SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 0000; **00**:1–34
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Assessment of C++ Object-Oriented Mutation Operators: A Selective Mutation Approach

Pedro Delgado-Pérez[1]*, Sergio Segura[2], Inmaculada Medina-Bulo[1]

[1] *University of Cádiz. Escuela Superior de Ingeniería, Spain.*
[2] *University of Sevilla. Escuela Superior de Ingeniería, Spain.*

SUMMARY

Mutation testing is an effective but costly testing technique. Several studies have observed that some mutants can be redundant and therefore removed without affecting its effectiveness. Similarly, some mutants may be more effective than others in guiding the tester on the creation of high-quality test cases. Based on these findings, we present an assessment of C++ class mutation operators by classifying them into two rankings: the first ranking sorts the operators based on their degree of redundancy, and the second regarding the quality of the tests they help to design. Both rankings are used in a selective mutation study analysing the trade-off between the reduction achieved and the effectiveness when using a subset of mutants. Experimental results consistently show that leveraging the operators at the top of the two rankings, which are different, lead to a significant reduction in the number of mutants with a minimum loss of effectiveness.

Received . . .

KEY WORDS: Mutation testing; class mutation operators; C++; selective mutation; quality of mutation operators

1. INTRODUCTION

Mutation testing [1] is a common fault-based testing technique to assess and enhance the fault-detection capability of test suites. This technique creates several modified versions (*mutants*) of the original program under test (PUT), which differ in a simple syntactic change injected by a *mutation operator*. Each mutant is then executed on the same test suite as the original program. If a test case distinguishes the original program from a mutant we say that the mutant has been *killed* and the test case has proved to be effective at finding faults in the program. Otherwise, the mutant remains *alive*. Mutants that keep the same program functionality and thus cannot be detected are referred to as *equivalent*. Mutation testing has traditionally been applied to procedural programs written in languages like Fortran [2] or C [3], using *traditional* or *standard mutation operators*. However, the increasing presence of object-oriented programs in industrial systems has progressively drawn

*Correspondence to: University of Cádiz. Escuela Superior de Ingeniería, Spain. E-mail: pedro.delgado@uca.es

the attention of mutation researchers toward other languages such as Java [4], C# [5] or C++ [6]. Contributions in this context mainly focus on the development of new tools and mutation operators (named *class* or *object-oriented* operators) specifically designed to create faults involving typical object-oriented features like inheritance or polymorphism.

Mutation testing is mainly used for two purposes: evaluate and refine test suites. During *Test Suite Evaluation* (TSE), mutation testing is used to assess how effective a test suite is at detecting faults in the PUT. The test suite effectiveness is measured using the *mutation adequacy score*, the ratio of killed mutants to the total of mutants derived from the PUT (excluding equivalent mutants). A test suite is said to be *adequate* if it achieves a mutation score of 100%, and *minimal* when it contains the minimum number of test cases that are essential to kill all the mutants. A mutant remaining alive uncovers a weakness in the test suite. During *Test Suite Refinement* (TSR), mutation testing guides the tester on the improvement of the suite by designing new test cases that kill the surviving mutants.

Mutation testing also suffers from several drawbacks. A key limitation of the technique is its high cost due to the large number of mutants that can be generated even in the case of small-sized programs. For instance, applying mutation testing to a numerical program of 78 lines of code written in Fortran yielded 7,435 mutants using traditional operators [7]. Another limitation is related to detection of equivalent mutants, which is a time-consuming, error-prone and manual task. In theory, equivalent mutants should be excluded from the set of mutants, but in practice this is not always possible since program equivalence is undecidable [8]. Consequently, even when the number of mutants is manageable, the effort required to identify equivalent mutants could make the application of the technique not affordable.

Multiple techniques have been propose to reduce the cost of mutation testing including high-order mutation [9] and mutant clustering [10]. *Selective mutation* is a well-known cost reduction technique to exclude some of the mutants without significant loss of effectiveness. We can distinguish two main selective approaches: *operator-based* and *mutant-based* selection [11]. On the one hand, operator-based mutant selection [12, 13] works under the assumption that not all mutation operators are equally effective and that there should be a *sufficient set of operators* that allows us to accurately predict the overall mutation score. The rationale behind operator-based selection is that some mutation operators are redundant and they can be therefore discarded. Intuitively, an operator is *redundant* if it produces mutants that are always killed by the test cases that kill mutants from other operators. In the mutation literature, an operator that only generates redundant mutants is said to be subsumed by the rest of mutation operators in the set [12]. As a notable example, Offutt et al. [7] found that using 16 mutation operators for Fortran is almost as strong as using the whole set of 22 operators achieving a reduction over 60% in the number of mutants. On the other hand, mutant-based selection [14, 15] also presumes the existence of redundancy but at the level of mutants instead of operators. In this way, in random mutant selection only a subset of the mutants from the full set of operators is randomly analysed, whereas the rest of the mutants are discarded. Although random selection has drawn less attention than operator-based selection, Zhang et al. [11] recently produced evidence that the former can be as effective as the latter with the same number of mutants. Most relevant studies on selective strategies have been reported for traditional operators in languages like C [11, 13, 16] or Fortran [7, 12, 17]. Some studies have addressed object-oriented languages like C# [18] or Java [19], where operators at the class level were tackled along with traditional operators.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

In a recent study, Estero-Botaro et al. [20] compared the effectiveness of mutation operators from a different perspective. In particular, the authors noticed that not all operators are equally effective at inducing the creation of *high-quality test cases*; these test cases detect non trivial faults which are not easy to find with a straightforward test case. Based on this premise, the authors proposed a metric to evaluate the quality of mutation operators according to their ability to generate hard-to-kill mutants.

Problem. Contrary to traditional mutation operators and well studied languages such as Java, the applicability of mutation testing to C++ object-oriented features is a research topic under development. In particular, it remains unclear whether C++ class mutation operators exhibit any *degree of redundancy* (percentage of redundant mutants generated by each operator) or to what extent they contribute to create high-quality test cases. As a result, it is unknown what are the most promising operators and the loss of accuracy we must concede when using them in a selective mutation strategy. Overall, the lack of experimental results on C++ class mutation operators hinders their applicability and discourage researchers and practitioners from using them.

Contribution. In this paper, we present an assessment of mutation operators at the class level for C++. We conjecture that the value of each mutation operator differs depending whether the test suite is being evaluated (TSE) or refined (TSR). During TSE, testers aim to reduce the number of redundant mutants since they add no value in the process. During TSR, testers wish to favour those operators that contribute to create high-quality test cases able to uncover hard-to-detect faults. Based on this idea, we rank mutation operators regarding their influence during TSE and TSR respectively. The first ranking sorts the operators based on their degree of redundancy; the second ranking sorts the operators regarding their potential to contribute on the creation of high-quality test cases (based on Estero-Botaro's metric [20]). These two rankings are used as the basis for a selective mutation study showing the trade-off between removing mutants and the loss in the effectiveness of the technique. We apply two selective strategies to this end: an operator-based selection and a rank-based mutant selection (i.e., favouring the selection of mutants from the top ranked operators). These outcomes are based on the results of several experiments with six open-source applications. The following are the main contributions of this paper:

- **A double assessment of C++ class mutation operators based on their influence during TSE and TSR respectively**. To the best of our knowledge, this is the first work assessing mutation operators from this double perspective.
- **A selective mutation study for TSE using the ranking based on mutant redundancy**. Among other findings, results show that applying the top 6 operators (out of 24) leads to a reduction over 31% in the number of mutants with a mutation score of 97.22%. With the same size of mutants, a rank-based mutant selection obtains 98.87% of adequacy.
- **A selective mutation study for TSR using the ranking based on test quality**. Among other results, experiments reveal that applying the top 7 operators leads to a reduction of almost 40% in the number of the mutants assuming a loss of only 13% in the number of test cases in an adequate and minimal test suite. With the same number of mutants, the percentage of test cases lost using rank-based mutant selection is under 6%.
- **A comparison between the two rankings**, where it is revealed that they are quite interrelated, except for a few operators with a substantially different position in both classifications. This fact supports the evaluation of mutation operators from a double perspective.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

- **A comparison between operator-based and mutant-based selection**, where experiments show that using a subset of mutants from all mutation operators is a preferable option in the case of class-level mutation testing (in the case of the highest reduction assessed, with a difference in the average over 5% and 10% in the evaluations for TSE and TSR respectively).
- **A comparison between rank-based and random mutant selection**, where the results of the rank-based strategy show that favouring the selection of mutants from the best-valued operators in both rankings offers a better outcome than the random selection of mutants overall (in the case of the highest reduction assessed, with a difference in the average of 0.3% and 1.85% in the evaluations for TSE and TSR respectively).

The remainder of this paper is structured as follows. Section 2 exposes the class mutation operators for C++ and the classification of mutants through the execution matrix. Section 3 and Section 4 describe the experiments performed for TSE and TSR respectively, and show the reported results. Section 5 discusses the empirical results and threats to validity. Section 6 studies related work in the scope of this paper, and last section presents conclusion and future work about the applied approach.

2. BACKGROUND

2.1. *Mutation operators for C++ object-oriented programs*

The study of mutation testing at the class level began in 1999 with the definition of the first class mutation operators by Kim et al. [21]. These mutation operators have been subject of study in recent years [18, 22, 23, 24]. Class-level operators are known to produce fewer mutants than traditional operators, and they appear with varying frequency depending on the features of the PUT [24]. In addition, equivalence is even a more pronounced issue when applying this kind of operators [22]. Still, operators at the class level deserve special attention because they are useful to test structures related to object-oriented features, which are not targeted by traditional operators.

In a previous work, some of the authors proposed a collection of class mutation operators for C++ [6]. These operators are similar to the ones defined for other object-oriented languages, such as Java [4] and C# [5], but taking into account specific features of the language (e.g., default parameters in a method). Moreover, new operators were defined regarding some characteristics not explored up to now, such as multiple inheritance or the existence of destructors.

Table I shows the C++ class operators under study classified by operator *groups* or *blocks*. Each block includes those operators addressing similar syntactic elements [4, 6]. These mutation operators have been implemented in the mutation system *MuCPP* [25]. We should note that *MuCPP* has been optimized towards increasing the percentage of *valid mutants* (complying with the grammar rules of the language) over the total number of mutants. Moreover, the system avoids creating some equivalent and trivial mutants (i.e., killed by every test case exercising the mutation) in order to enhance the operator effectiveness. This reduction is achieved through different heuristics, such as the ones proposed by Lee et al. [23].

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table I. C++ class-level mutation operators under study.

Block	Oper.	Description
Inheritance	**IHD**	Hiding variable deletion
	IHI	Hiding variable insertion
	ISD	Base keyword deletion
	ISI	Base keyword insertion
	IOD	Overriding method deletion
	IOP	Overriding method calling position change
	IOR	Overriding method rename
	IPC	Explicit call of a parent's constructor deletion
	IMR	Multiple inheritance replacement
Polymorphism and dynamic binding	**PVI**	*virtual* modifier insertion
	PCD	Type cast operator deletion
	PCI	Type cast operator insertion
	PCC	Cast type change
	PMD	Member variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PNC	New method call with child class type
Method overloading	**OMD**	Overloading method deletion
	OMR	Overloading method contents replace
	OAN	Argument number change
	OAO	Argument order change
Exception handling	**EHC**	Exception handling change
	EHR	Exception handler removal
Object and member replacement	**MCO**	Member call from another object
	MCI	Member call from another inherited class
Miscellany	**CTD**	*this* keyword deletion
	CTI	*this* keyword insertion
	CID	Member variable initialization deletion
	CDC	Default constructor creation
	CDD	Destructor method deletion
	CCA	Copy constructor and assignment operator overloading deletion

2.2. *Execution matrix*

An *execution matrix* contains the whole information about mutant execution, being useful for classifying mutants according to the values in the matrix [20]. We will resort to execution matrices throughout the paper to illustrate examples about the used metrics. Being M the set of mutants and T the set of test cases, the execution matrix with size $|M| \times |T|$ stores the result of running each mutant against each test case. That result depends on the behaviour of the mutant when compared with the original program. A mutant x killed by a test case y is represented with the value 1 in the intersection of the row x and the column y. On the contrary, the value 0 denotes that the mutation was not revealed by that test case.

A mutant, represented by a row in the execution matrix, is said to be:

- **Alive** when the row is filled with the value 0.
- **Dead** when there is at least one entry with the value 1 in the row.

Furthermore, Estero-Botaro et al. [26] defined two more specific terms to classify mutants:

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Operator	Mutant	Test cases					
		sp	$test_1$	$test_2$	$test_3$	$test_4$	$test_5$
o_1	m_1		1	0	0	0	0
	m_2		0	0	1	0	0
o_2	m_3		0	0	1	0	0
	m_4		0	0	0	1	0
o_3	m_5		0	0	0	0	1
	m_6		1	0	0	1	1
o_4	m_7		1	0	1	0	1
	m_8		0	0	1	1	1
o_5	m_9		0	1	0	0	0
	m_{10}		0	0	0	1	0

Figure 1. Example of matrix execution with size 10×5.

- A **resistant mutant** is killed by a single test case, and is identified as a row filled with the value 0 except for one entry with the value 1. In Figure 1, the mutant 1 from the operator 1 (m_1) is a resistant mutant.
- A **resistant hard to kill mutant** is killed by a single test case which only kills that mutant. In the execution matrix, it is identified as a row with a single entry y with the value 1 (just as a resistant mutant), where the rest of the entries in the column y are filled with the value 0. In Figure 1, m_1 is resistant but not resistant hard to kill because the $test_1$, which kills that mutant, also kills the mutants m_6 and m_7. The mutant 9 generated by the operator 5 does represent a resistant hard to kill mutant.

As mentioned in the introduction, a test suite is adequate when it detects all non-equivalent mutants. The execution matrix can also be useful to ascertain some properties of a test suite:

- **Non-redundant test suite**: when none of the test cases in an adequate test suite can be removed without losing the adequacy of the test suite. The test suite in Figure 1 is adequate and non-redundant, as we cannot discard any of the test cases maintaining the same mutation score.
- **Minimal test suite**: when a non-redundant test suite is of the minimum size, that is, there are no other non-redundant test suites of smaller size. The test suite in Figure 1 is also a minimal test suite.

We have to note that our concepts of non-redundant and minimal test suite are called as minimal and minimum test suite respectively by Amman et al. [27]. Therefore, in our work we focus on minimal test suites, which are called minimum test suites by the aforementioned authors.

3. ASSESSMENT BASED ON MUTANT REDUNDANCY

In this section, we assess the value of each mutation operator for TSE. To that end, we first present the addressed research questions followed by the evaluation metric, subject case studies and experiments performed.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

3.1. Research questions

The goal of this section is to answer the following research questions:

- **RQ1: What is the degree of redundancy of each mutation operator?** We aim to rank mutation operators based on the number of their mutants which are redundant regarding some of the mutants generated by the rest of operators in the set.
- **RQ2: Is a subset of mutants with low degree of redundancy sufficient for TSE?** We intend to know the loss of mutation score when selecting (1) a subset of the top ranked operators based on the degree of redundancy and (2) a subset of mutants in which the selection of mutants from the top ranked operators is favoured. This fact would allow us to analyse the trade-off between the reduction in the number of mutants and the effectiveness of the technique when evaluating a test suite.

3.2. Evaluation metric

We propose to measure the degree of redundancy of a mutation operator as the number of redundant mutants generated by the operator with respect to the mutants generated by the rest of operators. Roughly speaking, an operator is redundant if all its mutants are killed by test cases that are necessary to kill mutants from other operators. Formally, we define the metric *operator redundancy* to measure the degree of redundancy of a mutation operator o as follows:

$$R_o(T_{MO}) = \begin{cases} \frac{\left|D(T_{MO\setminus o})\right|}{|D(T_{MO})|} \times 100, & D_o \neq \emptyset \\ 100, & D_o = \emptyset \end{cases} \quad (1)$$

Where:

- D_o is the set of dead mutants from operator o.
- MO is the set of mutation operators.
- T_{MO} is an adequate test suite for the set of mutants in MO .
- $D(T_{MO})$ is the set of dead mutants with T_{MO}.
- $D(T_{MO\setminus o})$ is the set of dead mutants when using an adequate and minimal test suite derived from T_{MO} without considering the mutants from operator o (as we will discuss later in this paper, minimality is desirable to exclude test cases that may cause deviations in the values).

Equation 1 measures the operator redundancy (R_o) as the percentage of mutants killed by an adequate test suite for all the mutants excluding the mutation operator under evaluation. The lower the value of R_o, the less number of redundant mutants and therefore the more valued is that mutation operator. The value of R_o can range from 100 to 0:

- $R_o = 100$: all the mutants from the mutation operator o are redundant. This happens when the test cases that kill the mutants generated by o are still necessary to kill the mutants from other operators, i.e., $|D(T_{MO})| = \left|D(T_{MO\setminus o})\right|$. Another possibility is that all the mutants are equivalent ($D_o = \emptyset$), as stated in Equation 1.
- $R_o = 0$: the analysed mutation operator is the only operator in the set generating non-equivalent mutants (i.e., $T_{MO\setminus o} = \emptyset$).

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Operator	Mutant	Test cases		
		$test_1$	$test_2$	$test_3$
o_1	m_1	0	1	0
	m_2	1	0	0
o_2	m_3	1	1	1
	m_4	0	1	1
o_3	m_5	0	0	1
	m_6	1	0	1

Figure 2. Execution matrix to illustrate the metric *operator redundancy*.

As an example, consider the execution matrix in Figure 2. The set $\{test_1, test_2, test_3\}$ is an adequate and non-redundant test suite for the set of operators $\{o_1, o_2, o_3\}$ (T_{MO}) because all of those test cases are necessary to kill the mutants from those operators (in this case, it is also minimal). Then, we can compute the following adequate and minimal test suites when excluding an operator each time:

- $T_{MO\setminus o_1} = \{test_3\}$
- $T_{MO\setminus o_2} = \{test_1, test_2, test_3\}$
- $T_{MO\setminus o_3} = \{test_1, test_2\}$

The subset $\{test_1, test_2\}$ is an adequate and minimal test suite for $T_{MO\setminus o_3}$ as this subset kills all the mutants without considering o_3 (m_1-m_4). Once those adequate and minimal test suites are known, we can calculate the set of dead mutants associated with those test suites:

- $D(T_{MO\setminus o_1}) = \{m_3, m_4, m_5, m_6\}$
- $D(T_{MO\setminus o_2}) = \{m_1, m_2, m_3, m_4, m_5, m_6\}$
- $D(T_{MO\setminus o_3}) = \{m_1, m_2, m_3, m_4, m_6\}$

Finally, the value of the operator redundancy metric for these three operators can be calculated as follows ($D_o \neq \emptyset$ in all cases):

- $R_{o_1} = (4/6) \times 100 = 66.6$
- $R_{o_2} = (6/6) \times 100 = 100$
- $R_{o_3} = (5/6) \times 100 = 83.3$

Interpreting these results:

- The operator o_1 presents the lowest redundancy: only 66.6% of the mutants (4 out of 6) would be killed with an adequate test suite for the subset of operators $\{o_2, o_3\}$.
- The mutants from o_2 are redundant with regard to the mutants created by o_1 and o_3 ($R_{o_2} = 100$).
- The mutant 5 from o_3 is a non-redundant mutant as it would remain alive after using the subset $\{test_1, test_2\}$ ($R_{o_2} = 83.3$).

As a conclusion, a mutation operator with a low degree of redundancy increases the probability of losing effectiveness if mutants from that operator are discarded when following a selective mutation strategy. Therefore, the operators with the lowest R_o should be at the top of our ranking.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table II. Features of the case studies used in the experiments.

	Tcl	**Rpc**	**Dph**	**Txm**	**Kmy**	**Dom**	**Total**
Classes	9	13	13	20	17	11	83
Lines of code	3,228	2,194	3,667	2,620	13,709	2,117	27,535
Avg. Methods	21.1	11.2	16.4	15.6	35.6	23.6	20.6
Mutants	135	127	208	433	284	681	1,868
% Equivalent	14.8	31.5	33.2	21.0	30.6	34.7	29.1
% Undecided	0	0	4.8	7.4	1.4	1.5	3.0
$\|Original\ T\|$	17	26	61	57	241	46	-
$\|Adequate\ T\|$	24(3)	34(5)	70(5)	62(3)	248(10)	56(4)	-
$\|Minimal\ T\|$	15	15	22	15	36	25	-

3.3. Case studies and test suites

For the experiments, we applied mutation testing on six different real open-source applications and libraries, namely:

- *Matrix TCL Pro* (*Tcl*) [28]: library for performing matrix algebra calculations in C++ programs.
- *XmlRpc++* (*Rpc*) [29]: library implementing the XML-RPC protocol to incorporate client-server communication through HTTP support into other C++ programs.
- *Dolphin* (*Dph*) [30]: default navigational file manager used by desktop applications in KDE.
- *Tinyxml2* (*Txm*) [31]: lightweight and efficient XML parser that can be integrated into C++ applications.
- *KMyMoney* (*Kmy*) [32]: the personal finance manager by KDE.
- *QtDom* (*Dom*) [33]: Qt module that provides a C++ implementation of the DOM standard.

The test suites accompanying these programs were used and completed with additional test cases designed by hand until reaching an adequate test suite. This is a laborious task because a mutant may require complex scenarios involving different classes to be killed. In the process of generating new test cases, surviving mutants must be examined, taking often a long time to distinguish equivalent mutants from those whose semantic is very similar to the original program. There is an intrinsic bias in determining equivalence, as not always a tester can establish this state with high confidence. To minimize this threat in the calculations, we opted for classifying as *undecided* the mutants for which we were not sure whether they are equivalent or not, as Segura et al. [24] proposed in a previous study. In this regard, the concept of undecided mutant prevents skewing of results in the experiments.

For these experiments, we used the random adequate and minimal test suite generated by the exact algorithm that Estero-Botaro et al. [20] used in their study. Any metric is dependent on the test suite. Thus, we make use of minimal test suites because that property prevents the results from being distorted by unproductive test cases, as pointed by Estero-Botaro et al. [26].

Table II depicts several metrics about the aforementioned case studies: number of classes, lines of code, mean of methods in the analysed classes, the total of mutants, the percentage of equivalent and undecided mutants. The size of the original test suite, adequate test suite after adding new test cases (between parentheses, the number of test cases additionally modified) and minimal test suite are also shown.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table III. Mutants generated in each case study by operator (*M: mutants; D: dead; E: equivalent*).

Oper.	Tcl		Rpc		Dph		Txm		Kmy		Dom		Total		
	D	E	D	E	D	E	D	E	D	E	D	E	**M**	**D**	**E**
IHD	0	0	0	0	0	0	0	0	1	0	0	0	1	1	0
IHI	0	0	2	2	0	0	41	6	8	15	21	25	120	72	48
ISD	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0
ISI	0	0	2	1	9	2	0	0	0	3	1	1	19	12	7
IOD	0	0	1	2	15	3	24	1	1	0	28	2	79	69	8
IOP	0	0	0	0	0	0	8	0	0	0	0	2	10	8	2
IOR	0	0	0	15	3	27	10	1	0	0	0	1	57	13	44
IPC	0	0	1	0	2	3	0	0	12	6	8	0	32	23	9
PCI	0	0	2	1	0	0	138	20	14	1	293	155	659	447	177
PMD	0	0	0	0	0	0	0	3	0	1	0	4	8	0	8
PPD	0	0	0	1	0	0	5	2	4	14	2	12	42	11	29
PNC	0	0	0	0	0	0	0	0	0	0	2	0	2	2	0
OMD	38	8	9	1	2	1	23	14	9	4	16	6	131	97	34
OMR	33	1	10	0	5	1	0	0	32	0	16	0	98	96	2
OAN	0	0	0	0	0	0	0	0	3	4	0	0	7	3	4
MCO	3	0	38	10	68	7	18	1	76	7	36	7	285	239	32
MCI	0	0	0	0	0	0	13	26	0	0	0	0	39	13	26
EHC	0	0	1	1	0	0	0	0	1	5	0	0	8	2	6
CTD	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0
CTI	0	0	0	0	2	0	0	0	0	0	1	0	3	3	0
CID	38	2	14	3	23	18	24	10	26	22	6	9	196	131	64
CDC	0	0	2	0	0	0	3	0	4	1	0	0	10	9	1
CDD	0	2	2	3	0	2	3	3	2	2	0	4	23	7	16
CCA	3	7	2	0	0	5	0	4	0	2	4	8	37	9	26
Total	115	20	87	40	129	69	310	91	193	87	435	236	1,868	1,269	543

Finally, Table III shows a breakdown of the total number of mutants and their classification into dead (*D*) and equivalent (*E*) for each case study and mutation operator. In *Total*, the number of undecided mutants corresponds to the cases where the sum of dead and equivalent mutants is not equal to the number of mutants (*M*).

3.4. Experiment #1: Ranking mutation operators

In this experiment, we measure the operator redundancy as described in Section 3.2 for each of the studied mutation operators.

3.4.1. Setup We first determined an adequate and minimal test suite for the set of mutants (see Section 3.3). Then, the following process was performed for each mutation operator o generating at least one dead mutant:

1. Remove from the execution matrix the mutants generated by the mutation operator o ($MO \setminus o$).
2. Compute an adequate and minimal test suite for the remaining operators ($T_{MO \setminus o}$).
3. Include again in the execution matrix the mutants generated by the mutation operator o.
4. Remove from the execution matrix the columns of the test cases which were not in the computed adequate and minimal test suite.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

5. Calculate the operator redundancy of o ($R_o(T_{MO})$).

This procedure was carried out for each of the case studies and then a mean was calculated with the values obtained for each operator. Finally, a ranking was prepared in descending order of R_o.

3.4.2. Results The results of the operator redundancy metric of each operator and case study appear in Table IV. The mean calculated per operator is used to form the ranking, where *MCO* is the most valued operator on average. The standard deviation (*SD*) has also been included to show how much the metric varies among PUTs. The operators at the bottom of the table ($R_o = 100$) do not affect the TSE process when excluding one of them from the set of operators. The figures marked with '*' represent operators only producing equivalent mutants in that case study.

As illustrated, although ten operators have an operator redundancy of 100, the rest of operators show a redundancy degree between 89.67 and 99.92. The operator redundancy on average for all the mutation operators is high: 18 out of 24 mutation operators present a value over 99. This is explained by the fact that a test case usually reveals the mutations injected by different operators, so removing an operator does not always lead to a reduction in the number of test cases. These high values have also been observed in similar works when only one operator is removed [34].

The top 4 ranked operators are the ones producing the highest number of mutants (see Table III); it seems unlikely that removing a great quantity of mutants does not lead to a decrease in the number of necessary test cases. However, *IHI* is the fifth most prevalent operator and ranks at number 13 out of 24 operators. We run the Spearman's correlation test to know how the number of mutants influences this metric. The results in each of the programs range from -0.56 in *Rpc* to -0.73 in *Kmy* (95% confidence level except for *Tcl*). These results suggest that indeed there is an inverse correlation between the number of mutants generated by the operators and the value that the redundancy metric (TSE) assigns them, but the correlation is not very strong, which means that the operator redundancy does not depend only on the number of mutants generated by each operator. The top 5 operators are from different operator groups (see Table I). "Exception handling" is the only block not represented in that top 5. This fact suggests that the operators at the top of the ranking partially subsume the rest of operators in the same group. It also suggests that each operator block tackles different features, so it is less probable that operators from different groups are redundant among them.

3.5. Experiment #2: Selective mutation based on the ranking

This second experiment aims to leverage the ranking of mutation operators obtained in the previous experiment to undertake selective mutation. The goal is to observe the loss in the mutation adequacy score when some of the mutants are not included for TSE.

3.5.1. Setup In a first step, we grouped together the operators with a similar rate into five categories. We set the following ranges with a view to balance the number of operators in each category (see Table IV):

Category 1: $98 > R_o$

Category 2: $99 > R_o \geq 98$

Category 3: $99.5 > R_o \geq 99$

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table IV. Ranking of mutation operators based on mutant redundancy (*SD: standard deviation*).

Operator	Tcl	Rpc	Dph	Txm	Kmy	Dom	Mean	SD
MCO	100	83.90	91.47	100	64.76	97.93	89.67	13.70
PCI		100		94.83	97.92	80.22	93.24	8.94
OMD	89.56	98.85	100	100	97.92	99.76	97.68	4.06
CID	100	97.70	96.12	98.06	96.89	100	98.13	1.60
IOD		100	96.12	98.70	99.48	98.62	98.58	1.49
OAN					98.96		98.96	-
MCI				99.03			99.03	-
IPC		100	99.22		97.92	100	99.28	0.98
OMR	98.26	100	100		98.96	100	99.44	0.80
CDC		98.85		100	100		99.62	0.66
EHC		100			99.48		99.74	0.37
CDD	*100	98.85	*100	100	100	*100	*99.81	0.47
IHI		100		100	99.48	100	99.87	0.26
IOR		*100	100	99.67		*100	*99.92	0.17
IHD					100		100.00	-
ISD		100					100.00	-
PNC						100	100.00	-
CTD						100	100.00	-
CTI			100			100	100.00	0.00
ISI		100	100		*100	100	*100.00	0.00
IOP				100		*100	*100.00	0.00
PMD				*100	*100	*100	*100.00	0.00
PPD		*100		100	100	100	*100.00	0.00
CCA	100	100	*100	*100	*100	100	*100.00	0.00

Category 4: $100 > R_o \geq 99.5$

Category 5: $R_o = 100$

Once defined these categories, we applied two different selective mutation strategies:

Operator-based selection We performed the following steps for each case study from $i = 4$ to $i = 1$ (being i a variable to refer to a category[†]):

1. Select from the execution matrix the operators encompassed within categories [1...i] ($MO_{[1...i]}$).
2. Compute an adequate and minimal test suite for the selected operators ($T_{MO_{[1...i]}}$).
3. Include again in the execution matrix the mutants from the operators that were not in $MO_{[1...i]}$.
4. Calculate the mutation score associated with the test suite $T_{MO_{[1...i]}}$ and the reduction in the number of mutants.

Rank-based mutant selection In this strategy, we follow a similar approach to the *two-round random* selection technique proposed by Zhang et al. [11]. While in the two-round random technique the number of mutants selected from each operator is probabilistically speaking about the same, in rank-based mutant selection we seek to generate more mutants from the top ranked operators than from the operators at the bottom of the ranking based on mutant redundancy. Our rank-based mutant selection comprises two steps:

[†]The operators classified in the category 5 are removed in the first loop as we select the categories 1–4.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

1. *Operator selection*: The probability of being selected for an operator is proportional to its position in the ranking. As an example for three operators, the 1st operator in the ranking will be selected with probability $3/6$, the 2nd with $2/6$ and the 3rd with $1/6$.

2. *Mutant selection*: A mutant is randomly selected from the operator previously selected.

We performed the following steps for each case study from $i = 4$ to $i = 1$ (being i a variable to refer to a category):

1. Select using rank-based mutant selection with all operators as many mutants from D (set of dead mutants) as dead mutants are contained in the operators encompassed within categories [1...i] ($\left|D_{MO_{[1\ldots i]}}\right|$)[‡]. We call the set of selected mutants M_R from now on.

2. Compute an adequate and minimal test suite for the selected mutants (T_{M_R}).

3. Include again in the execution matrix the mutants that were not in M_R.

4. Calculate the mutation score associated with the test suite T_{M_R}.

We applied the above process 30 times[§] with different seeds and computed the average.

3.5.2. Results Table V classifies mutation operators into the five categories enumerated in Section 3.5.1 and presents the mutation score after performing the experimental procedure explained in that section. Each value of this table is the result of removing the operators within the categories under that row. As an example, only the operators *MCO*, *PCI* and *OMD* were applied to compute the mutation scores shown in the first row (category 1). Using these three operators, we achieved a mutation score over 90% in 4 out of 6 case studies. The second row presents the results of selecting the operators within category 1 and 2 (*MCO*, *PCI*, *OMD*, *CID*, *IOD* and *OAN*), where the mutation score was greater than 90% for all the case studies. We should note that the last row shows the data for the whole set of operators (the mutation score is therefore 100% because the test suite is adequate). We computed the mean (*Mean*) and the standard deviation (*SD*) of the results in all the case studies.

Table VI shows the reduction in the percentage of generated mutants because of the operators removed in each step, as well as the mean and the standard deviation (*SD*). As illustrated, applying the three operators in category 1, we achieve over 90% of the original mutation score with a reduction of more than half of the mutants (52.63%). Analogously, using the six operators from the categories 1 and 2 results in a mutation score of 97.22% on average (the standard deviation is 2.76) with a reduction in the number of mutants of 31.7% (standard deviation: 8.06). The mutation score gradually decreases when removing each of the categories, except for the operators in category 5. In this latter case, there is no lost of mutation score accuracy while achieving an average reduction in the number of mutants of 7.02%.

Table VII contains the results of the rank-based mutant selection technique. The mean mutation score (*M*) in each of the categories and the standard deviation (*SD*) are shown. As an example of the

[‡]We select the same number of mutants in both selective strategies in order to compare them later on in this paper.
[§]According to the guide by Arcuri and Briand [35], this is a common number of runs when assessing randomized algorithms.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table V. Mutation score when performing operator-based selective mutation based on the ranking of mutant redundancy (*SD: standard deviation*).

C.	Operators	Tcl	Rpc	Dph	Txm	Kmy	Dom	Mean	SD
1	MCO-PCI-OMD	93.0	92.0	89.1	94.8	79.3	94.9	90.52	5.89
2	CID-IOD-OAN	98.3	97.7	99.2	98.7	91.7	97.7	97.22	2.76
3	MCI-IPC-OMR	100	97.7	100	99.7	99.0	100	99.40	0.92
4	CDC-EHC-CDD-IHI-IOR	100	100	100	100	100	100	100	0
5	IHD-ISD-PNC-CTD-CTI ISI-IOP-PMD-PPD-CCA	100	100	100	100	100	100	100	0

Table VI. Reduction in the number of mutants by categories when applying operator-based selective mutation based on the ranking of mutant redundancy (*SD: standard deviation*).

Category	Tcl	Rpc	Dph	Txm	Kmy	Dom	Mean	SD
1	63.7	52.0	69.6	46.6	60.4	23.5	52.63	16.47
2	34.1	36.2	30.8	31.9	40.4	16.8	31.70	8.06
3	8.9	27.6	25.3	22.2	22.5	13.3	20.00	7.28
4	7.4	5.5	9.1	5.5	8.9	5.7	7.02	1.70
5	0	0	0	0	0	0	0	0

Table VII. Rank-based selection results based on the ranking of mutant redundancy (*M: mean; SD: standard deviation).*

PUT	1		2		3		4	
	M	SD	M	SD	M	SD	M	SD
Tcl	92.3	2.88	98.4	1.80	100	0.00	100	0.00
Rpc	95.1	3.61	98.8	1.86	99.8	0.42	99.9	0.58
Dph	93.1	2.57	98.2	1.39	99.1	0.97	99.3	0.75
Txm	99.8	0.49	100	0	100	0	100	0
Kmy	95.0	1.91	97.9	1.45	99.6	0.49	100	0.13
Dom	100	0	100	0	100	0	100	0
Total	95.89	3.30	98.87	0.92	99.77	0.35	99.87	0.27

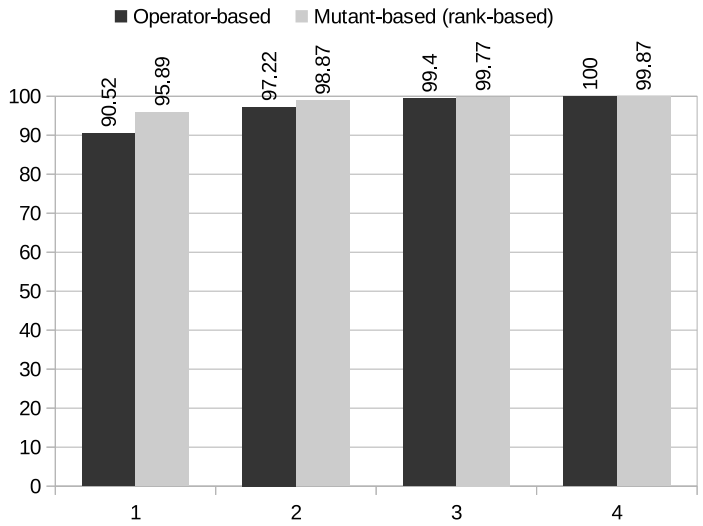


Figure 3. Comparison of the mutation score when using operator-based and mutant-based (rank-based) selection for the categories 1–4.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

meaning of the values in this table, the average in *Rpc* in category 2 (98.8%) is the mutation score when selecting the same number of dead mutants as dead mutants are generated by the operators included in the categories 1 and 2. In this way, the effectiveness of this strategy is comparable to the effectiveness of the operator-based strategy (97.7%) for the same case study and category (see Table V). As remarkable results, we can observe that in category 2 we achieve the full mutation score in two case studies and a total average score of 98.87%. The mutation score declines by 3% (95.89%) in category 1, but it is over 92% for all case studies. A comparison of the average results of the operator-based and rank-based strategy can be graphically seen in Figure 3 for each of the categories.

4. ASSESSMENT BASED ON TEST QUALITY

This section presents an evaluation of mutation operators for TSR analogous to the assessment undertaken for TSE in Section 3. In this regard, research questions, evaluation metric and experiments conducted are tackled.

4.1. Research questions

In this section we intend to answer the following research questions:

- **RQ3: What is the potential of each mutation operator for inducing the creation of high-quality test cases?** We aim at ranking mutation operators through their ability to generate hard to kill mutants, that is, mutants killed by few test cases which in turn kill few other mutants.
- **RQ4: Is a subset of mutants with a high potential to induce the generation of high-quality test cases sufficient for TSR?** We intend to know the loss in the number of test cases in the test suite when selecting (1) a subset of the top ranked operators based on test quality and (2) a subset of mutants in which the selection of mutants from the top ranked operators is favoured. This fact would allow us to analyse the trade-off between the reduction in the number of mutants and the effectiveness of the technique when refining a test suite.

4.2. Evaluation metric

Non-equivalent mutants that remain alive require additional test cases to be killed. However, a single test case could suffice to kill all of those surviving mutants if they model faults that are not difficult to reveal. Indeed, it is known that the more effective the test cases, the less effective the mutants [5]. The mutants offering resistance to be killed should be the most valued when determining a classification of operators for TSR. Therefore, giving a greater value to resistant and resistant hard to kill mutants (see Section 2.2) over other kinds of mutants seems a suitable approach in order to generate high-quality test cases. Estero-Botaro et al. [20] transformed this textual description into a mathematical formula. In this way, the formula is used to compute a quality metric which favours both resistant and resistant hard to kill mutants (see Equation 2). The use of an adequate, non-redundant and minimal test suite ensures that the calculation of the metric Q_m is not affected by the size of the test suite [20].

Prepared using stvrauth.cls
DOI: 10.1002/stvr

$$Q_m = \begin{cases} 0, & m \in E \\ 1 - \dfrac{1}{(|M| - |E|) \cdot |T|} \displaystyle\sum_{t \in K_m} |C_t|, & m \in D \end{cases} \tag{2}$$

Where:

- M is the set of valid mutants.
- E is the set of equivalent mutants.
- D is the set of dead mutants.
- T is an adequate and minimal test suite.
- K_m is the set of test cases that kill the mutant m.
- C_t is the set of mutants killed by the test case t.

This metric takes into account not only the number of test cases killing the mutant but also the number of mutants killed by those test cases at the same time. This is a desirable property because the fewer the mutants a test case detects, the more specific is that test case and therefore the fewer the mutants can induce the design of that test case. As a consequence, this metric seeks that the mutants killed by few test cases that in turn kill few other mutants are included in the subset of selected mutants: this will increase the probability that the more specific test cases are designed through the inspection of those mutants.

While resistant and resistant hard to kill mutants provide us with two clear examples of profitable mutants according to this metric, it is important to note that the rest of the mutants also receive a mark between 0 and 1 depending on the number of test cases and mutants killed by those test cases: the lower the number of test cases killing the mutant, the better the mutant. In the same line, the lower the number of mutants killed by those test cases, the better the mutant.

On this basis, they defined the quality metric of a particular mutation operator as the mean of the quality metric of the mutants generated with that mutation operator (see Equation 3).

$$Q_o = \frac{1}{|M_o|} \sum_{m \in M_o} Q_m \tag{3}$$

Where M_o is the set of mutants generated by the operator o.

The metric Q_o can be used as a means to rate operators by their potential to help the tester to enhance the fault detection power of the test suite. The operators with the highest quality metric should be the most valued. Notice that this quality metric penalizes the existence of equivalent mutants ($Q_m = 0$, as stated in Equation 2); this metric can be computed even when the operator only generates equivalent mutants (in that case, $Q_o = 0$).

As an example of this metric, consider the execution matrix in Figure 4. Being T_o an adequate and minimal test suite for the mutation operator o, we can compute the following adequate and minimal test suites for each mutation operator:

- $T_{o_1} = \{test_1, test_2\}$
- $T_{o_2} = \{test_3\}$

T_{o_1} is formed by $\{test_1, test_2\}$ because the test cases $test_1$ and $test_2$ can be used to kill the three mutants from operator o_1 (m_1, m_2, m_3). Then, the value of the quality metric for these two operators is:

DOI: 10.1002/stvr

Operator	Mutant	Test cases		
		$test_1$	$test_2$	$test_3$
o_1	m_1	0	1	0
	m_2	1	0	0
	m_3	1	0	1
o_2	m_4	0	0	0
	m_5	0	0	1
	m_6	1	0	1

Figure 4. Execution matrix to illustrate the *quality metric*.

1. Quality of operator o_1 ($|M_{o_1}| = 3$, $|E_{o_1}| = 0$, $|T_{o_1}| = 2$, $C_{test_1} = \{m_2, m_3\}$, $C_{test_2} = \{m_1\}$):

   - $Q_{m_1} = 1 - 1/((3-0) \cdot 2) = 0.83$ where $K_{m_1} = \{test_2\}$
   - $Q_{m_2} = 1 - 2/((3-0) \cdot 2) = 0.67$ where $K_{m_2} = \{test_1\}$
   - $Q_{m_3} = 1 - 2/((3-0) \cdot 2) = 0.67$ where $K_{m_3} = \{test_1\}$

   $Q_{o_1} = (0.83 + 0.67 + 0.67)/3 = 0.72$

2. Quality of operator o_2 ($|M_{o_2}| = 3$, $|E_{o_2}| = 1$, $|T_{o_2}| = 1$, $C_{test_3} = \{m_5, m_6\}$):

   - $Q_{m_4} = 0$ (equivalent)
   - $Q_{m_5} = 1 - 2/((3-1) \cdot 1) = 0$ where $K_{m_5} = \{test_3\}$
   - $Q_{m_6} = 1 - 2/((3-1) \cdot 1) = 0$ where $K_{m_6} = \{test_3\}$

   $Q_{o_2} = (0 + 0 + 0)/3 = 0$

Interpreting these results, the operator o_1 is more valued than o_2 with this metric ($Q_{o_1} = 0.72 > Q_{o_2} = 0$) because of the following facts:

- The operator o_2 generates an equivalent mutant (m_4), which is penalized by the metric.
- Both m_5 and m_6 (o_2) can be killed with a single test case ($test_3$), which always results in $Q_m = 0$.
- As $test_3$ suffices to kill the mutants generated by o_3, $test_1$ and $test_2$ may not be generated without considering o_1. Furthermore, $test_2$ would be generated only after analysing the first mutant (m_1) from o_1, as m_1 is the only mutant killed by $test_2$.

As a conclusion, a mutation operator with a high value of Q_o increases the probability of missing some test cases when performing a selective mutation strategy without mutants generated by that operator. Therefore, the operators with the highest Q_o should be at the top of the ranking.

4.3. Experiment #1: Ranking mutation operators

This first experiment aims to apply the quality metric described in Section 4.2 to each mutation operator.

4.3.1. Setup The same case studies and adequate test suites presented in Section 3.3 were used in this experiment. We should note that in the experiments conducted by Estero-Botaro et al. [20], the

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table VIII. Ranking of mutation operators based on test quality (*SD: standard deviation*).

Operator	Tcl	Rpc	Dph	Txm	Kmy	Dom	Mean	SD
IOD		-	0.80	0.78	-	0.89	0.82	0.06
MCO	-	0.74	0.79	0.72	0.89	0.73	0.77	0.07
OMR	0.88	0.92	0		0.98	0.89	0.73	0.41
OMD	0.80	0.82	-	0.52	0.68	0.71	0.71	0.12
IPC		-	0.30		0.65	0.79	0.58	0.25
CID	0.83	0.74	0.52	0.58	0.52	0.29	0.58	0.19
ISI		-	0.57		-	-	0.57	-
CDC		-		-	0.47		0.47	-
PCI		-		0.71	0	0.60	0.44	0.38
OAN					0.38		0.38	-
IHI		0		0.72	0.29	0.39	0.35	0.30
MCI				0.30			0.30	-
IOR		0	0.07	0.65		-	0.24	0.36
PPD		-		0	0.17	0.11	0.14	0.09
CCA	0.17	-	0	0	-	0.25	0.10	0.13
CDD	-	0.30	-	0	0	0	0.07	0.15
IOP				0		-	0	-
PMD				-	-	0	0	-
EHC		-			0		0	-

metric was measured only for operators generating at least four mutants. Despite class operators are known to generate fewer mutants than traditional operators, we have maintained this condition in our experiments. We calculated the quality metric of the operators for each case study (as explained by Estero-Botaro et al. [20]) and computed a mean with the values obtained for each operator. Finally, a ranking was prepared in ascending order of Q_o.

4.3.2. Results Table VIII shows the results of applying the quality metric for each operator and case study, where operators are sorted by the mean in our case studies (the standard deviation is also calculated). *IOD* (0.82) is the most valued operator on average, followed by *MCO* (0.77) and *OMR* (0.73). On the contrary, the operators *IOP*, *PMD* and *EHC* present the lowest quality metric, so they are at the bottom of the classification. Mutation operators which could not be rated with this metric in a case study are marked with '-' (as aforementioned, the threshold is set in operators generating at least four mutants). We should remark that five operators (*IHD*, *ISD*, *PNC*, *CTD* and *CTI*) are not shown in the table as they did not generate more than three valid mutants in any of the case studies (see Table III). The mutation operator *OMR*, ranked 3rd, was the only operator obtaining values over 0.9 in some of the subject programs. The quality metric in the rest of operators with $Q_o > 0$ range from 0.07 to 0.71, with varying standard deviations across the ranking. Note that mutants from operators with $Q_o = 0$ in a case study are either equivalent or all the mutants are killed by all the test cases in the adequate and minimal test suite for the operator, as shown in the example in Section 4.2.

4.4. Experiment #2: Selective mutation based on the ranking

As explained in Section 3.5.1, in the second experiment we perform selective mutation using the ranking obtained in the previous experiment (see Section 4.3). The goal is to observe the loss in the number of test cases in an adequate and minimal test suite for the full set of mutants when applying operator-based and rank-based mutant selection for TSR. Recall that the quality metric favours the

Prepared using stvrauth.cls
DOI: 10.1002/stvr

mutants killed by few test cases which kill few mutants at the same time. It is expected that the mutants produced by the best-valued operators help design test cases that kill few mutants, that is, high-quality test cases. Therefore, the mutation score will not be representative of the full set of mutants in this case, yet we will be retaining test cases which are not easy to design.

4.4.1. Setup We gathered the operators with a similar quality metric into five categories, but also trying to balance the number of operators in each category (see Table VIII):

Category 1: $0.70 < Q_o$

Category 2: $0.50 < Q_o \leq 0.70$

Category 3: $0.25 < Q_o \leq 0.50$

Category 4: $0.00 < Q_o \leq 0.25$

Category 5: $Q_o = 0.00$

The five mutation operators that could not be assessed are included in the fifth category, as they are supposed not to have a significant influence on the results.

Operator-based selection Once defined these categories, we performed the following steps for each case study from $i = 4$ to $i = 1$ (being i a variable to refer to a category):

1. Select from the execution matrix the operators encompassed within categories $[1...i]$ ($MO_{[1...i]}$).
2. Compute an adequate and minimal test suite for the selected operators ($T_{MO_{[1...i]}}$).
3. Calculate the loss of test cases with respect to the original adequate and minimal test suite, $|T_{MO}| - \left|T_{MO_{[1...i]}}\right|$, and the reduction in the number of mutants.

Rank-based mutant selection As in Section 3.5.1, we executed 30 times the following steps for each case study from $i = 4$ to $i = 1$ (being i a variable to refer to a category) and computed the average:

1. Select with the rank-based technique the same size of dead mutants from all operators as mutants of this kind are contained in the operators encompassed within categories $[1...i]$ ($\left|D_{MO_{[1...i]}}\right|$). Recall, M_R represents the set of selected mutants.
2. Compute an adequate and minimal test suite for the selected mutants (T_{M_R}).
3. Calculate the loss of test cases with respect to the original adequate and minimal test suite: $|T_{MO}| - |T_{M_R}|$.

4.4.2. Results Table IX classifies mutation operators into the five categories (*C*) enumerated in Section 4.4.1. This table shows the percentage of loss in the number of test cases from the original adequate and minimal test suite as a consequence of removing the mutants from the operators under that category. Again, we obtained the mean as well as the standard deviation (*SD*) of the results

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table IX. Percentage of test cases loss when performing operator-based selective mutation based on the ranking of test quality (*SD: standard deviation*).

C.	Operators	Tcl	Rpc	Dph	Txm	Kmy	Dom	Mean	SD
1	IOD-MCO-OMR-OMD	0	20.0	18.2	53.3	30.6	20.0	23.68	17.57
2	IPC-CID-ISI	0	13.3	0	33.3	13.9	20.0	13.42	12,65
3	CDC-PCI-OAN-IHI-MCI	0	6.7	0	6.7	2.8	0	2.70	3.28
4	IOR-PPD-CCA-CDD	0	0	0	0	2.8	0	0.47	1.10
5	IOP-PMD-EHC IHD-ISD-PNC-CTD-CTI	0	0	0	0	0	0	0	0

Table X. Reduction in the number of mutants by categories when applying operator-based selective mutation based on the ranking of test quality (*SD: standard deviation*).

Category	Tcl	Rpc	Dph	Txm	Kmy	Dom	Mean	SD
1	38.5	44.1	48.5	79.8	53.9	83.5	58.05	19.01
2	8.9	27.6	19.7	71.3	29.3	79.7	39.42	28.99
3	8.9	20.5	19.7	9.7	11.4	6.1	13.20	5.96
4	0	2.4	1	2.7	2.9	1.5	1.75	1.13
5	0	0	0	0	0	0	0	0

Table XI. Rank-based selection results based on the ranking of test quality (*M: mean; SD: standard deviation).*

PUT	1		2		3		4	
	M	SD	M	SD	M	SD	M	SD
Tcl	9.6	5.7	0	0	0	0	0	0
Rpc	8.9	7.1	1.6	2.9	0.2	1.2	0	0
Dph	10.9	6.4	1.5	3.0	1.5	3.0	0.5	1.4
Txm	28.0	8.1	19.1	5.7	0	0	0	0
Kmy	6.0	2.4	0.6	1.2	0	0	0	0
Dom	15.5	4.4	11.5	5.2	0	0	0	0
Total	13.14	7.91	5.72	7.82	0.29	0.61	0.08	0.19

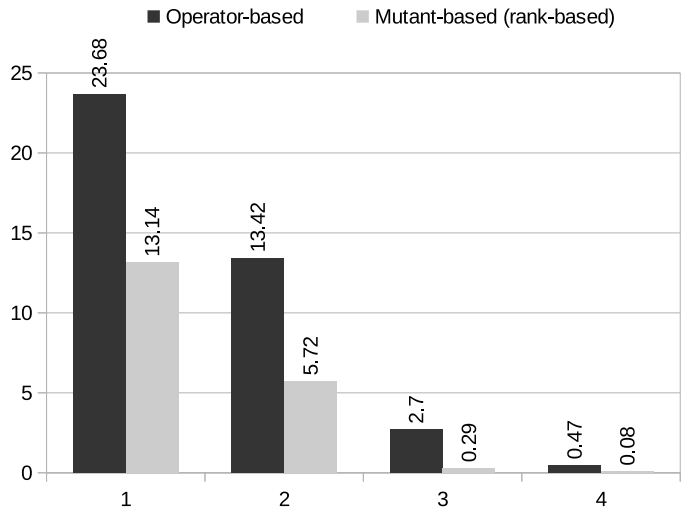


Figure 5. Comparison of the percentage of test cases loss when using operator-based and mutant-based (rank-based) selection for the categories 1–4.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

of each case study. Table X depicts the percentage of reduction in the number of mutants by not considering the operators removed in each step.

From Table IX, we can observe an increasing drop in the number of test cases in the adequate and minimal test suite, from 0.47% after removing the operators within category 5 to 23.68% only using the operators within category 1. Considering the 16 operators with $Q_o > 0$ (from category 1 to 4), the same number of test cases remains in the adequate and minimal test suite, except for a loss of one test case in *Kmy* (2.8%). The reduction in the number of mutants is not very relevant when removing the operators within categories 4 and 5, but meaningful when selecting the first 7 operators in the ranking (39.42%). In that case, we assume a loss of 13.42% of test cases. We should note an increasing standard deviation because of dissimilar results among case studies, especially *Tcl* and *Txm* in the categories 1 and 2. In addition, the number of test cases in the minimal test suite is not very high in most case studies (from 15 to 36 test cases, as can be seen in Table II), so the reduction of each test case implies a great percentage.

Table XI shows the results of the rank-based mutant selection. By using the same size of mutants as in the operators within categories 1 and 2, we assume a mean loss of test cases of 5.72% with a standard deviation of 7.82. This percentage increases to 13.14% as a consequence of discarding the number of dead mutants generated by the operators in category 2. Overall, we can also observe that the standard deviations progressively increase from category 4 to 1: the fewer the mutants selected, the more varied are the results in the different executions. As in the previous section for TSE, we show in Figure 5 the average results for operator-based and rank-based selection together.

5. DISCUSSION

5.1. Validation of operator-based selection results

As a sanity check, we compared our operator-based selection results with three new rankings of operators. We carried out operator-based selective mutation by establishing categories of operators with these new rankings, as done in Section 3.5.1 and Section 4.4.1. To this end, we followed classical approaches to selective mutation:

- **Random:** random sort of mutation operators. For a direct comparison between categories with the same number of operators, we maintained the same sizes of the categories from the original experimental results (see Section 3.5.2 and Section 4.4.2).
- **Number of mutants (Size):** sort of mutation operators by the number of mutants [7, 15], where the most prolific operators are at the bottom of the ranking.
  In order to retain a significant number of mutants at all times, the category size is proportional to the number of mutants generated in the analysed programs (see Table III). Thus, we divided the total number of mutants (1,868) by 5 categories, which results in 374 mutants per category. Then, we included as many operators as needed to complete 374 mutants, which depends on the mutants produced by each operator. As an example, *PCI* (the most prolific operator) is the only operator in the category 5 as it generates 659 mutants, which suffices to reach the number of mutants set for a category.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table XII. Arrangement of the rankings *Random*, *Size* and *Block* classified into categories for TSE and TSR.

Category	**Random** $_{TSE}$	**Random** $_{TSR}$	**Size** $_{TSE,TSR}$	**Block** $_{TSE,TSR}$
1	IPC,OMR,ISD	IPC,OMR, ISD,ISI	CTD,ISD,IHD,PNC, CTI,OAN,EHC,PMD, CDC,IOP,ISI,CDD, IPC,CCA,MCI,PPD, IOR	IHD,IHI,ISD, ISI,IOD,IOP, IOR,IPC
2	ISI,CCA,OMD	CCA,OMD, CTI	IOD,IHI, OMR,OMD	CTD,CTI,CID, CDC,CDD,CCA
3	CTI,PNC,MCI	PNC,MCI,IOD, MCO,IOP	CID	PCI,PMD,PPD, PNC
4	IOD,MCO,IOP, CDC,PCI	CDC,PCI, CID,PMD	MCO	OMD,OMR, OAN
5	CID,PMD,IOR, IHD,IHI,CTD, CDD,OAN, EHC,PPD	IOR,IHD,IHI, CTD,CDD, OAN,EHC, PPD	PCI	MCO,MCI,EHC

Table XIII. Comparison of the mutation score when using operator-based selective mutation testing for TSE with the rankings *Random*, *Size* and *Block* (*M: mean; SD: standard deviation).*

Category	***Original***		***Random***		***Size***		***Block***	
	M	SD	M	SD	M	SD	M	SD
1	90.52	5.89	48.89	29.90	56.02	21.47	53.70	31.09
2	97.22	2.76	74.66	14.10	82.76	13.52	78.15	14.77
3	99.40	0.92	78.17	13.94	85.18	13.48	84.36	15.62
4	100	0	97.54	2.12	95.50	7.75	89.43	13.75

Table XIV. Comparison of the percentage of test cases loss when using operator-based selective mutation testing for TSR with the rankings *Random*, *Size* and *Block* (*M: mean; SD: standard deviation).*

Category	***Original***		***Random***		***Size***		***Block***	
	M	SD	M	SD	M	SD	M	SD
1	23.68	17.57	66.71	19.38	66.97	11.76	64.46	22.17
2	13.42	12.65	48.09	23.73	26.56	11.45	37.68	12.38
3	2.68	3.28	19.31	12.98	17.69	9.81	30.79	17.92
4	0.45	1.10	4.07	4.75	5.35	7.34	13.92	10.51

- **Operator type (Block):** mutation operators of the same operator block are grouped together [12] (see Table I). These blocks are sorted by the number of operators that they contain. The block with more operators ("inheritance") is at the top of the ranking.

  The category size in this ranking depends on the number of operators within each group. In this regard, we only counted operators creating at least one mutant in our case studies. For instance, 3 out of 4 operators from the "method overloading" block were applied (*OMD*, *OMR* and *OAN*). Because of the few operators, the groups "exception handling" and "object and member replacement" were gathered in a category.

The final arrangement of these three rankings and their classification into categories is depicted in Table XII. We should remark that the categories for TSE and TSR are different in the case of *Random* because their category size is related to the number of operators within each category in the original experiments.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

For each ranking, Table XIII shows the mutation score achieved by the operators of each category following the selective mutation procedure described in Section 3.5.1. Analogously, Table XIV presents the percentage of reduction in the number of test cases in the adequate and minimal test suite. The column *Original* illustrates the mean (*M*) and the standard deviation (*SD*) obtained in our original rankings (see column *Mean* and *SD* in Table V and IX).

The results of the original rankings on average are clearly better than the results of the rankings *Random*, *Size* and *Block*. The ranking *Random* shows the worst performance in general in both the calculations of the mutation score and the loss in the percentage of test cases, except when removing the first turn of operators. The ranking *Size* gets better results than *Block* in both selective strategies in 7 out of 8 cases (four categories for TSE plus other four categories for TSR). On few occasions, the rankings *Random* and *Size* match the outcome of our original results in a pair '(*case study*, *category*)', but the averaged results are still very far from the ones achieved with the original rankings. As an exception, we have to note that the ranking *Block* is able to surpass the ranking based on test quality for *Txm* when selecting the operators from category 1 and the operators from the categories 1 and 2. Nevertheless, the high standard deviations in these three rankings suggest that they are not consistent.

5.2. Validation of rank-based mutant selection results

Similarly to the previous sanity check, we aim to compare our rank-based mutant selection results with other strategies for the selection of mutants [11]:

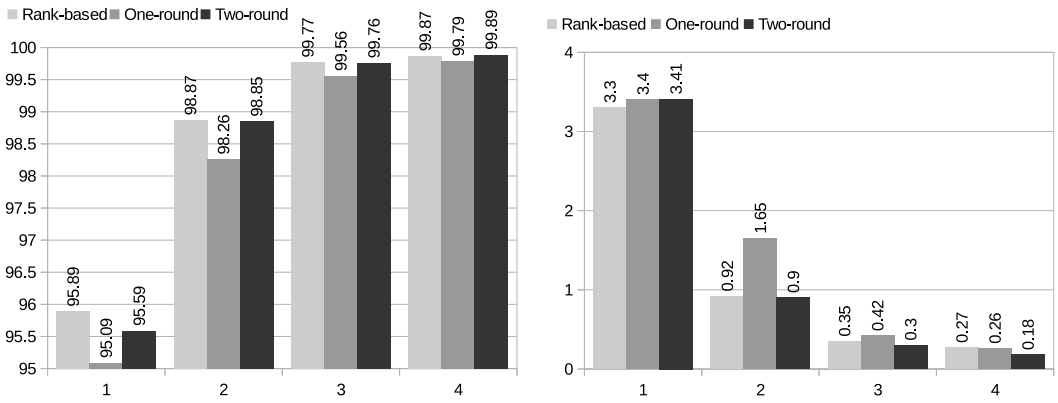
- **One-round random (One-round):** random selection of mutants from all the operators (equal probability of selecting each of the mutants).
- **Two-round random (Two-round):** in the first round, one operator is selected randomly; in the second round, one mutant is selected randomly from the operator selected in the first step (equal probability of selecting a mutant from each of the operators).

In both strategies, we select the same number of mutants in each category for TSE and TSR as in the rank-based mutant selection. Figures 6 and 7 show graphically the comparative performances of the strategies for TSE and TSR respectively (mean and standard deviation).

The original rank-based mutant selection based on the operator classification for TSE and TSR outperforms in all the cases the *One-round* strategy on average. *One-round* also shows a worse performance than *Two-round*. While *One-round* was able to obtain a better result in TSE for the two first categories when analysing *Dph*, the original strategy was better in the rest of the cases except for a few ties (with a remarkable difference of 2.6% in the pair '(*Kmy*, *category 2*)'). Regarding TSR, we can find a noteworthy gap between the two strategies in '(*Dom*, *category 1*)'): 18.1% for rank-based selection and 30.6% for *One-round*.

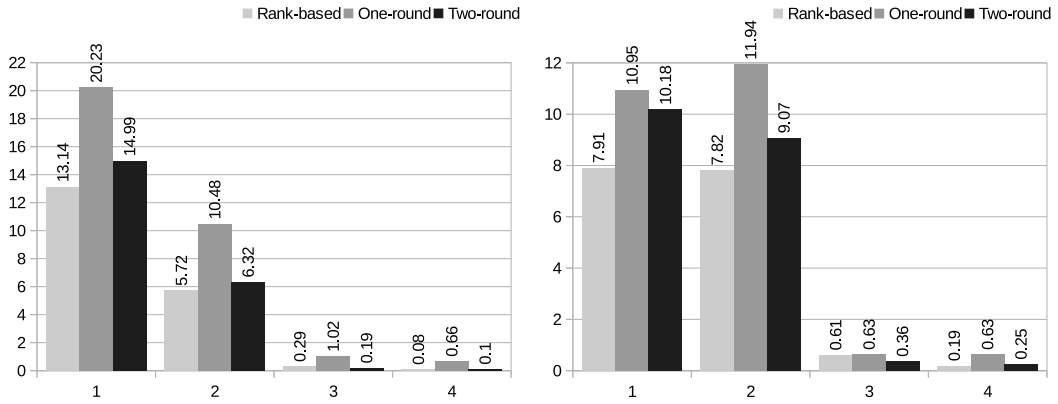
As for *Two-round*, the original strategy gets better results in 6 out of 8 cases on average and the gap between the two selective approaches widens as the number of mutants selected decreases. This outcome is quite interesting as that means that the operator rankings work better for large reductions of mutants. If we focus on the first category, while in TSE the rank-based strategy surpasses *Two-round* by 0.3% on average (note that the margins are narrow because of the nature of the mutation score), the difference is more notable when measuring the percentage of test cases lost (1.85%). The standard deviation in *Two-round* is also higher than in the original strategy for both evaluations in

Prepared using stvrauth.cls
DOI: 10.1002/stvr



(a) Mean (b) Standard deviation

Figure 6. Comparison of the mutation score when using *Rank-based* selective mutation testing for TSE with the rankings *One-round* and *Two-round* (*left: mean; right: standard deviation).*



(a) Mean (b) Standard deviation

Figure 7. Comparison of the percentage of test cases loss when using *Rank-based* selective mutation testing for TSR with the rankings *One-round* and *Two-round* (*left: mean; right: standard deviation).*

that category, which means that the former strategy is less stable than the latter when few mutants are selected. Regarding the individual programs, *Two-round* only produces better results overall in *Dph*. There are relevant differences in favour of rank-based mutant selection in several cases, like in the pair '(*Txm*, *category 1*)') with a difference of 6.6% in the percentage of test cases lost.

5.3. Answer to research questions

Answer to RQ1: What is the degree of redundancy of each mutation operator?

Results show that 14 out of 24 operators generate non-redundant mutants with the test suite used in the experiments (see Table IV). The most prolific operators are generally the operators with the lowest degree of redundancy. Moreover, it is interesting to observe that all operator blocks but "exception handling" are represented in the top 5 of the ranking. This fact conveys that each operator block indeed targets different object-oriented features, that is, these groups are not redundant among

Prepared using stvrauth.cls
DOI: 10.1002/stvr

them. We should note that the operators *IMR*, *PVI*, *PCD*, *PCC*, *OAO* and *EHR* produce no valid mutants in these case studies, so we cannot calculate the operator redundancy for them.

Answer to RQ2: Is a subset of mutants with low degree of redundancy sufficient for TSE?

Yes. Results reveal that (1) using the top 6 operators (*MCO*, *PCI*, *OMD*, *CID*, *IOD* and *OAN*) allows us to reduce the number of mutants (31.7%) with an acceptable measurement of the overall mutation score (97.22%), as can be seen from Tables V and VI, and (2) favouring the selection of mutants from the top ranked operators leads to an average mutation score of 98.87% with the same number of mutants (see Table VII). Different approaches to selective mutation report worse results on average in all cases (see Table XIII and Figure 6), which supports that the operator redundancy is a good indicator for TSE when performing a selective strategy.

Answer to RQ3: What is the potential of each mutation operator for inducing the creation of high-quality test cases?

The potential of 19 mutation operators for TSR with high-quality test cases is presented in Table VIII. In addition to the operators generating no valid mutants, five operators are not evaluated (*IHD*, *ISD*, *PNC*, *CTD* and *CTI*) and other three operators obtain the lowest value (*IOP*, *PMD* and *EHC*). The existence of several mutation operators with the lowest metric in different case studies matches with the few mutants generated by class-level operators and the high equivalence percentage that they usually present. Unlike the ranking based on mutant redundancy, none of the operators from the "polymorphism and dynamic binding" group is in the top 5 of the ranking based on test quality, finding the first one (*PCI*) in the 9th position.

Answer to RQ4: Is a subset of mutants with a high potential to induce the generation of high-quality test cases sufficient for TSR?

Yes. (1) Using the top 12 rated operators (see Table IX), 2.7% of the test cases are not included in the adequate and minimal test suite on average, with a reduction of 13.2% in the number of mutants (see Table X). When cutting out also the five operators within category 3, we should assume a decrease of 13.42% of test cases while 39.42% of the mutants are not examined; (2) using rank-based selection of mutants, we observe a smaller decrease (13.14%) with just the same number of mutants as in the four operators in category 1 (see Table XI). As in the answer to RQ2, the outcome when applying selective mutation following other strategies (see Table XIV and Figure 7) shows that the quality metric is a good indicator for TSR when performing a selective strategy.

5.4. Comparison between rankings

In this section, we compare the rankings arranged in Section 3.4.2 and 4.3.2, discussing similarities and differences in order to know whether there is a connection between the rankings based on mutant redundancy for TSE and test quality for TSR.

An overall view of these rankings allows us to observe a patent similarity between them. It can be observed from the two rankings that *MCO*, *OMD* and *IOD* are fruitful class mutation operators because these operators occupy the first positions in both classifications, whereas *PMD* and *IOP* are not so useful because they are at the bottom of these two rankings. This fact suggests that the most suitable mutation operators for TSE are also the most appropriate in terms of TSR.

However, looking at the assessments of each operator more carefully, we can notice some discordant results between both rankings. *PCI* falls from 2nd in the ranking based on operator redundancy (see Table IV) to 9th in the ranking related to test quality (see Table VIII). As a

Prepared using stvrauth.cls
DOI: 10.1002/stvr

conclusion, while *PCI* shows a low operator redundancy, test cases are quite effective with the mutants from this operator, so *PCI* is not such an useful operator to generate many new test cases not considered yet. On the contrary, *OMR* climbs six positions (from 9th in Table IV to 3rd in Table VIII) and *ISI* eight positions (from 15th to 7th). *EHC*, *CDD*, *IOD*, *OAN* and *MCI* also exhibit a significant change in their positions. These differences between the rankings validate the need to undertake a separate study of mutation operators as the one accomplished in this paper.

Finally, regarding the class operators specifically designed for C++ [6], only *CDD* and *CCA* generated mutants in our experiments. *CDD* is in the 12th and 16th position in the ranking for TSE and TSR respectively, while *CCA* is placed the last and the 15th. Consequently, as these two operators occupy low positions in both rankings, their mutants are candidates to be discarded in most selective processes regardless of the goal when applying mutation testing.

5.5. Comparison between selective mutation strategies

Comparing the results reported by operator-based and rank-based mutant selection is the goal of this section. We study this aspect separately for the two evaluations:

- **TSE:** While operator-based selection obtains 100% in the mutation score when removing the operators at the bottom of the ranking, the rank-based mutant selection offers better performance in the other three categories, especially in category 1 where the gap is over 5% on average (90.52% vs 95.89%).
- **TSR:** Rank-based mutant selection gets much better results in all of the categories. Interestingly, the average result of the rank-based strategy in the first category not only outperforms the result of operator-based selection in the same category but also in the second category (13.42% vs 13.14%).

In general, rank-based mutant selection also seems a more stable strategy when we analyse the standard deviations in the different categories. Surprisingly, a simple random selection of mutants also turned out to be better than operator-based selection except for the fourth category in both TSE and TSR evaluations (see Section 5.2). We presume that this fact is related with the aforementioned conclusion that each operator block addresses different object-oriented features (see Section 3.4.2). Unlike traditional operators, we suspect that several of these class operators are hardly redundant among them because they target completely different parts of the code. For instance, *CDD* addresses destructor methods whereas *EHC* tackles exceptions. The fact that *Two-round* random outperforms *One-round* random also supports this idea. As a consequence, we might be reducing the coverage of the test suite when removing some of the class operators, which can diminish the benefits of selective mutation. Therefore, even though the operator-based selection results are acceptable, a strategy for the selection of mutants from all operators seems more suitable when it comes to class-level mutation.

5.6. Threats to validity

There are several aspects that pose a threat to the validity of the results reported in this paper.

Number of mutants. Although we selected six different libraries and applications, some of the operators were never applied or only produced few mutants in few case studies. As stated,

Prepared using stvrauth.cls
DOI: 10.1002/stvr

the appearance of mutants at the class level depends on the object-oriented features used by the programmer, and these operators are less prolific than traditional operators. As a result, by maintaining the threshold in the number of mutants to apply the quality metric used by Estero-Botaro et al. [20], the metric could not be computed for several operators (notably in *XmlRpc++*, where Q_o could not be measured for eight operators). Altogether, the number of mutants supposes a threat to the generalization of the results as some operators could not be appropriately evaluated.

Mutant equivalence. Equivalence is an undecidable problem, thus judging a mutant to be equivalent is an error-prone task, especially when analysing third-party applications for which it is not trivial to acquire a full insight into the source code. To counter this threat, we used the concept of *undecided* so that the mutants of uncertain condition were not recorded as equivalent.

Test suites. To the best of our knowledge, there is no an available test case generator for object-oriented programs in C++ which tries to produce new test cases specifically with the target of killing surviving mutants as *EvoSuite* does for Java [36]. Test cases were therefore manually designed with a view to kill non-equivalent mutants remaining alive, but we proceeded with the utmost care to develop consistent test cases. Nonetheless, the metrics used in these experiments, and consequently the results, are subject to the test suite.

Comparison results. Finally, in order to check that the results when carrying out the selective strategy based on the rankings are not coincidental, we prepared three combinations of operators following traditional approaches to operator-based selective mutation and executed other two strategies for random mutant selection. These strategies yielded worse results, but new combinations could be arranged to confirm the observed tendency. We should note that we have compared the techniques under the same number of non-equivalent mutants, as done by Zhang et al. [11]. However, the rank-based strategy will select a variable number of equivalent mutants in practice, which might impact the results. Recently, Papadakis et al. [37] provided evidence that mutation-based assessments comparing test techniques are vulnerable to a potential threat to validity: the presence of redundant mutants. Future evaluations should remove as many redundant mutants as possible and observe if the same results hold.

6. RELATED WORK

6.1. Selective mutation testing

Operator-based selective mutation was first conceived by Mathur [38] for the purpose of reducing the large computation expenses. The approach of removing some of the mutation operators has been investigated since then by many researchers [7, 12, 13, 15, 16, 17, 39]. In this regard, Wong and Mathur [15] limited mutation testing to the use of two Fortran operators (*ABS* and *ROR*), as they can achieve similar results than the 22 operators included in Mothra. Offutt et al. [7] performed selective mutation omitting the *N* most commonly applied operators. Among other results, by excluding the 6 operators that engendered more mutants, adequate test suites for the remaining mutants (around 40% of the complete set of mutants) maintain a high correlation with the full mutation score (99.71%).

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Shortly after that, they experimented with selective mutation by excluding all the operators that belong to the same operator block instead [12]. Based on the experimental results, they determined that only 5 mutation operators (those mutating operators within expressions) suffice to obtain an average effectiveness value of 99.5%. Offutt et al. [7] defined the *operator strength* as the number of mutants killed by a test suite generated only to kill the mutants of that operator. In our paper, we take a different approach by defining the operator redundancy to count the number of mutants that remain alive when the test suite is generated for all the operators apart from that operator.

Mresa and Bottaci [17] also changed the traditional approach so far by evaluating operators regarding two factors: mutation score and cost information about test data generation as well as equivalent mutant identification. In their experiments considering these cost factors, they found that operator-based selection is a preferable option when compared to random selection of mutants but only if low mutation scores are required. They used effective and non-redundant test cases in their empirical procedure. In the experiments conducted in this paper, the operators are assessed separately for TSE and TSR regarding redundancy and test quality respectively. Moreover, we go a step further by imposing minimality to the test suites, as encouraged by Estero-Botaro et al. [20].

The statistical analysis procedure defined by Namin et al. [16] identified 28 operators among 128 implemented in Proteum as sufficient for an accurate measurement of the mutation score for all the operators. The results of their approach to select a subset of operators executing a large set of mutants and test cases does not support the intuition that one operator from each operator group should be selected, as in the guidelines proposed by Barbosa et al. [13]. In our experiments, 5 out of 6 operator blocks are represented in the top 5 ranked operators for TSE, but only 3 in the case of TSR. The 10 mutation operators selected by Barbosa et al. [13] showed effectiveness values between 95.8% and 100% when applied to 27 cases studies. Delamaro et al. [39] proposed to use a greedy algorithm for choosing a reduced set of C mutation operators, successively adding the operators that increased the overall score the most. They concluded that the high redundancy among the operators makes difficult to establish a single way to select the best operators. The study directed by Zhang et al. [40] showed that selective mutation scales with regard to the size of the PUT.

Random mutant selection, also known as *mutant sampling*, was proposed by Budd [14] and Acree [41], where they showed that just sampling 10% of the mutants is sufficient to predict the mutation score for all the mutants with high accuracy. Despite the particular attention received by operator-based selection in the literature, a growing body of research in recent years gives evidence that it is not superior to random mutant selection [11, 42, 43]. This conclusion was drawn by Zhang et al. [11] when comparing random selection of mutants with several sufficient sets of operators in the literature (5 operators in Offutt et al [12], 10 operators in Barbosa et al. [13] and 28 operators in Namin et al. [16]). The experiments by Gopinath et al. [43] also suggest that removing operators could offer limited benefit in comparison to random mutant selection. Finally, Zhang et al. [42] applied 8 different random strategies for the selection of mutants, concluding that operator-based and random mutant selection can be combined to further reduce the cost. We also study both selective techniques when used with class-level operators. Mutant-based selection and especially a rank-based strategy results in more representative results of the full set of mutants in this case.

DOI: 10.1002/stvr

6.2. Object orientation

Selective mutation testing applied to class mutation operators has been previously studied for other object-oriented languages. Derezińska and Rudnik [18] conducted their experiments with C# on 18 class operators and 8 standard operators at the same time using three case studies. The results evidence that, even with a considerable reduction of object-oriented mutants (using 74% of the mutants) still 93% of the original mutation score can be achieved.

Likewise, Ma et al. [22] explored the elimination of some unnecessary class operators in Java that generated very few mutants. Bluemke and Kulesza [19] performed a selective reduction of mutants generated by Java operators, including class-level operators. In their experiments, they showed that the strategy can significantly reduce the cost (between 40% and 60% of mutants) while preserving an acceptable mutation score and code coverage. In our work, by using rank-based mutant selection, over 30% of the mutants can be excluded but declining less than 1.2% the mutation score. In the same way, almost 40% of the mutants can be saved but losing 5.72% of test cases.

In previous work using class operators for C++ [6], the mutation score was calculated around two case studies and the test suites distributed with them, showing which operators spawned more mutants that remained undetected. Several improvement rules for these operators where proposed later, analysing their impact in the resources required to generate and execute the mutants for five applications [25]. In addition, that paper studied how useful class operators are by comparing them with a set of traditional operators. However, in this paper we evaluate class operators by their usefulness for TSE and TSR using six different programs with adequate test suites, and finally we carry out a selective strategy in the light of the results.

6.3. Other related works

Most of the approaches to decrease the overall cost of mutation testing have been collected in the survey by Jia and Harman [44]. In addition to selective mutation, many authors have concentrated their efforts on reducing the number of mutants produced. Some helpful techniques are *high-order mutation* [9], *mutant clustering* [10] and *evolutionary mutation testing* [45].

While the work in this paper aims to obtain a classification of mutation operators generalizable for every PUT, the mutation tool *MuRanker* [46] ranks mutants depending on a prediction about the difficulty to create a test case to kill them. The ranking displayed with this tool is thereby particular for each PUT. Javalanche [47] remarks the mutants with a high impact (they are less likely to be equivalent) as those mutants that can really help to improve a test suite because they are easier to assess by a tester. That mutation tool takes a different approach to the quality metric by Estero-Botaro et al. [20], where the most difficult to kill mutants are the most valued. Kusano and Wang [48] developed the mutation tool *CCMutator* for C++, similar to *MuCPP*, but they focus on the generation of mutants for concurrency constructs in multi-threaded applications.

Moghadam and Babamir [49] recently proposed to estimate the mutation adequacy score taking into account several object-oriented metrics which capture the structural complexity of the analysed program. Another related work was undertaken by Just et al. [50], analysing how redundancy affects both the efficiency and effectiveness of mutation testing. Wright et al. [51] use the term *redundant mutant* in a broad sense, also considering as redundant those mutations which may be produced

Prepared using stvrauth.cls
DOI: 10.1002/stvr

by different operators. This type of ineffective mutants should be removed to avoid consuming resources unnecessarily.

Yao et al. [52] defined a non-equivalent mutant as *stubborn* when it is not killed by a branch adequate test suite. Thus, stubborn mutants are theoretically more difficult to kill than resistant mutants given that a resistant mutation may not have been executed by the test suite before adding the test case to detect that mutation. A resistant hard to kill mutant requires that the test case killing it does not kill any other mutants, and this double perspective is not contemplated by stubborn mutants. Ammann et al. [27] proposed using *minimal sets of mutants* to avoid redundancy and its impact when interpreting the mutation score. Resistant mutants, despite being killed by a single test case, may turn out to be redundant; this does not hold in resistant hard to kill mutants, so they would be included in a minimal set of mutants.

7. CONCLUSION

In this paper, we presented an evaluation of class mutation operators for the C++ programming language. To that purpose, mutation operators were separately assessed based on their usefulness during test suite evaluation and test suite refinement. In particular, mutation operators were classified into two rankings ordered by the redundancy of their mutants, and the quality of the tests they help to produce respectively. These two rankings share commonalities but also differences which support this novel twofold evaluation. Additionally, both rankings were used as the basis for a selective mutation study showing the trade-off between removing some of the mutants and the loss in the effectiveness of the technique. In practice, the tester might want to select a subset of mutants depending on (1) the goal (evaluation or refinement of the test suite) and (2) the exhaustiveness required in the testing process.

The evaluation results on six open-source applications show that both rankings serve as an accurate reference of the value of each mutation operator. Thus, just using six operators led to an average decrease in the number of mutants of 31.7% without a significant loss of mutation score (2.78%). Similarly, only seven operators were necessary to retain almost 87% of the test cases contained in the adequate and minimal test suite constructed for the full set of mutants (reduction of 39.42% in the number of mutants). Rank-based mutant selection (favouring the analysis of the mutants from the top ranked operators) reported even better results for both evaluations with the same number of mutants: 98.87% (mutation score) and 5.72% (percentage of test cases loss). Therefore, in the case of class mutation operators, selection of mutants in a rank-based manner from all mutation operators has shown to be more beneficial than operator-based selective mutation. The proposed rank-based strategy also outperformed the random technique for the selection of mutants.

The ongoing work includes the definition of new evaluation metrics derived from the ones proposed by Estero-Botaro et al. [20]. When computing the quality metric of a mutant, it would be interesting to know how many mutants from other operators are killed by the test cases killing that mutant (i.e., measure the operator quality in the context of the complete set of mutation operators and not only using the mutants from that operator). Also, including which mutants are covered by which test cases as an additional source of information in the metric can allow for a more accurate assessment. This evaluation applied to other programming languages can provide insights about the

Prepared using stvrauth.cls
DOI: 10.1002/stvr

usefulness of each mutation operator. In addition, we aim at devising an evolutionary approach in order to better estimate the trade-off between a loss in the mutation score or test cases and the reduction of the cost of applying mutation testing, including expenses of equivalent mutant detection.

8. ACKNOWLEDGEMENTS

This paper was partially funded by the research scholarship PU-EPIF-FPI-PPI-BC 2012-037 of the University of Cádiz, and partially supported by the European Commission (FEDER) and Spanish Government projects DArDOS (TIN2015-65845-C3-3-R) and CICYT BELI (TIN2015-70560-R), and the Andalusian Government projects THEOS (TIC-5906) and COPAS (P12-TIC-1867). We also thank Francisco Palomo-Lozano for allowing us to use his algorithm to find minimal test suites and his version of Knuth's algorithm S to select mutants randomly.

REFERENCES

1. Offutt AJ, Untch RH. Mutation 2000: Uniting the orthogonal. *Mutation Testing for the New Century*, *The Springer International Series on Advances in Database Systems*, vol. 24, Wong W (ed.). Springer US, 2001; 34–44, doi:10.1007/978-1-4757-5939-6_7. URL http://dx.doi.org/10.1007/978-1-4757-5939-6_7.
2. King KN, Offutt AJ. A Fortran language system for mutation-based software testing. *Softw. Pract. Exper.* Jun 1991; **21**(7):685–718, doi:10.1002/spe.4380210704. URL http://dx.doi.org/10.1002/spe.4380210704.
3. Agrawal H, DeMillo R, Hathaway B, Hsu W, Hsu W, Krauser E, Martin R, Mathur A, Spafford E. Design of mutant operators for the C programming language. *Technical Report*, Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue University, West Lafayette, Indiana Mar 1989.
4. Ma YS, Kwon YR, Offutt AJ. Inter-class mutation operators for Java. *Proceedings of XIII International Symposium on Software Reliability Engineering*, Kawada S (ed.), IEEE Computer Society: Annapolis (Maryland), 2002; 352–363, doi:10.1109/ISSRE.2002.1173287. URL http://dx.doi.org/10.1109/ISSRE.2002.1173287, ISSRE 2002.
5. Derezińska A. Quality assessment of mutation operators dedicated for C# programs. *Proceedings of VI International Conference on Quality Software*, Kellenberger P (ed.), IEEE Computer Society: Beijing (China), 2006; 227–234, doi:10.1109/QSIC.2006.51. URL http://dx.doi.org/10.1109/QSIC.2006.51, ISSN 1550-6002.
6. Delgado-Pérez P, Medina-Bulo I, Domínguez-Jiménez JJ, García-Domínguez A, Palomo-Lozano F. Class mutation operators for C++ object-oriented systems. *Annals of telecommunications* 2015; **70**(3-4):137–148, doi:10.1007/s12243-014-0445-4. URL http://dx.doi.org/10.1007/s12243-014-0445-4.
7. Offutt AJ, Rothermel G, Zapf C. An experimental evaluation of selective mutation. *Proceedings of 15th International Conference on Software Engineering, 1993*, 1993; 100–107, doi:10.1109/ICSE.1993.346062. URL http://dx.doi.org/10.1109/ICSE.1993.346062.
8. Budd TA, Angluin D. Two notions of correctness and their relation to testing. *Acta Informatica* 1982; **18**(1):31–45, doi:10.1007/BF00625279. URL http://dx.doi.org/10.1007/BF00625279.
9. Jia Y, Harman M. Higher order mutation testing. *Information and Software Technology* Oct 2009; **51**(10):1379–1393, doi:10.1016/j.infsof.2009.04.016. URL http://dx.doi.org/10.1016/j.infsof.2009.04.016.
10. Hussain S. Mutation clustering. Master's Thesis, King's College London 2008.
11. Zhang L, Hou SS, Hu JJ, Xie T, Mei H. Is operator-based mutant selection superior to random mutant selection? *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, ACM: New York, NY, USA, 2010; 435–444, doi:10.1145/1806799.1806863. URL http://dx.doi.org/10.1145/1806799.1806863.

12. Offutt AJ, Lee A, Rothermel G, Untch RH, Zapf C. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.* Apr 1996; **5**(2):99–118, doi:10.1145/227607.227610. URL http://dx.doi.org/10.1145/227607.227610.
13. Barbosa EF, Maldonado JC, Vincenzi AMR. Toward the determination of sufficient mutant operators for C. *Software Testing, Verification and Reliability* 2001; **11**(2):113–136, doi:10.1002/stvr.226. URL http://dx.doi.org/10.1002/stvr.226.
14. Budd TA. Mutation analysis of program test data. PhD Thesis, Yale University 1980.
15. Wong WE, Mathur AP. Reducing the cost of mutation testing: An empirical study. *techreport*, Purdue University, West Lafayette, Indiana 1993.
16. Namin AS, Andrews JH, Murdoch DJ. Sufficient mutation operators for measuring test effectiveness. *ACM/IEEE 30th International Conference on Software Engineering, 2008. ICSE '08*, 2008; 351–360, doi:10.1145/1368088.1368136. URL http://dx.doi.org/10.1145/1368088.1368136.
17. Mresa ES, Bottaci L. Efficiency of mutation operators and selective mutation strategies: an empirical study. *Software Testing, Verification and Reliability* 1999; **9**(4):205–232, doi:10.1002/(SICI)1099-1689(199912)9:4<205::AID-STVR186>3.0.CO;2-X. URL http://dx.doi.org/10.1002/(SICI)1099-1689(199912)9:4<205::AID-STVR186>3.0.CO;2-X.
18. Derezińska A, Rudnik M. Quality evaluation of object-oriented and standard mutation operators applied to C# programs. *Objects, Models, Components, Patterns*, *Lecture Notes in Computer Science*, vol. 7304, Furia C, Nanz S (eds.). Springer Berlin Heidelberg, 2012; 42–57, doi:10.1007/978-3-642-30561-0_5. URL http://dx.doi.org/10.1007/978-3-642-30561-0_5.
19. Bluemke I, Kulesza K. Reduction in mutation testing of Java classes. *9th International Conference on Software Engineering and Applications (ICSOFT-EA), 2014*, 2014; 297–304, doi:10.5220/0004992102970304. URL http://dx.doi.org/10.5220/0004992102970304.
20. Estero-Botaro A, Palomo-Lozano F, Medina-Bulo I, Domínguez-Jiménez JJ, García-Domínguez A. Quality metrics for mutation testing with applications to WS-BPEL compositions. *Software Testing, Verification and Reliability* 2014; doi:10.1002/stvr.1528. URL http://dx.doi.org/10.1002/stvr.1528.
21. Kim S, Clark JA, McDermid JA. The rigorous generation of Java mutation operators using HAZOP. *Proceedings of the 12th International Cofference Software and Systems Engineering and their Applications (ICSSEA 99)*, Paris, France, 1999.
22. Ma YS, Kwon YR, Kim SW. Statistical investigation on class mutation operators. *ETRI Journal* Apr 2009; **31**(2):140–150, doi:10.4218/etrij.09.0108.0356. URL http://dx.doi.org/10.4218/etrij.09.0108.0356.
23. Lee HJ, Ma YS, Kwon YR. Empirical evaluation of orthogonality of class mutation operators. *Software Engineering Conference, 2004. 11th Asia-Pacific*, 2004; 512–518, doi:10.1109/APSEC.2004.49. URL http://dx.doi.org/10.1109/APSEC.2004.49.
24. Segura S, Hierons RM, Benavides D, Ruiz-Cortés A. Mutation testing on an object-oriented framework: An experience report. *Information and Software Technology* 2011; **53**(10):1124–1136, doi:10.1016/j.infsof.2011.03.006. URL http://dx.doi.org/10.1016/j.infsof.2011.03.006, special Section on Mutation Testing.
25. Delgado-Pérez P, Medina-Bulo I, Palomo-Lozano F, García-Domínguez A, Domínguez-Jiménez JJ. Assessment of class mutation operators for C++ with the MuCPP mutation system. *Information and Software Technology* 2016; doi:10.1016/j.infsof.2016.07.002. URL http://dx.doi.org/10.1016/j.infsof.2016.07.002.
26. Estero-Botaro A, Palomo-Lozano F, Medina-Bulo I. Quantitative evaluation of mutation operators for WS-BPEL compositions. *Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW), 2010*, 2010; 142–150, doi:10.1109/ICSTW.2010.36. URL http://dx.doi.org/10.1109/ICSTW.2010.36.
27. Ammann P, Delamaro ME, Offutt J. Establishing theoretical minimal sets of mutants. *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, IEEE Computer Society: Washington, DC, USA, 2014; 21–30, doi:10.1109/ICST.2014.13. URL http://dx.doi.org/10.1109/ICST.2014.13.
28. Matrix TCL Pro, version 2.2. http://www.techsoftpl.com/matrix/download.php. [Online; accessed 14-July-2016].
29. XmlRPC, version 0.7. http://xmlrpcpp.sourceforge.net/. [Online; accessed 14-July-2016].
30. Dolphin. https://www.kde.org/applications/system/dolphin. [Online; accessed 14-July-2016].
31. Tinyxml2. https://github.com/leethomason/tinyxml2. [Online; accessed 14-July-2016].
32. KMyMoney, version 4.6.4. https://sourceforge.net/projects/kmymoney2/. [Online; accessed 14-July-2016].

33. QtDOM. https://github.com/qtproject/qtbase/tree/dev/src/xml/dom. [Online; accessed 14-July-2016].
34. Bluemke I, Kulesza K. *Proceedings of the Ninth International Conference on Dependability and Complex Systems DepCoS-RELCOMEX. June 30 – July 4, 2014, Brunów, Poland*, chap. Reductions of Operators in Java Mutation Testing. Springer International Publishing: Cham, 2014; 93–102, doi:10.1007/978-3-319-07013-1_9. URL http://dx.doi.org/10.1007/978-3-319-07013-1_9.
35. Arcuri A, Briand L. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* May 2014; **24**(3):219–250, doi:10.1002/stvr.1486. URL http://dx.doi.org/10.1002/stvr.1486.
36. Fraser G, Arcuri A. EvoSuite: Automatic test suite generation for object-oriented software. *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, ACM: New York, NY, USA, 2011; 416–419, doi:10.1145/2025113.2025179. URL http://dx.doi.org/10.1145/2025113.2025179.
37. Papadakis M, Henard C, Harman M, Jia Y, Le Traon Y. Threats to the validity of mutation-based test assessment. *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, ACM: New York, NY, USA, 2016; 354–365, doi:10.1145/2931037.2931040. URL http://dx.doi.org/10.1145/2931037.2931040.
38. Mathur AP. Performance, effectiveness, and reliability issues in software testing. *Computer Software and Applications Conference, 1991. COMPSAC '91., Proceedings of the Fifteenth Annual International*, 1991; 604–605, doi:10.1109/CMPSAC.1991.170248. URL http://dx.doi.org/10.1109/CMPSAC.1991.170248.
39. Delamaro ME, Deng L, Li N, Durelli VHS, Offutt AJ. Growing a reduced set of mutation operators. *Brazilian Symposium on Software Engineering (SBES), 2014*, 2014; 81–90, doi:10.1109/SBES.2014.14. URL http://dx.doi.org/10.1109/SBES.2014.14.
40. Zhang J, Zhu M, Hao D, Zhang L. An empirical study on the scalability of selective mutation testing. *IEEE 25th International Symposium on Software Reliability Engineering (ISSRE), 2014*, 2014; 277–287, doi:10.1109/ISSRE.2014.27. URL http://dx.doi.org/10.1109/ISSRE.2014.27.
41. Acree AT Jr. On mutation. PhD Thesis, Atlanta, GA, USA 1980.
42. Zhang L, Gligoric M, Marinov D, Khurshid S. Operator-based and random mutant selection: Better together. *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, 2013; 92–102, doi:10.1109/ASE.2013.6693070. URL http://dx.doi.org/10.1109/ASE.2013.6693070.
43. Gopinath R, Alipour MA, Ahmed I, Jensen C, Groce A. On the limits of mutation reduction strategies. *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, ACM: New York, NY, USA, 2016; 511–522, doi:10.1145/2884781.2884787. URL http://dx.doi.org/10.1145/2884781.2884787.
44. Jia Y, Harman M. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on* Oct 2011; **37**(5):649 –678, doi:10.1109/TSE.2010.62. URL http://dx.doi.org/10.1109/TSE.2010.62.
45. Domínguez-Jiménez JJ, Estero-Botaro A, García-Domínguez A, Medina-Bulo I. Evolutionary mutation testing. *Information and Software Technology* Oct 2011; **53**(10):1108–1123, doi:10.1016/j.infsof.2011.03.008. URL http://dx.doi.org/10.1016/j.infsof.2011.03.008.
46. Namin AS, Xue X, Rosas O, Sharma P. MuRanker: a mutant ranking tool. *Software Testing, Verification and Reliability* 2015; **25**(5-7):572–604, doi:10.1002/stvr.1542. URL http://dx.doi.org/10.1002/stvr.1542.
47. Schuler D, Zeller A. Javalanche: Efficient mutation testing for Java. *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, ACM: New York, NY, USA, 2009; 297–298, doi:10.1145/1595696.1595750. URL http://dx.doi.org/10.1145/1595696.1595750.
48. Kusano M, Wang C. CCmutator: A mutation generator for concurrency constructs in multithreaded C/C++ applications. *IEEE/ACM 28th International Conference on Automated Software Engineering (ASE), 2013*, IEEE, 2013; 722–725, doi:10.1109/ASE.2013.6693142. URL http://dx.doi.org/10.1109/ASE.2013.6693142.
49. Moghadam M, Babamir S. Mutation score evaluation in terms of object-oriented metrics. *4th International eConference on Computer and Knowledge Engineering (ICCKE), 2014*, 2014; 775–780, doi:10.1109/ICCKE.2014.6993419. URL http://dx.doi.org/10.1109/ICCKE.2014.6993419.
50. Just R, Kapfhammer G, Schweiggert F. Do redundant mutants affect the effectiveness and efficiency of mutation analysis? *IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST), 2012*, 2012; 720–725, doi:10.1109/ICST.2012.162. URL http://dx.doi.org/10.1109/ICST.2012.162.

51. Wright C, Kapfhammer G, McMinn P. The impact of equivalent, redundant and quasi mutants on database schema mutation analysis. *14th International Conference on Quality Software (QSIC), 2014*, 2014; 57–66, doi: 10.1109/QSIC.2014.26. URL http://dx.doi.org/10.1109/QSIC.2014.26.
52. Yao X, Harman M, Jia Y. A study of equivalent and stubborn mutation operators using human analysis of equivalence. *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, ACM: New York, NY, USA, 2014; 919–930, doi:10.1145/2568225.2568265. URL http://dx.doi.org/10.1145/2568225.2568265.

DOI: 10.1002/stvr