoint. □
- M_{suf} subsumes all other mutations because it contains one minimal-distance mutation for each input tuple and from Definition 5 follows that $|I_\Delta(m_i)| = 1$ for each such minimal-distance mutation. Hence, Definition 6 is fulfilled for every additional mutation. □

The subsequent sections show that such a set M_{suf} indeed exists for the COR, UOI, and ROR mutation operators.

2.2. Non-redundant COR mutation operator

Generally, the COR mutation operator replaces a boolean expression `a <op> b` where `a` and `b` denote boolean subexpressions or literals and `<op>` is one of the logical connectors `&&` or `||`. Valid mutations for such a boolean expression belong to one of the following three categories:

1. Conditional operator
   - Logical connector `&&`: `a && b`
   - Logical connector `||`: `a || b`
   - Equivalence operator: `a == b`
   - Exclusive OR operator: `a != b`
2. Special operator
   - Evaluation to left hand side: `lhs`
   - Evaluation to right hand side: `rhs`
   - Evaluation to true: `true`
   - Evaluation to false: `false`
3. Unary boolean operator
   - Negation of left operand: `!a <op> b`
   - Negation of right operand: `a <op> !b`
   - Negation of expression: `!(a <op> b)`

Note that the three logical operators `^`, `|`, and `&` are omitted for two reasons. First, the logical exclusive OR operator `^`, when applied to boolean values, is semantically equivalent to the included operator `!=`. Hence, the inclusion of both operators would inevitably introduce redundancy. Second, the logical operators `|` and `&` produce the same boolean output as their conditional equivalents with the exception that they do not exploit the possible short-circuit evaluation. The logical operators are excluded since they are subsumed by the mutations included in the three categories.

Table I. Set of non-redundant and subsumed mutations for the logical connector &&. A rectangle indicates for which input tuples (a, b) a mutation's output differs from the original version.

		Original version	**Non-redundant COR mutations**				**Subsumed COR mutations**			**Subsumed UOI mutations**		
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. Set of non-redundant and subsumed mutations for the logical connector ||. A rectangle indicates for which input tuples (a, b) a mutation's output differs from the original version.

		Original version	**Non-redundant COR mutations**				**Subsumed COR mutations**			**Subsumed UOI mutations**		
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

Based on the given Definition 7, Table I highlights the sufficient set of non-redundant mutations for the logical connector &&. All mutations within this set have a minimal distance, in compliance with Definition 5 — the rectangles in the table highlight the outputs of the mutations that differ from the original version. Furthermore, the input tuples for which these mutations produce an incorrect output are disjoint, and hence their union forms the sufficient set.

Table I also gives all subsumed mutations for the logical connector &&. None of the subsumed mutations fulfills Definition 5 (minimal distance). Besides manifesting a greater distance, the input tuples for which the subsumed mutations lead to an incorrect output are not disjoint. All subsumed mutations fulfill Definition 6 since the minimal-distance mutations collectively cover all possible input tuples. For example, the `lhs` mutation computes an incorrect output for the input tuple `a=1` and `b=0`. If a test detects the `lhs` mutation, the same test also detects all of the subsumed mutations that compute the same incorrect output for this input tuple. The logical negation !(a && b) is furthermore a trivial mutation (Definition 3) since it produces an incorrect output for every input tuple.

Considering the logical connector ||, Table II shows the minimal-distance mutations that form the sufficient set of non-redundant mutations. Additionally, Table II gives all subsumed COR and UOI mutations for this logical connector, where each subsumed mutation again manifests a greater distance, as visualized by the rectangles.

For both logical connectors, the sufficient set of non-redundant COR mutations subsumes all mutations associated with the UOI mutation operator. Hence, the UOI mutation operator should be omitted when mutating conditional expressions due to the manifested redundancy. Note that this subsumption of the UOI mutation operator only holds for boolean expressions.

```
1  public void pattern1 (int x){
2      Var v;
3
4      if(flag && (v=getVar())!=null){
5
6          v.foo(x);
7      }
8      return;
9  }
10
11 public void pattern2 (int x){
12     Var v;
13
14     if(flag || (v=getVar())==null){
15         return;
16     }
17     v.bar(x);
18 }
```

Listing 1. Two common patterns with uninitialized local variables exploiting the short-circuit property of the logical connectors && and ||.

Regarding the sufficient set of non-redundant mutations, where every mutation has a minimal distance of 1 and the corresponding input tuples are disjoint, detecting all mutations requires combinatorial testing of the inputs `a` and `b`. Yet, this relation only holds for conditional expressions with one logical connector as combinatorial testing of composed expressions would require higher-order mutants [12], which are not considered in this paper.

Using the sufficient set of non-redundant mutations, the reduction of the number of mutations generated by the COR and UOI mutation operators is theoretically 60% — only 4 out of 10 mutations are necessary. However, the actual reduction depends on the applicability of the mutation operators. There are two common patterns within Java programs that exploit the short-circuit property of the logical connectors && and `||`, as shown in Listing 1. Within this listing the short-circuit property is utilized to avoid a pre-initialization of a local variable in line 2 and 12.[†] Since the Java compiler strictly requires that every local variable is initialized before use, replacing the logical connector && by `||`, and vice versa, is not valid in the illustrated methods. The application of the special operators `true`, `false`, and `lhs` would also lead to invalid mutations.

The actual reduction of all generated mutants depends on two factors: (1) The number of occurrences of the depicted patterns and (2) the ratio of COR and UOI mutants to all generated mutants. Sections 3.2 and 3.3 provide details about the ratio of COR and UOI mutants and the overall decrease of generated mutants.

[†]The Java Virtual Machine is a stack machine, and thus a local variable is only stored on the stack once it has been initialized.

2.3. Composed conditional expressions

The definitions and subsumption hierarchies in Section 2.1 and 2.2 are given for conditional expressions with two literals, referred to as base case. By considering the following two cases this section shows that the given subsumption hierarchies also hold in composed expressions:

1. The mutated conditional expression is an enclosed expression.
2. The mutated conditional expression is an enclosing expression.

Recall that only first-order mutants are considered, which means that each mutant contains exactly one mutation. Consider the following example:

$$\overbrace{\underbrace{\texttt{a \&\& b}}_{expr_1}\texttt{ || c}}^{expr_2}$$

Regarding the first case in which the enclosed $expr_1$ is mutated, the given subsumption hierarchies hold because of the following two reasons:

1. Given the well-defined semantics for the evaluation of an expression, the evaluation of the enclosing expression $expr_2$ has no impact on the values of the literals `a` and `b`.
2. Let op_1 be the operator of an enclosed expression $expr_1$ and op_2 be the operator of an enclosing expression $expr_2$. Let further m_i be a minimal-distance mutation and m_s be a subsumed mutation of the operator op_1. The following two implications hold for every mutation m_i and m_s:

$$\forall a, b : m_i(a, b) = op_1(a, b) \Rightarrow op_2(m_i(a, b), c) = op_2(op_1(a, b), c)$$

$$\forall a, b : m_i(a, b) = m_s(a, b) \Rightarrow op_2(m_i(a, b), c) = op_2(m_s(a, b), c)$$

In other words, the subsumption hierarchy holds for an enclosed expression since its outcome is not affected by the evaluation of an enclosing expression.

Regarding the second case in which the enclosing expression $expr_2$ is mutated, the enclosed expression $expr_1$ can be viewed as a predicate `p`, which is not mutated:

$$\underbrace{\texttt{a \&\& b}}_{\texttt{p}}\texttt{ || c}$$

This simplifies the mutated expression to `p || c`, which represents the base case and, thus, the given subsumption hierarchy holds for the expression `p || c`.

2.4. Non-redundant ROR mutation operator

The ROR mutation operator targets expressions `a <op> b` with `<op>` representing one of the 6 relational operators (`<`, `<=`, `>`, `>=`, `==`, `!=`). The ROR mutation operator replaces such a relational operator with all other relational operators. Additionally, it applies the special operators `true` and `false` — since every relational operator maps to a boolean output, the entire expression can be replaced with `true` and `false`, respectively.

The definitions given in Section 2.1 are also applicable for the ROR mutation operator. Rather than considering the actual values of the inputs `a` and `b`, all possible input tuples (`a`, `b`) are grouped into three intervals representing the relation between `a` and `b`: `a < b`, `a == b`, and `a > b`.

Table III. Set of non-redundant and subsumed mutations for all relational operators. Rectangles indicate for which intervals a mutation's output differs from the original version.

Interval	Original version	Non-redundant ROR mutations			Subsumed ROR mutations			
	a > b	false	a >= b	a != b	a < b	a <= b	a == b	true
a < b	0	0	0	[1]	[1]	[1]	0	[1]
a == b	0	0	[1]	0	0	[1]	[1]	[1]
a > b	1	[0]	1	1	[0]	[0]	[0]	1
	a < b	false	a <= b	a != b	a > b	a >= b	a == b	true
a < b	1	[0]	1	1	[0]	[0]	[0]	1
a == b	0	0	[1]	0	0	[1]	[1]	[1]
a > b	0	0	0	[1]	[1]	[1]	0	[1]
	a == b	false	a <= b	a >= b	a > b	a != b	a < b	true
a < b	0	0	[1]	0	0	[1]	[1]	[1]
a == b	1	[0]	1	1	[0]	[0]	[0]	1
a > b	0	0	0	[1]	[1]	[1]	0	[1]
	a >= b	true	a > b	a == b	a < b	a <= b	a != b	false
a < b	0	[1]	0	0	[1]	[1]	[1]	0
a == b	1	1	[0]	1	0	[1]	[0]	[0]
a > b	1	1	1	[0]	[0]	[0]	1	[0]
	a <= b	true	a < b	a == b	a > b	a >= b	a != b	false
a < b	1	1	1	[0]	[0]	[0]	1	[0]
a == b	1	1	[0]	1	0	[1]	[0]	[0]
a > b	0	[1]	0	0	[1]	[1]	[1]	0
	a != b	true	a < b	a > b	a == b	a >= b	a <= b	false
a < b	1	1	1	[0]	[0]	[0]	1	[0]
a == b	0	[1]	0	0	1	[1]	[1]	0
a > b	1	1	[0]	1	[0]	1	[0]	[0]

Using the three intervals `a < b`, `a == b`, and `a > b`, Table III shows the sufficient and subsumed mutations for all relational operators. Since there are only three intervals for all possible input tuples, the sufficient set of non-redundant mutations for each relational operator contains three minimal-distance mutations (Definition 5 and 7). Note that Kaminski et al. [13] previously suggested the same sets of sufficient ROR mutations, using fault hierarchies. Therefore, the given subsumption hierarchies for the ROR mutation operator confirm those prior results, using a different approach (i.e., sufficient set of minimal-distance mutations).

Table IV. Summary of the subject programs investigated in the empirical study. Non-comment and non-blank lines of code (LOC) as reported by SLOCCount [15].

	Application	Version	LOC	Relational operators	Conditional operators
trove	GNU Trove	3.0.2	116,750	7,937	1,945
chart	jFreeChart	1.0.13	91,174	2,762	781
itext	iText	5.0.6	76,229	5,293	1,760
math	Commons Math	2.1	39,991	3,233	428
time	Joda-Time	2.0	27,139	1,324	364
lang	Commons Lang	3.0.1	19,495	1,618	695
jdom	JDOM	2beta4	15,163	1,023	216
jaxen	Jaxen	1.1.3	12,440	815	159
io	Commons IO	2.0.1	7,908	345	139
num4j	Numerics4j	1.3	3,647	312	133
total			409,936	24,662	6,620

3. EMPIRICAL EVALUATION

Given the findings of Section 2, the authors implemented the non-redundant versions of the COR, UOI, and ROR mutation operators in the Major mutation framework [10, 14]. This section describes an empirical study that used this enhanced version of Major to investigate the effect of redundant mutants on the efficiency and accuracy of mutation analysis.

3.1. Methodology

The goal of this empirical study was to investigate the effects of redundant mutants on the efficiency and accuracy of mutation analysis. Specifically, the empirical study aimed to answer the following four research questions:

RQ1: What is the ratio of COR, UOI, and ROR mutants compared to all generated mutants?
See Section 3.2.

RQ2: How does the elimination of redundant mutants affect the efficiency of mutation analysis?
See Section 3.3.

RQ3: How does the elimination of redundant mutants affect the mutation score?
See Section 3.4.

RQ4: How does the elimination of redundant mutants affect the mutation coverage ratio?
See Section 3.5.

To answer those research questions, the empirical study considered the ten open-source subject programs that are summarized in Table IV. Since the study focused on redundant mutants associated with the COR, UOI, and ROR mutation operators, Table IV gives the counts for the occurrences of relational and conditional operators in addition to the general description and lines of code.

The following sections report on the effects of redundant mutants on the efficiency and accuracy of mutation analysis, according to the following two sets of mutants:

- M_{all}: Set of all mutants that includes subsumed COR, UOI, and ROR mutants (baseline).
- M_{red}: Reduced set of mutants that does not include subsumed COR, UOI, and ROR mutants.

Table V. Characteristics of the analyzed test suites. Statement coverage (StmtCov) and branch coverage (BranchCov) as reported by Cobertura [17]. T_{dev} denotes the developer-written test suites and T_{gen} the generated test suites. The low coverage of T_{dev} for trove is caused by the inclusion of generated classes not tested by T_{dev}. The low coverage of T_{gen} for chart is caused by the inclusion of GUI classes not tested by T_{gen}.

	Classes	T_{dev}			T_{gen}		
		Tests	**StmtCov**	**BranchCov**	**Tests**	**StmtCov**	**BranchCov**
trove	691	544	7%	6%	13,527	77%	67%
chart	585	2,130	57%	46%	3,254	37%	28%
itext	408	75	20%	11%	4,468	57%	46%
math	408	2,169	88%	85%	2,643	68%	62%
time	156	3,855	90%	80%	2,172	75%	64%
lang	99	2,039	93%	90%	2,453	77%	71%
jdom	131	1,723	95%	94%	1,256	64%	47%
jaxen	197	699	78%	55%	1,210	83%	57%
io	100	309	39%	29%	624	57%	54%
num4j	73	218	97%	96%	341	65%	69%

Note that the two sets M_{all} and M_{red} both include mutants from all mutation operators — they only differ in the number of COR, UOI, and ROR mutants.

In addition to the test suites that are released with the subject programs, the EvoSuite [16] test generation tool was employed to generate an additional test suite for each program. EvoSuite was executed with its default configuration using branch coverage as test objective. The following two abbreviations are used to refer to the developer-written and generated test suites, whose characteristics are summarized in Table V:

- T_{dev}: The developer-written test suites that are released with the subject programs.
- T_{gen}: The test suites that are generated with EvoSuite.

This study distinguishes between *generated mutants* and *covered mutants*. A test is said to cover a mutant if it reaches and executes the mutated code. While the number of generated mutants is test-independent and reflects the complexity and structure of the mutated program, the number of covered mutants always depends on a given test suite. The mutation score S is usually defined as the ratio of number of detected mutants to the number of generated mutants. Taking mutation coverage information into account, a more detailed view on the mutation score is $S = C * S_C$ where C denotes the mutation coverage ratio and S_C represents the mutation score with respect to the number of covered mutants [18]. This separation offers a better view on the two aspects of test effectiveness — that is, test coverage and test oracle strength [18, 19].

Besides, exploiting mutation coverage information is the state-of-the-art optimization in mutation testing. A test that does not cover a mutant does not need to be executed on that mutant as it cannot possibly detect it. In order to avoid biased results, the empirical study always employed the mutation coverage optimization when evaluating the efficiency improvements due to the elimination of redundant mutants.

Some mutants lead to infinite loops, for instance, the ones derived from mutating loop conditions. To prevent the mutation analysis process from getting stuck, these infinite loops have to be identified.

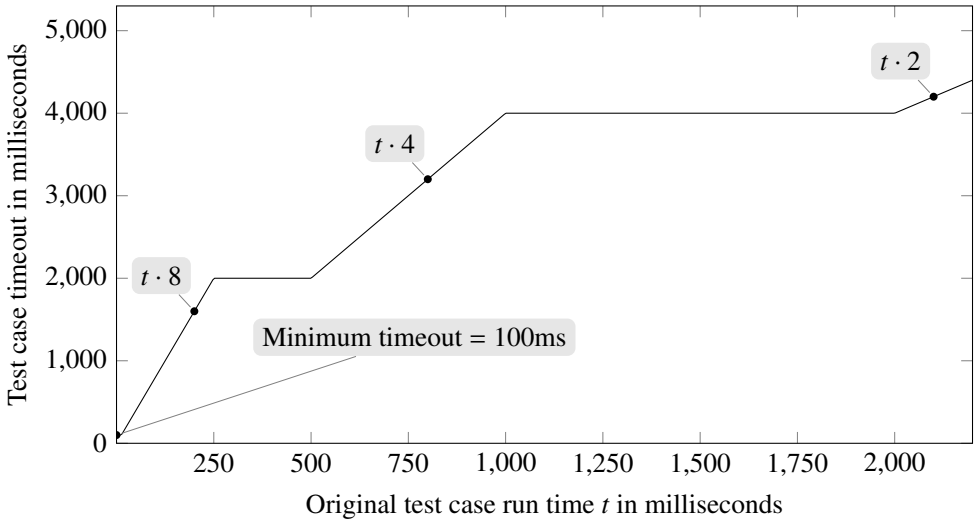

Figure 1. Function used to determine the timeout for mutants (base timeout factor=8).

Since it is undecidable in general whether or not a mutant will eventually terminate, a timeout approximation was applied that depends on the run time of the original program. Figure 1 visualizes the function used to determine the timeout for a certain test. This function uses a base timeout factor, which decreases for longer run times, down to a minimum of two. Because the run time of a short test is more likely to be influenced by small delays, inherent in the system, a considerably larger timeout factor is used for short-running tests in conjunction with a lower bound of 100 milliseconds. Since mutants are usually covered by several tests [20], the timeout is not fixed for a mutant but rather depends on the run time of the test that covers that mutant. With regard to the individual tests of which the run times differ by several orders of magnitude, this variable timeout is more sensitive to varying run times. This approximation, like all heuristics for infinite loop detection, may produce false-positive results. Nevertheless, the determined timeout only leads to an interruption of a test execution if its run time is significantly prolonged. Interruption means that the execution of the test that analyzes a certain mutant is stopped and the corresponding mutant is marked as being detected.

3.2. The ratio of COR, UOI, and ROR mutants

To answer the first research question, this section determines the ratio of mutants generated by the COR, UOI, and ROR mutation operators. Figure 2 visualizes the ratio of mutants associated with the COR, UOI, and ROR mutation operators (dark gray and black bars) compared to the number of mutants generated by applying all mutation operators (light gray bar). Ranging from 31% for *math* to 64% for *trove*, the number of mutants generated by the COR, UOI, and ROR mutation operators is a substantial portion of all generated mutants. The mean ratio of 45% suggests that eliminating redundant COR, UOI, and ROR mutants offers great potential for efficiency improvements. In addition to the ratio of COR and UOI mutants, Figure 3 illustrates, for each subject program, the distribution of the number of logical connectors used in conditional expressions. With a mean value of 80% across the ten programs and a range between 63% for *num4j* and 86% for *math*, the number of conditional expressions with only one connector is predominant for all programs.

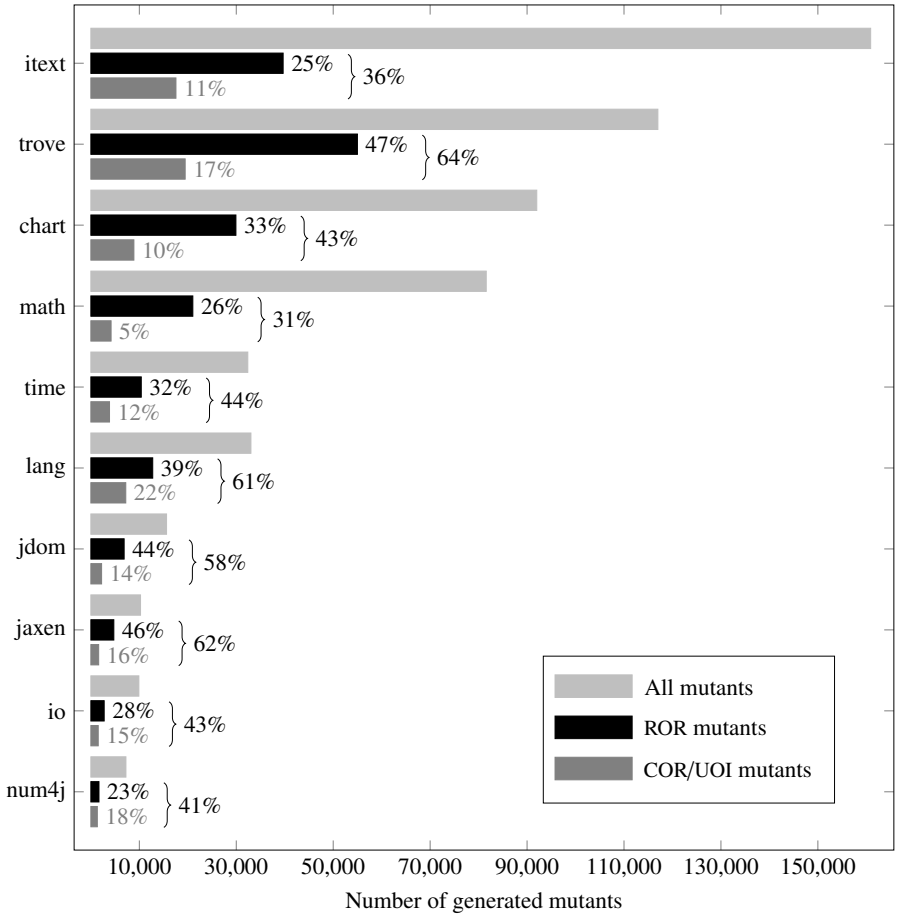

Figure 2. Ratio of the number of COR/UOI and ROR mutants to the number of all generated mutants.

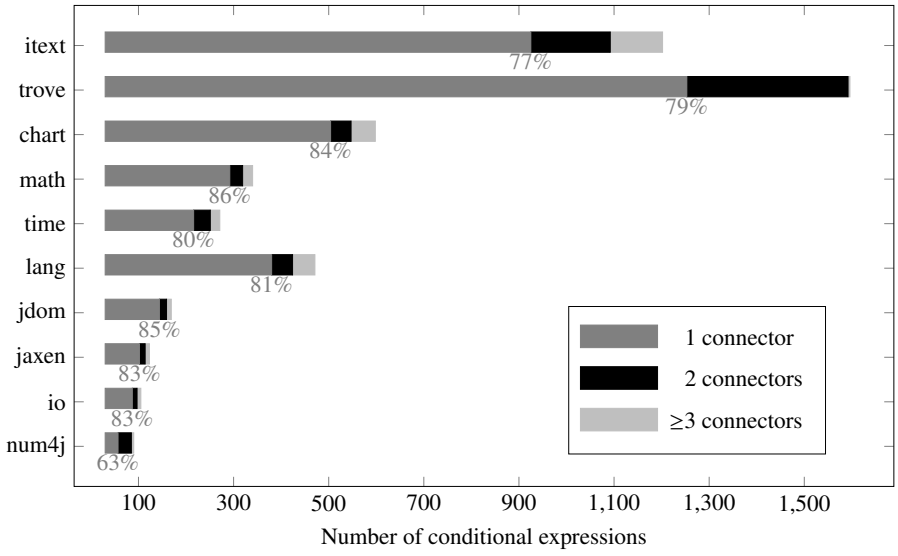

Figure 3. Distribution of the number of logical connectors in conditional expressions.

Table VI. Decrease in the number of generated mutants, total mutation analysis run time, and the number of analyzed mutants. Run time is reported in minutes, and the overall row is calculated by considering all tests over all mutants across all 10 programs. T_{dev} denotes the developer-written test suites and T_{gen} the generated test suites. Baseline is the set of all mutants M_{all}, which includes the subsumed COR, UOI, and ROR mutants. All decreases are statistically significant at the 1% level (Wilcoxon signed-rank test).

	Generated mutants	T_{dev}		T_{gen}	
		Run time	**Analyzed mutants**	**Run time**	**Analyzed mutants**
itext	126,781 (-21%)	427 (-31%)	19,541 (-24%)	1,034 (-28%)	76,245 (-21%)
trove	72,959 (-38%)	38 (-21%)	6,137 (-35%)	96 (-47%)	52,028 (-39%)
chart	68,503 (-26%)	582 (-26%)	36,298 (-28%)	145 (-18%)	21,565 (-28%)
math	66,787 (-18%)	473 (-12%)	60,148 (-19%)	209 (-13%)	43,300 (-21%)
time	23,781 (-27%)	340 (-22%)	19,577 (-29%)	222 (-35%)	17,719 (-30%)
lang	21,056 (-36%)	24 (-37%)	20,196 (-37%)	16 (-48%)	17,735 (-39%)
jdom	10,800 (-31%)	136 (-24%)	10,266 (-32%)	11 (-42%)	4,948 (-39%)
jaxen	7,132 (-30%)	431 (-11%)	4,679 (-30%)	62 (-15%)	4,760 (-37%)
io	7,319 (-26%)	5.0 (-32%)	4,255 (-18%)	1.5 (-38%)	4,998 (-28%)
num4j	5,437 (-25%)	1.8 (-36%)	5,243 (-25%)	1.3 (-38%)	3,747 (-31%)
overall	410,555 (-27%)	2,457 (-22%)	186,340 (-27%)	1,797 (-28%)	247,045 (-29%)

3.3. Effect of redundant mutants on the efficiency of mutation analysis

Answering the second research question, this section reports on the results of a full mutation analysis performed to investigate the actual efficiency improvements due to the elimination of redundant mutants. Using the smaller yet sufficient set of non-redundant mutations for the COR, UOI, and ROR mutation operators implies that fewer mutants are generated overall. As shown in Table VI, using the sufficient set of non-redundant mutations for the COR, UOI, and ROR mutation operators significantly affects the number of generated mutants. The overall number of generated mutants (i.e., all generated mutants for all subject programs) decreased by 27%. Depending on the ratio of the COR/UOI and ROR mutants to all mutants (Figure 2), the decrease ranges between 18% for *math* and 38% for *trove*. Note that this exclusion of mutants does not reduce the effectiveness of the reduced set of mutants M_{red} since only subsumed (redundant) mutants are excluded.

Besides the decrease in the number of generated mutants, Table VI gives, for each subject program, the decrease in the total mutation analysis run time and the number of analyzed mutants for the developer-written and generated test suites. With an overall decrease in run time of 22% for the developer-written test suites and 28% for the generated test suites, the results demonstrate a significant speed-up for all subject programs. The observed speed-up depends on the ratio of the COR, UOI, and ROR mutants generated for the programs and the run time of the tests that do not cover these mutants. For instance, the test suites for *math* and *jaxen* contain a few long-running tests that cover many mutants but only a few COR, UOI, and ROR mutants. Since the run time of these tests is a considerable proportion of the total run time, eliminating redundant COR, UOI, and ROR mutants only yields a modest decrease in total run time for *math* and *jaxen*. Figure 4 visualizes the distribution of the ratio of analyzed mutants and total mutation analysis run time for the developer-written and generated test suites.

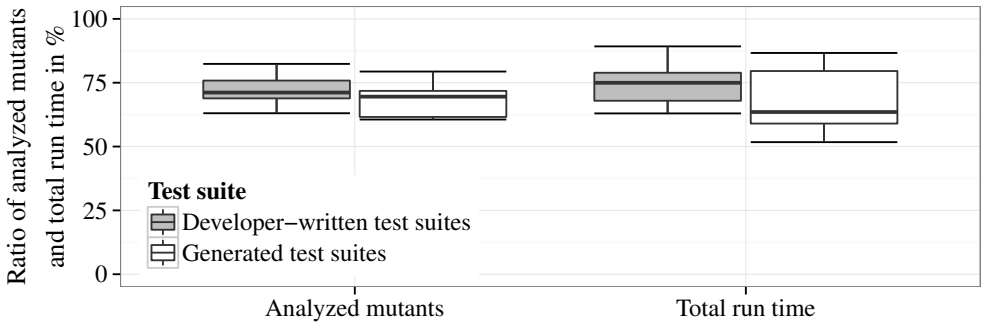

Figure 4. Ratio of analyzed mutants and total mutation analysis run time in %. Baseline is the set of all mutants M_{all}, which includes the subsumed COR, UOI, and ROR mutants.

Table VII. Difference between the mutation scores when using the set of all mutants M_{all} and the reduced set of mutants M_{red}. Mutation scores calculated with respect to the number of covered mutants. T_{dev} denotes the developer-written test suites and T_{gen} the generated test suites. The overall row is calculated by considering all covered and detected mutants across all 10 programs. The differences between the mutation scores are statistically significant at the 1% level for T_{dev} and T_{gen} (Wilcoxon signed-rank test).

	T_{dev}		T_{gen}	
	M_{all}	M_{red}	M_{all}	M_{red}
itext	0.278	0.250 (-10%)	0.424	0.390 (-8.1%)
trove	0.657	0.580 (-12%)	0.688	0.618 (-10%)
chart	0.439	0.363 (-17%)	0.439	0.362 (-18%)
math	0.831	0.816 (-1.8%)	0.525	0.482 (-8.2%)
time	0.892	0.876 (-1.9%)	0.644	0.623 (-3.3%)
lang	0.790	0.740 (-6.3%)	0.560	0.509 (-9.0%)
jdom	0.828	0.802 (-3.1%)	0.618	0.563 (-8.9%)
jaxen	0.556	0.471 (-15%)	0.643	0.461 (-28%)
io	0.783	0.776 (-0.8%)	0.486	0.403 (-17%)
num4j	0.686	0.668 (-2.7%)	0.636	0.607 (-4.6%)
overall	0.679	0.644 (-5.2%)	0.547	0.485 (-11%)

3.4. Effect of redundant mutants on the mutation score

This section answers the third research question and shows how the elimination of redundant mutants affects the mutation score with respect to the number of covered mutants — that is, the number of detected mutants divided by the number of covered mutants. Table VII gives the mutation scores for the test suites T_{dev} and T_{gen} when using the set of all mutants M_{all} and the reduced set of mutants M_{red}. When applying the reduced set of mutants, the mutation score decreases for all test suites with an overall decrease of 5% for T_{dev} and 11% for T_{gen}.

Unless redundant mutants are removed, the mutation score is overestimated for all test suites investigated in this empirical study. This means that redundant mutants cause the mutation score to less accurately reflect the test suites' effectiveness and the elimination of redundant mutants therefore improves the expressiveness of the mutation score. An accurate mutation score is of particular importance if it is used as an absolute value, for instance, to drive test data generation [21].

Table VIII. Comparison of mutation coverage with statement and branch coverage. Statement coverage and branch coverage as reported by Cobertura [17]. Mutation coverage is given for the reduced set of mutants M_{red} (values in parentheses give mutation coverage ratios for the set of all mutants M_{all}).

(a) Coverage ratios for developer-written test suites T_{dev}.

	Mutation Coverage		**Statement Coverage**	**Branch Coverage**
itext	0.15	(0.16)	0.20	0.11
trove	0.08	(0.08)	0.07	0.06
chart	0.53	(0.55)	0.57	0.46
math	0.90	(0.91)	0.88	0.85
time	0.82	(0.85)	0.90	0.80
lang	0.96	(0.97)	0.93	0.90
jdom	0.95	(0.96)	0.95	0.94
jaxen	0.66	(0.65)	0.78	0.55
io	0.58	(0.52)	0.39	0.29
num4j	0.96	(0.97)	0.97	0.96

(b) Coverage ratios for generated test suites T_{gen}.

	Mutation Coverage		**Statement Coverage**	**Branch Coverage**
itext	0.60	(0.60)	0.57	0.46
trove	0.71	(0.73)	0.77	0.67
chart	0.31	(0.33)	0.37	0.28
math	0.65	(0.67)	0.68	0.62
time	0.75	(0.78)	0.75	0.64
lang	0.84	(0.88)	0.77	0.71
jdom	0.46	(0.52)	0.64	0.47
jaxen	0.67	(0.73)	0.83	0.57
io	0.68	(0.70)	0.57	0.54
num4j	0.69	(0.75)	0.65	0.69

3.5. Effect of redundant mutants on the mutation coverage ratio

Answering the fourth research question, this section investigates the effect of redundant mutants on the mutation coverage ratio. It also measures the correlation between mutation coverage and two code coverage criteria, namely statement and branch coverage. Additionally, this section measures and discusses the correlation between the mutation coverage ratio and the mutation score.

Table VIII provides the mutation coverage ratios as well as the ratios for statement and branch coverage for the developer-written and generated test suites. Figure 5 and 6, furthermore, illustrate the correlation between mutation coverage and the two code coverage criteria. For the investigated test suites, the correlation between mutation coverage and both coverage criteria remains very strong after eliminating redundant mutants. This observation suggests that mutation coverage can be used as an alternative adequacy criterion for measuring code coverage.

Besides measuring the correlation, it is also important to consider the run time necessary to measure the coverage ratios. In contrast to the mutation score, mutation and code coverage ratios can be ascertained with only one execution of the test suite using an instrumented version of the original program. Using Major to measure mutation coverage and Cobertura to measure code coverage, mutation coverage could be calculated 45% faster on average in the experiments. Given the strong

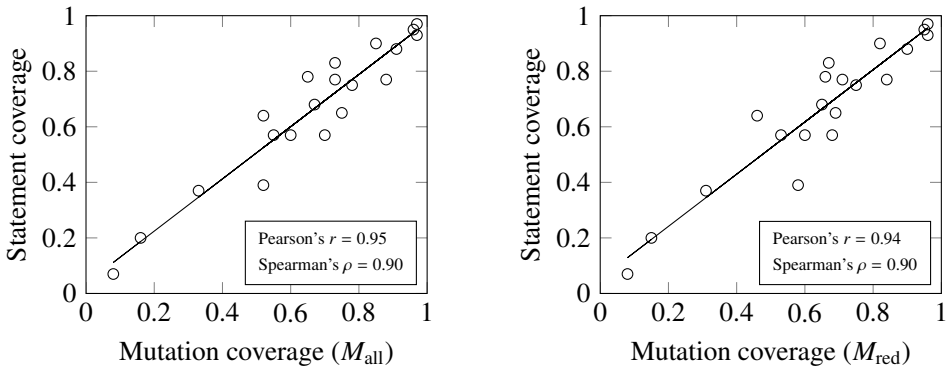

Figure 5. Correlation between mutation coverage and statement coverage.

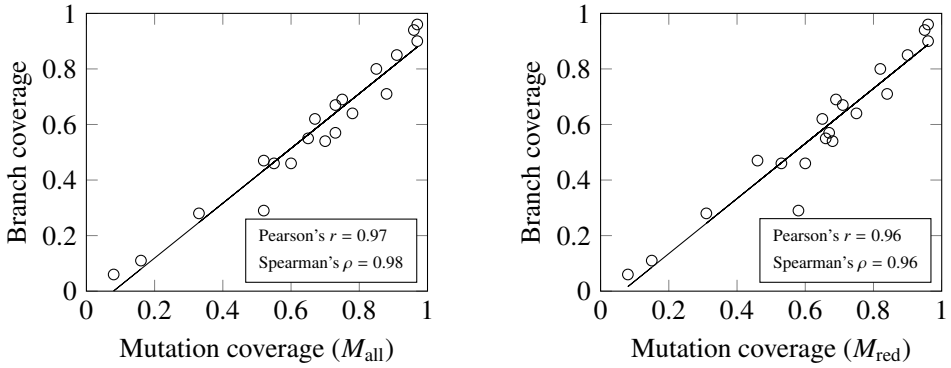

Figure 6. Correlation between mutation coverage and branch coverage.

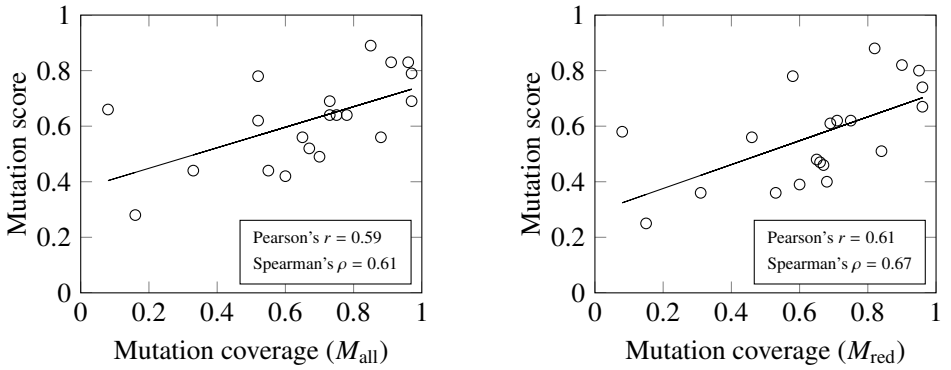

Figure 7. Correlation between mutation coverage and mutation score using all test suites.

correlation and low costs of determining mutation coverage, one could exclusively employ mutation analysis to assess the overall effectiveness (i.e., coverage and test oracle strength) of a test suite. The costs for measuring coverage may, however, depend on the code instrumentation and the employed tool, and therefore additional studies are necessary to confirm the run-time observations.

Furthermore, Figure 7 shows that there is only a moderate correlation between mutation coverage and the mutation score for the investigated test suites, suggesting that a high degree of mutation coverage is not necessarily a good indicator of test oracle strength.

3.6. Threats to Validity

This section examines the threats to validity for the conducted empirical study. The chosen set of sufficient mutation operators could be a threat to internal validity. Different or additional operators may affect both the number and the ratio of redundant mutants. However, the mutation operators employed in the presented study are frequently used in the literature and provide comparable results (e.g., [5, 6, 22]). In addition to the mutation operators, the function chosen to determine the timeout for the mutants could be another threat to internal validity. A timeout factor that is too small would result in a noticeable number of false-positives and introduce a bias towards the run-time improvements. This threat was controlled by manually analyzing samples of mutants that ran into a timeout and also by comparing the numbers of timeouts for different timeout factors. Mutants that were detected due to a timeout exhibited syntactic changes that caused the detecting test to run exceptionally long or infinitely. Furthermore, the empirical study reports on two alternative efficiency metrics (decrease in the number of generated mutants and decrease in the number of analyzed mutants) — all three metrics led to the same overall conclusion.

The representativeness of the selected subject programs might be a potential threat to external validity. To address this issue, the empirical study considered programs that vary in terms of size and operation purpose. Additionally, the preliminary study [23] was extended to verify that the results generalize to a certain extent. For this purpose, more subject programs were added and the entire study was repeated with a generated test suite for each program.

Defects in the employed mutation tool could be another threat to validity. This threat was controlled by employing several example programs and manually analyzing the resulting mutants and data. Besides, the Major mutation framework was used in previous studies without encountering any problems. Considering the results about the accuracy and expressiveness of the mutation score, the determined mutation score only adequately reflects the effectiveness of a test suite if the injected mutants are representative for real faults. This threat to construct validity is, however, not unique to this study but rather applies to all studies using mutation analysis. The assumption that mutants are a valid substitute for real faults is widely accepted due to existing empirical evidence [24, 25, 26]. Provided that this assumption holds, the exclusion of redundant mutants not only improves the efficiency of mutation analysis but also the expressiveness of the mutation score, which is commonly used as a proxy for test effectiveness.

4. RELATED WORK

Several cost-reduction approaches for mutation analysis have been suggested in the literature (cf. [6, 27]). This section discusses the closest related work that focuses on the reduction of mutants.

Generally, mutant reduction techniques reduce the quantity of mutants either by decreasing the number of mutation operators or by sampling the set of generated mutants [4, 5, 28]. The reduced set of mutants can be applied more efficiently but incurs a loss of information. Additionally, existing approaches view the mutation operators, in their originally defined form, as atomic. Hence, there are still redundancies within the reduced set of mutants. In contrast to prior studies, this paper focuses on the reduction of mutants without any loss of information — only subsumed mutants are excluded.

The set of applicable mutation operators depends on the programming language. While this paper focuses on the mutation analysis of unit tests for Java programs, mutation operators have been suggested for many other languages, such as Fortran or C (e.g., [7, 29]). Moreover, mutation analysis is also applicable in the context of integration or system testing (e.g., [6, 18, 30]).

Considering run-time improvements for a given set of generated mutants, exploiting mutation coverage information is the most common optimization to avoid unnecessary mutant executions (e.g., [10, 16, 31]). Mutation coverage is used as the baseline for efficiency evaluations in this paper to provide a fair comparison. Just et al. investigated efficiency improvements due to test suite prioritization in a related study, which primarily focused on test suite characteristics and different test suite prioritization approaches to improve the efficiency of mutation analysis [20]. In contrast, this paper investigates the effect of eliminating redundant mutants on the efficiency and accuracy of mutation analysis.

With regard to redundant mutants, Kaminski et al. investigated the ROR mutation operator and showed that a subset of all valid replacements is sufficient for this operator [13]. They additionally claimed, without further investigation, that this reduction would improve efficiency. In contrast, this paper confirms that a sufficient set of non-redundant mutations exists for the ROR mutation operator, and also provides sufficient sets of non-redundant mutations for the COR and UOI mutation operators. Furthermore, this paper provides an empirical study that investigates the actual efficiency improvements due to the elimination of redundant mutants. In connection with testing conditional and relational operators, Tai developed a theory for testing the predicates in conditional statements [32].

Considering mutant reduction techniques, higher order mutation aims at generating fewer yet subtle mutants [12]. A mutant created by combining two simple (first-order) mutants is referred to as second order mutant. 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 detect than the first-order mutants of which they were created. The computational costs for higher order mutation are significantly greater because of the combinatorial explosion, but search-based approaches seem to be a feasible solution to this problem [12].

Besides redundant mutants, the equivalent mutant problem is another crucial challenge in mutation testing. Equivalent mutants are harmful to the run time 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: (1) Approaches that avoid generating equivalent mutants during the mutant generation process (e.g., [33, 34]) and (2) approaches that focus on the detection of equivalent mutants (e.g., [35, 36, 22]).

5. CONCLUSIONS AND FUTURE WORK

This paper investigates how redundant mutants affect the efficiency and accuracy of mutation analysis. Focusing on three well-known mutation operators, namely the conditional operator replacement (COR), unary operator insertion (UOI), and relational operator replacement (ROR) mutation operators, it makes the following contributions.

This paper first develops a subsumption hierarchy and provides a sufficient set of non-redundant mutations for the COR, UOI, and ROR mutation operators. The paper shows that 4 out of 10 COR mutations are sufficient and that the COR mutation operator subsumes the UOI mutation operator for boolean expressions. The paper also confirms prior results and shows that 3 out of 7 ROR mutations are sufficient.

The empirical study in this paper demonstrates how redundant mutants affect the efficiency and accuracy of mutation analysis. Besides showing how prevalent COR, UOI, and ROR mutants are for real-world programs, the study reveals that eliminating redundant mutants decreases the total mutation analysis run time by more than 20%. Moreover, the study shows that the inclusion of redundant mutants misleadingly overestimates the mutation score. Comparing mutation coverage with statement coverage, branch coverage, and the mutation score, the study reveals that mutation coverage has a very strong correlation with statement and branch coverage but only a moderate correlation with the mutation score.

Given the results reported in the empirical study, areas for future work include the determination of sufficient sets of non-redundant mutations for other mutation operators, such as the arithmetic operator replacement (AOR). The notion of determining a minimal distance between a mutation and its original version should be transferable to other mutation operators as well. In addition, this paper investigates intra-operator redundancies but mutants derived from different mutation operators might also exhibit redundancies. Therefore, the investigation of inter-operator redundancies is another area for future work.

The Major mutation framework and the experimental data is publicly available at:

`http://mutation-testing.org`

6. ACKNOWLEDGEMENTS

The authors would like to thank Gregory M. Kapfhammer, Suzanne Millstein, Jonathan Burke, and the anonymous reviewers for the helpful and detailed comments on earlier versions of this manuscript. This work was partially supported by LGFG Baden-Württemberg and the German Research Foundation (DFG).

REFERENCES

1. Budd TA. Mutation Analysis of Program Test Data. PhD Thesis, Yale University 1980.
2. DeMillo RA, Lipton RJ, Sayward FG. Hints on test data selection: Help for the practicing programmer. *IEEE Computer* 1978; **11**(4):34–41.
3. Zhu H, Hall PAV, May JHR. Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)* 1997; **29**(4):366–427.
4. Offutt AJ, Lee A, Rothermel G, Untch RH, Zapf C. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 1996; **5**(2):99–118.
5. Siami Namin A, Andrews JH, Murdoch DJ. Sufficient mutation operators for measuring test effectiveness. *Proceedings of the International Conference on Software Engineering (ICSE)*, 2008; 351–360.
6. Jia Y, Harman M. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering (TSE)* 2011; **37**(5):649–678.

7. King KN, Offutt AJ. A Fortran language system for mutation-based software testing. *Software: Practice and Experience* 1991; **21**(7):685–718.
8. Ma YS, Offutt J, Kwon YR. MuJava: A mutation system for Java. *Proceedings of the International Conference on Software Engineering (ICSE)*, 2006; 827–830.
9. Kim S, Clark JA, McDermid JA. Class mutation: Mutation testing for object-oriented programs. *Proceedings of the Net. Object Days Conference on Object-Oriented Software Systems*, 2000; 9–12.
10. Just R, Schweiggert F, Kapfhammer GM. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2011; 612–615.
11. Just R, Ernst MD, Fraser G. Efficient mutation analysis by propagating and partitioning infected execution states. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2014; 315–326.
12. Jia Y, Harman M. Higher order mutation testing. *Information and Software Technology (IST)* 2009; **51**:1379–1393.
13. Kaminski G, Ammann P, Offutt J. Improving logic-based testing. *Journal of Systems and Software (JSS)* 2013; **86**(8):2002–2012.
14. Just R, Kapfhammer GM, Schweiggert F. Using conditional mutation to increase the efficiency of mutation analysis. *Proceedings of the International Workshop on Automation of Software Test (AST)*, 2011; 50–56.
15. SLOCCount. The official web site of David A. Wheeler's SLOCCount. `http://www.dwheeler.com/sloccount` Accessed June 2013.
16. Fraser G, Arcuri A. Evosuite: automatic test suite generation for object-oriented software. *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2011; 416–419.
17. Cobertura. The official web site of the Cobertura project. `http://cobertura.sourceforge.net` Accessed August 2012.
18. Just R, Schweiggert F. Automating unit and integration testing with partial oracles. *Software Quality Journal (SQJ)* 2011; **19**(4):753–769.
19. Schuler D, Zeller A. Checked coverage: an indicator for oracle quality. *Software Testing, Verification and Reliability (JSTVR)* 2013; **23**(7):531–551.
20. Just R, Kapfhammer GM, Schweiggert F. Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis. *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, 2012; 11–20.
21. Fraser G, Zeller A. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering (TSE)* 2012; **28**(2):278–292.
22. Schuler D, Zeller A. (Un-)covering equivalent mutants. *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, 2010; 45–54.
23. Just R, Kapfhammer GM, Schweiggert F. Do redundant mutants affect the effectiveness and efficiency of mutation analysis? *Proceedings of the International Workshop on Mutation Analysis (Mutation)*, 2012; 720–725.
24. Andrews JH, Briand LC, Labiche Y, Namin AS. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering (TSE)* 2006; **32**(8):608–624.
25. Andrews JH, Briand LC, Labiche Y. Is mutation an appropriate tool for testing experiments? *Proceedings of the International Conference on Software Engineering (ICSE)*, 2005; 402–411.
26. Just R, Jalali D, Inozemtseva L, Ernst MD, Holmes R, Fraser G. Are mutants a valid substitute for real faults in software testing? *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*, 2014; 654–665.
27. Offutt AJ, Untch RH. Mutation 2000: Uniting the orthogonal. *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, 2001; 45–55.
28. Wong WE, Mathur AP. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software (JSS)* 1995; **31**(3):185–196.
29. Agrawal H, Demillo R, Hathaway R, Hsu W, Hsu W, Krauser E, Martin RJ, Mathur A, Spafford E. Design of mutant operators for the C programming language. *Technical Report SERC-TR-41-P*, Software Engineering Research Center, Purdue University 1989.
30. Delamaro ME, Maidonado J, Mathur AP. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering (TSE)* 2001; **27**(3):228–247.
31. Schuler D, Zeller A. Javalanche: Efficient mutation testing for Java. *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2009; 297–298.
32. Tai KC. Theory of fault-based predicate testing for computer programs. *IEEE Transactions on Software Engineering (TSE)* 1996; **22**(8):552–562.

33. Offutt AJ. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 1992; **1**(1):5–20.
34. Yao X, Harman M, Jia Y. A study of equivalent and stubborn mutation operators using human analysis of equivalence. *Proceedings of the International Conference on Software Engineering (ICSE)*, 2014; 919–930.
35. Offutt AJ, Craft WM. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification and Reliability (JSTVR)* 1994; **4**:131–154.
36. Hierons R, Harman M, Danicic S. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability (JSTVR)* 1999; **9**:233–262.