SiMut: Exploring Program Similarity to Support the Cost Reduction of Mutation Testing

Alessandro V. Pizzoleto, Fabiano C. Ferrari
Computing Department
Federal University of São Carlos
São Carlos, SP, Brazil
{alessandro.pizzoleto, fcferrari}@ufscar.br

Lucas D. Dallilo
Department of Computer Systems
University of São Paulo
São Carlos, SP, Brazil
lucasdallilo@usp.br

Jeff Offutt
Department of Computer Science
George Mason University
Fairfax, VA, USA
offutt@gmu.edu

***Abstract*—Scientists have created many cost reduction techniques for mutation testing, and most of them reduce cost with minor losses of effectiveness. However, many of these techniques are difficult to generalize, difficult to scale, or both. Published results are usually limited to a modest collection of programs. Therefore, an open question is whether the results of a given cost reduction technique on programs studied in the paper will hold true for other programs. This paper introduces a conceptual framework, named *SiMut*, to support the cost reduction of mutation testing based on historical data and program similarity. Given a new, untested program u, the central idea is applying to u the same cost reduction strategy applied to a group G of programs that are similar to u and have already been tested with mutation, and check for consistency of results in terms of reduced costs and quality of test sets. *SiMut* includes activities to compute program abstractions and similarity. Based on this information, it supports the application of mutation cost reduction techniques to both G and u. This paper presents the concepts behind *SiMut*, a proof-of-concept implementation of *SiMut*, and results from a pilot study. Finally, we discuss some issues related to the use of *SiMut*, focusing on the composition of a representative dataset to properly explore the potential of our framework.**

***Index Terms*—mutation testing; cost reduction; program similarity; reuse of historical data**

I. Introduction

Mutation testing [14] (or simply *mutation*) helps testers create high quality test sets. It takes a program p and systematically generates slightly modified versions of p, called *mutants*. *Mutation operators* $O = \{o_1, o_2, ... , o_n\}$ are used to create mutants. Ideally, each mutant is a faulty program and should behave differently from p for at least one test case. When such a test is found, the mutant is said to be *killed* by the test case. Until then, the mutant remains *alive* and either a test case must be found or designed to kill the mutant, or the mutant should be determined to be equivalent. An *equivalent mutant* behaves the same as p on all tests, so cannot be killed. The *mutation score* (MS) is a value in the interval $[0, 1]$ that represents the quality of the test set with respect to a given group of mutants. The closer MS is to 1, the better the test set is. The MS formula MS = K/(M-E) uses the dead (K), equivalent (E), and the total number of mutants (M) generated. We also consider mutants generated by a single operator o_i. A set of tests that kills all o_i mutants is said to be o_i*-adequate*, and the mutation score for o_i is the score of the o_i-adequate tests on all mutants created for the program.

Empirical studies have found that mutation yields tests that detect more faults than tests derived with most other structural criteria [7, 18, 27, 30, 31], and that can reveal real faults [3, 13, 22]. Mutation is also very often used as a test quality assessment instrument to evaluate tests derived using other testing techniques and criteria. For example, as recently highlighted in a literature review [40] that selected 502 mutation-related studies, all published between 2008 and 2017, 217 used mutants only for test assessment purposes.

Although effective, mutation is also expensive [21, 24, 40, 42]. Two major factors affecting the cost are (i) the large number of mutants (even for small programs), and (ii) the manual analysis required for unkilled mutants. Since detecting equivalent mutants is generally undecidable [5], equivalence detection can only be partially automated [29, 35, 39]. These factors have limited the adoption of mutation in industry, leading to almost four decades of research into reducing the cost of mutation, as reported in recent literature surveys [40, 42].

Experimental results indicate that many mutation cost reduction techniques reduce costs with minor losses of effectiveness [42], however many techniques are difficult to generalize, difficult to scale, or both. For example, Kurtz et al. [26] found significant cost savings, but only if each program is analyzed individually. Thus, a significant open question is whether the results of *any cost reduction technique* on previous programs will hold true for new programs.

For example, *selective mutation* [4, 36] defines an empirically determined subset of mutation operators, a *sufficient* set O_s, that dramatically reduces the number of mutants (and thereby the cost) without reducing the test quality. However, Kurtz et al. [26] found that the ideal sufficient set is different for each program. This problem can be generalized in terms of particular sets of programs, or even in terms of technologies. For instance, while investigating the ability of mutation to reveal real faults, Siami-Namin and Kakarla [46] concluded that "*the major question to address is whether the researchers need to apply exactly the same set of operators or the same number of mutants as used*" in previous experiments, "*for any other experimentations regardless of underlying programming languages*." Papadakis et al. [40] also observed that "*when using mutation testing, it is important to carefully select* *mutant operators that are appropriate to the programming language studied*".

We are currently investigating whether we can establish similarities among programs such that the most effective sufficient set of mutation operators for a prior program will likely also be an effective sufficient set for a similar program. In a previous paper [12], we reported results from an experiment with a small set of 38 Java programs. We *clustered* programs based on internal metrics, and asked whether sufficient sets of mutation operators that were previously determined to be effective for programs in the cluster are also effective for new programs in the cluster. That is, can internal metrics be used to determine similarity among programs that can be used to predict an effective sufficient set of mutation operators. Notice that such a program similarity-based approach can use any cost reduction technique that relies on empirical training, including but not limited to, supervised Machine Learning algorithms and operator-based mutant selection,

This paper extends our prior paper in three ways. It:

1) Externalizes a conceptual framework, named *SiMut*, to support the cost reduction of mutation testing based on historical data and program similarity. *SiMut* uses varied program abstractions beyond internal metrics, varied program similarity calculations beyond clustering, and the application of varied cost reduction techniques.
2) Presents a proof-of-concept implementation of the main activities defined in *SiMut*, as well as results from a pilot study with a dataset that is larger than in our prior work. The artifacts included in the database were used and produced in prior experiments with mutation testing.
3) Discusses key issues that should be handled while composing a dataset to apply the approach embedded in *SiMut*.

We analyzed three extensive literature reviews on mutation testing [21, 40, 42]. Two [21, 40] summarized mutation-related research published from 1977 to 2017; they account for more than 800 analyzed studies, and provided a broad picture of the area. The other [42] analyzed 175 studies and focused on characterizing goals and techniques for cost reduction, including used metrics and achieved results. None of studies analyzed in the reviews had goals similar to ours. Therefore, to the best of our knowledge, this is the first approach that combines the use of information from previously tested programs and program similarity to support the testing of new programs using mutation at reduced cost.

This section summarizes the context, motivation, goals and contributions of this paper. Basic background is presented in Section II. Section III describes the *SiMut* framework, Section IV describes our implementation, and Section V reports on results of a pilot study. Section VI discusses key issues related to the application of *SiMut*, as well as limitations of this work. Section VII summarizes related work. Finally, Section VIII brings our conclusions and points out future work.

II. Background

This section describes basic background on program abstractions and program similarity. Both concepts are used in the *SiMut* framework, described in Section III.

A. Program Abstractions

Program abstraction is used to simplify the process of analyzing complex programs. The abstraction tool creates a relationship between a concrete program and an abstract program [10]. We summarize various forms of program abstractions, including structural graphs, sequence diagrams, obfuscated source code, simplified source code, and static and dynamic internal metrics.

In the context of program abstractions, our two experiments (one from our prior work [12] and another here) used the Chidamber and Kemerer (CK) static internal metrics [9] as program abstractions. We then used them to compute similarity among programs.

B. Program Similarity

The literature has several definitions for program similarity. For example, in a comparison study of software clone detection techniques, Roy et al. [45] defined two types of similarity between code fragments, one based on text (similar program code), and based the other on functionality (similar program behavior).

Another definition comes from Walenstein et al. [49]. The definitions are similar to the ones proposed by Roy et al. [45]. Walenstein et al. conceptually defined two types of similarity: behavioral (semantic) and representational (syntactic). Semantic similarity compares the meaning not the structure. Examples are the comparison of program input and output, and the program execution with searching for statements correspondence (in relation to the order of statement executions). Syntactic similarity, on the other hand, compares the structure not the meaning, in different levels of abstraction, from code blocks to architectural levels.

Representational similarity can be computed based on statically computed internal software metrics, which map code features or structures to the domain of real numbers. We use this technique in this paper, as described in Section V. Another example is "feature-based" similarity, which determines the similarity between programs using the number of feature matches or properties identified in the programs.

III. The *SiMut* Framework

This section describes the conceptual framework named *SiMut* (Program **Si**milarity to support Cost Reduction of **Mut**ation). *SiMut* supports the cost reduction of mutation testing based on historical data and on program similarity as a core element. Is is comprised of a main process (*overall process*) and a sub-process that is responsible for composing groups of similar programs. Both processes, and some usage scenarios, are described below.

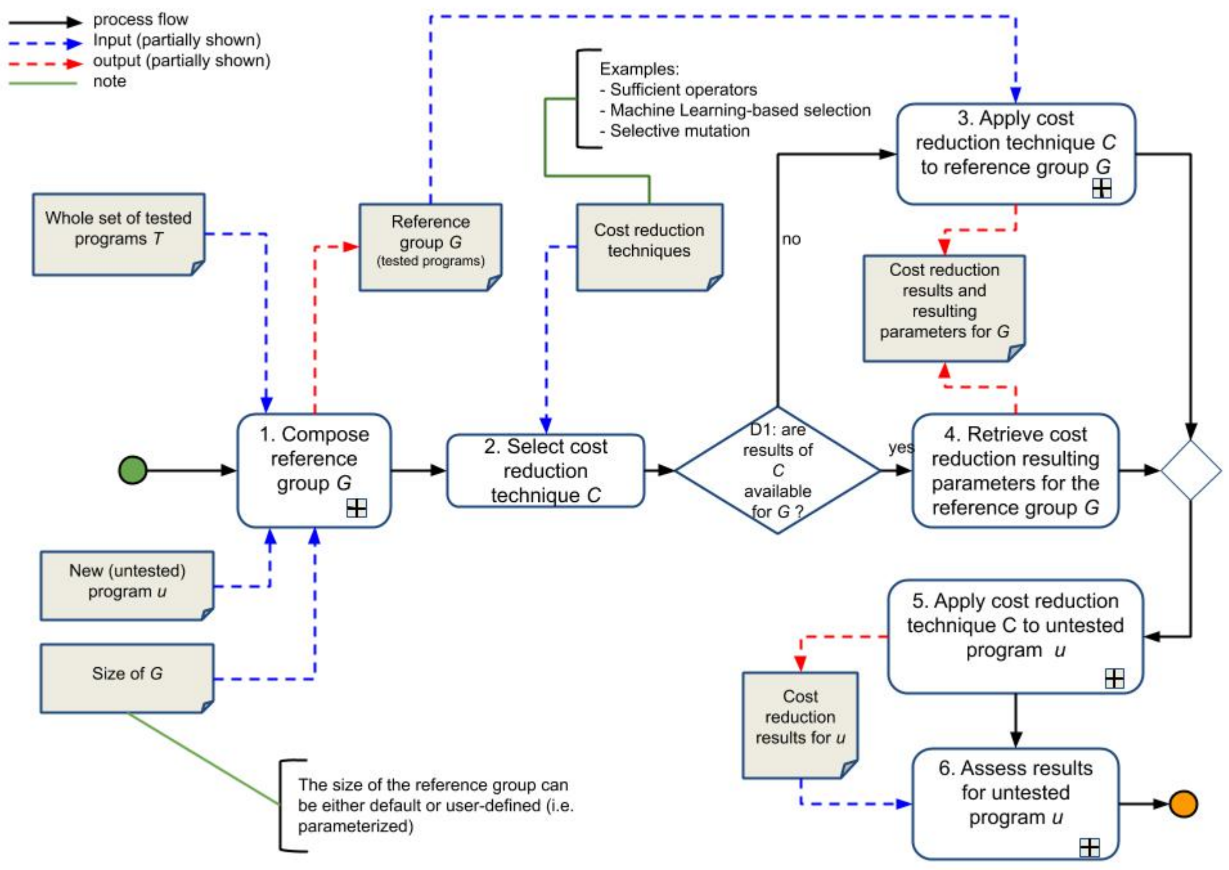
process flow
Input (partially shown)
output (partially shown)
note
Examples:
- Sufficient operators
- Machine Learning-based selection
- Selective mutation
Whole set of tested programs T
Reference group G (tested programs)
Cost reduction techniques
3. Apply cost reduction technique C to reference group G
no
Cost reduction results and resulting parameters for G
1. Compose reference group G
2. Select cost reduction technique C
D1: are results of C available for G ?
yes
4. Retrieve cost reduction resulting parameters for the reference group G
New (untested) program u
Size of G
5. Apply cost reduction technique C to untested program u
Cost reduction results for u
6. Assess results for untested program u
The size of the reference group can be either default or user-defined (i.e. parameterized)

Fig. 1. Overall process.

A. *Overall Process*

Figure 1 represents the overall process of *SiMut*. We use Business Process Model and Notation (BPMN) to represent the process. Key items (*e.g.* sub-processes, tasks, inputs, output, and additional notes) are described next.

1. Compose reference group G: This sub-process composes a group G of tested programs that are similar to a new (*u*ntested) program u. A *tested program* has a mutation-adequate test suite. The number of programs in G can be either user-defined or default (*e.g.*, defined by the similarity algorithm).

- Inputs: A set T of programs tested with mutation, the untested program u, and (optional) the size of G.
- Output: The reference group G.
- Notes: For more details, see the *Composition of the Reference Group (G)* process in Section III-B.

2. Select cost reduction technique C: A cost reduction technique C to apply to G is chosen from existing techniques such as *selective mutation* [36], *optimization of analysis of mutants* based on Machine Learning algorithms [6, 8], *sufficient operators* [4] and *minimal mutation* [2]. Users decide which technique to use. If results for C applied to G are already available, they are retrieved (process 4). Otherwise, they are calculated using process 3.

- Input: A collection of available cost reduction techniques.
- Output: The selected cost reduction technique C.

3. Apply cost reduction technique C to reference group G: This sub-process applies C to the programs in G. The resulting measures R_G and the resulting cost reduction parameters S are stored. Examples of parameters are a sufficient set of mutation operators or a trained Machine Learning algorithm. Parameters S will be next applied to the untested program u.

- Input: The reference group G and the selected cost reduction technique C.
- Output: The cost reduction results R_G and the cost reduction resulting parameters S.

4. Retrieve cost reduction resulting parameters S for the reference group G.

5. Apply cost reduction technique C to untested program u: The parameters S from applying C to the reference group G are used to apply C to the new program u.

- Inputs: The untested program u, the selected cost reduction technique C, and the cost reduction parameters S.

- Output: R_u, the results of applying C to u based on S.

6. Assess results for untested program u: Here we evaluate the results R_u. The evaluation includes the mutation score produced by S-adequate tests when all mutants are considered. The evaluation can also compare cost reduction savings with respect to the reference group G.

- Inputs: The cost reduction results R_u, all mutants, and (optional) the cost reduction results R_G.
- Output: Varied (depends on the evaluation goals).

Notes about the overall process: Sub-process 6 is for experimental purposes only. This step would not be executed when *SiMut* is used in practice.

B. Choosing Programs in the Reference Group

This section describes how programs are chosen for the reference group G, as illustrated in Figure 2.

1.1. Computing measures for u: We need to apply the same measures to u that will be used to calculate similarity. Measures that may be calculated (depending on current experimental goals) include internal program metrics [9], control flow graphs [34], and obfuscated source code. We refer to these as *pre-processed abstractions* in the rest of the paper.

- Input: The untested program u.
- Output: A set of measures A_u for u.
- Notes: We assume that the same measures have already been gathered for programs in T. The process of gathering such measures is the same as process #1.1 described previously.

1.2. Select similarity approach S: Next we select a similarity calculation approach S to apply to u with respect to the tested programs T. Example similarity calculations include clustering (applied in the exploratory study presented in Section V), lexical distance calculation, and information diversity calculation.

- Input: A set of available similarity calculation approaches.
- Output: The selected similarity calculation approach S.

1.3. Apply approach S: Next we calculate similarity between u and T to determine the reference group G of programs that are similar to u.

- Inputs: The selected similarity calculation approach S, the set of measures A_u for u, the sets of measures for the programs included in the set of tested programs T, and (optional) the size of G.
- Output: A preliminary version of the reference group G (this group may be refined in the next process).

1.4. Generate reference group G: G is a subset of the programs in T that will be used in the overall process described in Section III-A. G contains programs from T that are most similar to u.

- Input: The selected similarity calculation approach S; the initial reference group G; and the size of G.
- Output: The final reference group G.

Notes about the process for computing G: As noted in sub-process 1.1, the composition of the reference group requires that the same measures gathered for u have already been gathered for programs in T. This information must be available to be retrieved and used in sub-process 1.3 (*Apply approach S*).

C. Examples of Usage Scenarios

This section illustrates the use of *SiMut* with two scenarios, each applying a different mutation cost reduction technique. The first uses *sufficient operators* [4]. The second uses *optimization of mutant analysis* through automatic mutant classification based on Machine Learning (ML) algorithms [6, 8]. Both scenarios require the execution of particular paths of the process depicted in Figure 1. A third technique, *one-op mutation* [48], was used in the study described in Section V.

Scenario 1—Mutant classification based on sufficient operators*:* This scenario computes a sufficient set of mutation operators for u using a procedure such as the *Sufficient* procedure by Barbosa et al. [4]). The sufficient set of operators will be obtained based on the reference group G using the following steps:

- Sub-process 1: Compose the reference group G of similar programs.
- Task 2: Select the cost reduction technique C (sufficient operators).
- Decision D1: Answer is no (results for C applied to G are not available).
- Sub-process 3: Apply technique C is to G, getting results R_G and cost reduction parameters S (sufficient operators).
- Sub-process 5: Apply the sufficient operators identified for G to the untested program u, obtaining results R_u.
- Sub-process 6 (optional): Assess results R_u, for example, with respect to test set quality (full mutation score), and with respect to R_G.

Scenario 2—Mutant classification based on machine learning algorithms: The tester reuses a trained ML algorithm that automatically classifies mutants based on their utility. An example approach is by Chekam et al. [6, 8], who applied supervised learning to train an algorithm to automatically classify mutants based on their probability of revealing faults. This scenario assumes a reference group G that was already used to train the ML algorithm. The algorithm parameters are retrieved and reused to classify mutants of the untested program u using the following steps:

- Sub-process 1: Compose the Reference group G of similar programs.
- Task 2: Select the cost reduction technique C (mutant classification based on ML).
- Decision D1: Answer is yes (results for C applied to G are available).
- Sub-process 4: Retrieve the parameters S from the trained ML algorithm.
- Sub-process 5: Apply the ML-based classifier to mutants of the untested program u, obtaining results R_u.

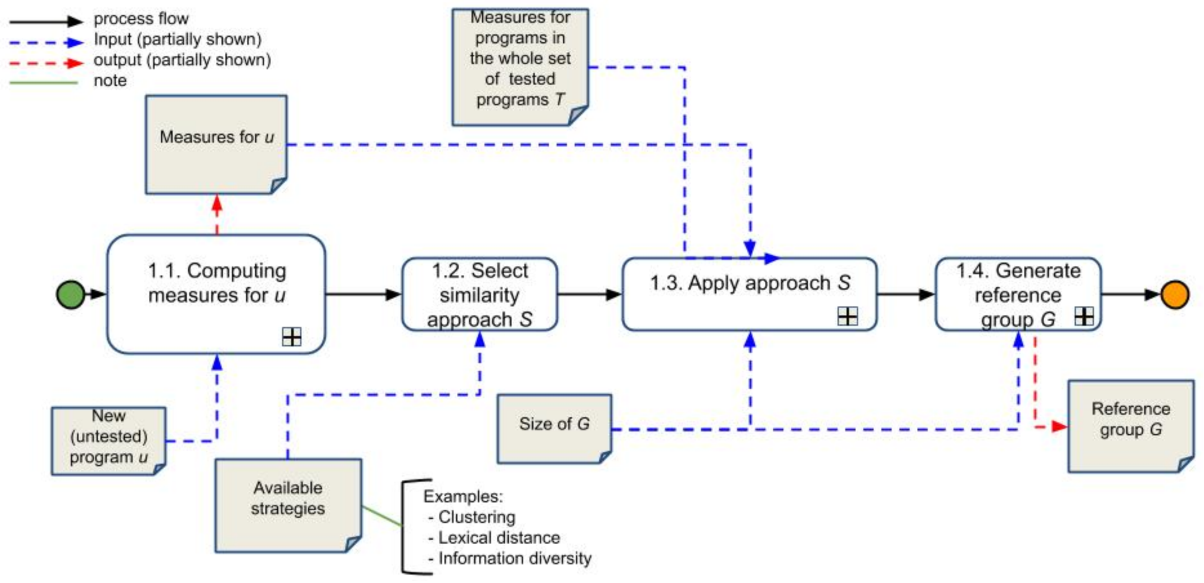

Fig. 2. Composition of the reference group of tested programs.

- Sub-process 6 (optional): Assess results R_u, for example, with respect to precision of the classifier, and with respect to the full mutation score produced by test cases that are adequate for the classified mutants.

IV. Proof-of-Concept Implementation

This section describes a proof-of-concept implementation of *SiMut*. The prototype includes three main modules, *Metrics Collector*, *Clustering Calculator*, and *Mutation Score Calculator*. It is available at https://doi.org/10.6084/m9.figshare.11787474.v1. Metrics Collector and Clustering Calculator are Java programs that use an external tool to collect metrics and an API to cluster information. Mutation Score Calculator is written in SQL and C#. Details are provided next.

Metrics Collector: We use Analizo[1] [47] to calculate metrics. Analizo computes 10 project level metrics and 16 class level metrics, including the CK metrics [9]. Metrics were gathered in three steps. First, the programs to be analyzed are identified and stored in a file system. Second, the metrics are collected and stored in memory for each source file identified in step one. Third, the metrics are stored in a relational database.

Clustering Calculator: The clustering process runs in four steps. First, all values from the Collector module are normalized in a linear fashion and placed into the interval [0, 1]. Second, the normalized values are formatted for the X-Means algorithm [41]. Third, we use the X-means algorithm, as implemented in the Weka API [17], to produce the clusters. Fourth, the results are presented in terms of the clusters generated and which programs belong to each cluster.

Mutation Score Calculator: Program mutation usually finds a test that kills a mutant, then does not run that mutant against subsequent tests. However, we need to know all tests that kill all mutants. This information allows us to create a complete *killing matrix* for each program, where rows represent all the mutants, columns represent all the tests, and a cell is marked if that test killed that mutant.

We use a database of mutation-related artifacts (described in Section V-B) to identify mutants generated by each operator, and calculate mutation scores for each operator. For the experiment described in Section V, the mutation scores of each operator were calculated for the programs separately, for each cluster created, and for the entire set of programs. This allowed us to determine the best mutation operator (One-Op) for each option (program, cluster, and total set). Note that our database includes information about mutants and tests that kill them; therefore, we do not re-run the mutants on the test sets.

V. Pilot Study

We ran a pilot study to demonstrate the feasibility of our process, as well as to illustrate how results can be interpreted. At this point, we do not analyze statistical significance because of limitations on the data, as discussed in Section VI-A.

A. Study Overview

Our research goal is to determine if internal program information (as measured by internal metrics) can be used to predict mutation testing results. More specifically, for an untested program u, we evaluate whether the best mutation operators for programs similar to u are also the best operators for u as well. We measure operators based on their mutation scores. This study uses the same procedures as in our previous study [12], but considers a substantially larger database of mutation-related artifacts.

This study followed the *SiMut* process from Section III. We selected CK metrics as the program abstraction, clustering as

[1]http://www.analizo.org/, accessed in February, 2020.

TABLE I
SUMMARY OF ARTIFACTS INCLUDED IN THE DATABASE.

Source	# Programs	# Tests	# Mutants	# Operators	Operators	Tool
[15]	29	18	4,161	15	LOI; SDL; AOIS; VDL; COI; AOIU; ROR; AOR; ODL; CDL; COR; ASRS; AODU; AODS; COD	muJava
[38, 43]	9	303	1,347	10	AOIS; AOIU; AOR; ROR; LOI; AODS; COR; AODU; COD; COI	muJava
[16]	13	11	2,418	18	JID; JSD; JSI; AOIS; AOIU; AOR; ASRS; COI; LOI; ROR; EAM; JDC; JTD; JTI; IOD; AODU; COD; COR	muJava
[37]	21	105	10,147	30	EAM; EOC; IOD; IOR; IPC; ISI; JSD; JSI; PCI; OMR; PCC; ISD; PRV; OAN; PMD; PPD; EMM; IOP; PCD; JID; ROR; UOI; SWA; LCR; NUF; ABS; TEX; AOR; DEC; INC	muJava
[19]	211	934	19,851	10	DEC; SWA; AOR; ABS; INC; UOI; NUF; ROR; LCR; TEX	Bacterio

the similarity calculation technique, and one-op [48] as the mutation cost reduction technique.

B. Subject Programs

We used Java programs from previous mutation-related research. We used our previous Systematic Literature Review [42] to find subject programs that are available online. We also contacted all authors of studies that used Java programs to request access to the artifacts from their experiments.

After analyzing these materials, we had a collection of artifacts that included programs, mutants, mutation operators used, test cases, and information on which tests killed which mutants. We had to omit artifacts that did not have enough information for our research.

Table I summarizes our subject artifacts, as drawn from six prior experiments [15, 16, 19, 37, 38, 43]. These artifacts are available at https://tinyurl.com/mutation2020.

The database includes 283 Java programs, most of which have a single class. We used 37,924 first-order mutants generated from the mutation operators listed in the *Operators* column by the muJava [33] and Bacterio [44] tools. We also used 1,371 test cases, which are linked to the mutants they killed. The studies indicated the tests were mutation-adequate, so mutants not killed are assumed to be equivalent. Disregarding duplicate programs, the database includes 218 items.

As shown in Table I, the 15 mutation operators in muJava were applied to the 29 programs in study [15]. However, the next row shows that only 10 muJava operators were applied to the 9 programs. In general, different programs and different studies might use different mutation operators within the same dataset. This issue imposes restrictions to the reuse of historical information and is further discussed in Section VI.

C. Used Metrics

Chidamber and Kemerer proposed six metrics for object-oriented software [9]. Table II defines the metrics, and Table III shows the values gathered using the Analizo tool for some of the programs[2]. Given that most programs only have a single class, we determined that only the ACCM and RFC metrics were meaningful. The other CK metrics are related to class hierarchies and structures that are not common in our subject programs. Impacts of this 2-value (ACCM and RFC) representation of each program is discussed in Section VI.

[2]Metrics for other programs are omitted for space but included in the repository https://tinyurl.com/mutation2020.

D. Program Sets and Clusters

We separated the programs into three sets, based on which mutation operators were applied to them. This allowed us to combine programs used in previous studies, and to create groups of programs that are homogeneous in terms of applied mutation operators. For instance, consider a set of programs $B = \{P_1, P_2, P_3, P_4, P_5\}$. If the LOI and AOR operators were applied to P_1, P_2 and P_5, and the AOR and ROR operators were applied to P_1, P_3 and P_4, we created two sets of programs $B_1 = \{P_1, P_2, P_5\}$ and $B_2 = \{P_1, P_3, P_4\}$. Then, for B_1 we considered mutants generated by LOI and AOR, and for B_2 we considered mutants generated by AOR and ROR. The sizes of the program sets are shown in the second column of Table IV.

The numbers of programs per cluster formed by the X-Means algorithm are shown in the *# Programs per Cluster* columns in Table IV. Two clusters were formed for B_1, 3 for B_2, and 5 for B_3. The numbers of mutation operators applied to each program set are shown in the *# Operators* column and the mutation operators are listed in the *Operators* column.

E. Summary of Results

Table V summarizes the results of this pilot study. $R1$ and $R2$, defined below, may have positive (*V*), negative (*X*), or neutral results (*N*). Note that, to achieve such results, from each set (B_1, B_2, B_3), we took each program as an untested program u (for experimental purposes only), and computed the best mutation operators for u, for the cluster to which u is associated with (except u), for the other clusters formed for the program set (called *remaining clusters*), and for all programs in the program set.

- $R1$: We compute $R1$ to evaluate if the cluster of similar programs is a good predictor of the best mutation operator for an untested program u, when compared to clusters of programs that are less similar to u. O_u is the set of best operators for u, and OC_i is the set of best operators for each cluster i. $R1$ evaluates the intersection of O_u and OC_i, considering the cluster that u is associated with, and the remaining clusters formed from the program set.
- $R2$: We next compute $R2$ to evaluate if the cluster of similar programs is a good predictor of the best mutation operator for an untested program u, when compared to the entire set of programs. OP_u is the set of best operators for the entire set of programs. $R2$ evaluates the intersection between the sets O_u and OP_u, then compares this result

TABLE II
CK METRICS COLLECTED BY ANALIZO.

ACCM (*Average of Cyclomatic Complexity per Method*):	Used as substitute to WMC (Weighted Methods per Class), consists of summing up all separate paths within a method (cyclomatic complexity proposed by McCabe [32]), and then summing up all method-specific complexities.
CBO (*Coupling between Object Classes*):	Corresponds to the number of interactions of a chosen class with other classes at the level of methods and variables.
DIT (*Depth of Inheritance Tree*):	Measures the depth of the inheritance tree of a given class.
LCOM4 (*Lack of Cohesion of Methods version 4*)	The methods are organized in pairs, say $m1$ and $m2$. The number of class attributes that $m1$ and $m2$ have in common are counted, then the number of class attributes $m1$ and $m2$ do not have in common are counted. LCOM4 is the difference between those counts.
NOC (*Number of Children*):	Count the number of subclasses of a class.
RFC (*Response for Class*):	The number of methods inside the class added to the number of external methods used by the class.

TABLE III
CK METRICS FROM A SUBSET OF PROGRAMS.

			CK Metrics					
Source	Program	LOC	ACCM	CBO	DIT	LCOM4	NOC	RFC
[38, 43]	Bisec	24	2	0	0	3	0	5
[38, 43]	BubCorrecto	35	1.57	0	0	2	0	14
[38, 43]	Find	44	3.25	0	0	1	0	11
[38, 43]	Fourballs	28	2.5	0	0	1	0	11
[38, 43]	Mid	41	1.44	0	0	4	0	18
[38, 43]	Triangulo	54	3.6	0	1	2	0	16
[38, 43]	PluginTokenizer	103	1.62	0	1	6	0	30
[38, 43]	Ciudad	262	2.65	0	0	1	0	55
[38, 43]	IgnoreList	36	2.66	0	0	1	0	7
[15]	BoundedQueue	78	2	0	0	1	0	26
[15]	countPositive	63	3.5	0	0	2	0	2
...								

TABLE IV
PROGRAM SETS AND CLUSTERS.

Program	#	# Programs per Cluster					#	Operators
Sets	Programs	C_1	C_2	C_3	C_4	C_5	Operators	
B_1	36	17	19				10	AOIS; AOIU; AOR; CDL; COI; COR; LOI; ROR; SDL; VDL
B_2	82	47	29	6			10	ABS; AOR; DEC; INC; LCR; NUF; ROR; SWA; TEX; UOI
B_3	120	22	74	13	3	8	5	AOR; LOI; NUF; ROR; VDL

TABLE V
SUMMARY OF RESULTS.

Results	Program Sets		
	B_1	B_2	B_3
$R1$	V(14), N(10), X (12)	V(0), N(30), X(52)	V(0), N(90), X(30)
$R2$	V(14), N(22), X (0)	V(0), N(82), X(0)	V(1), N(110), X(9)

with OC_u, which is the set of best operators for the cluster u is associated with.

For both $R1$ and $R2$, positive (P) and neutral (N) results favor the similarity approach, given that the cluster was a predictor at least as good as the other clusters, or at least as good as the entire set of programs. Note that positive results regarding $R2$ motivate the evolution of the research, since one might consider just a subset of programs to calculate cost reduction parameters for applying a particular mutation cost reduction technique.

In this pilot study, for B_1 the clusters were good predictors of best mutation operators in 67% (24/36) of the cases when compared with the other clusters, and in 100% (36/36) of the cases when compared with all programs. These percentages were 37% (30/82) and 100% (82/82) for B_2, and 75% (90/120) and 93% (111/120) for B_3.

As we are trying to show feasibility as well as describing how the approach works via example, we did not compute significance in this pilot study, nor compare results with our prior work [12]. Next we discuss issues that should be handled while identifying or finding artifacts for this type of experiment or when applying this approach in practice.

VI. DISCUSSIONS AND LIMITATIONS

This section discusses issues related to the selection and preparation of artifacts and techniques to be used in an approach such as the one introduced by *SiMut*. We highlight three issues: (i) how representative the set of programs in the population is, (ii) how comprehensive the mutation operator set is, and (iii) the tools used to generate the mutants.

(i) How representative the set of programs in the population is: We used programs that had been used in previous studies [15, 16, 19, 37, 38, 43]. Despite the relatively large number of programs (218), they are small and simple, and most did not use OO-specific constructs such as class hierarchies and composition. We only used programs that came with mutants whose operators were identified and test cases that killed all non-equivalent mutants. These constraints make it

difficult to create a representative population of programs, and it is unlikely our programs were. In the future, we hope to either select metrics that are appropriate for the population of programs, or create a larger and more representative population of programs.

(ii) How comprehensive the mutation operator set is: Questions such as "*Are there enough operators?*", or "*Are there too few operators?*" should be considered when selecting the cost reduction technique to use. The set of operators should be aligned with the intent of the technique. For instance, if using sufficient operators [4], we should analyze many operators. In our pilot study, the programs used relatively few operators, making it harder to explore different scenarios, for example, trying subsets of sufficient operators of varying sizes. In this study, the same few operators (ROR and SDL) dominated.

(iii) The tools used to generate mutants: The mutation operators implemented in each tool vary, and sometimes vary in different releases of the same tool. The details of how some operators are implemented also vary, depending on design and implementation decisions.

In this context, Kintis et al. [25] compared the operators from three mutation tools for Java programs: PIT, muJava and Major. For example, consider Table VI. PIT implements the Math operator (M) as described in the first row. muJava and Major implements arithmetic mutation using several individual operators (AORB, SOR, and LOR in muJava, and AOR, SOR and LOR in Major). Kintis et al. [25] also found differences in the same operator. For example, conditional operator replacement (COR) for muJava and Major is illustrated in Table VI.

TABLE VI
MUTATION OPERATORS IMPLEMENTED IN DIFFERENT TOOLS (ADAPTED FROM KINTIS ET AL. [25])

Operator		Description
PIT	M	$\{(op1, op2) \mid (op1, op2) \in \{(+,-),(-,+),(*,/),$ $(/,*),(\%,*),(\&,\mid),(\mid,\&),(\wedge,\&),(<<,>>),$ $(>>,<<),(>>>,<<)\}\}$
muJava	AORB	$\{(op1, op2) \mid op1, op2 \in \{+,-,*,/,\%\} \wedge op1 \neq op2\}$
	SOR	$\{(op1, op2) \mid op1, op2 \in \{>>,>>>,<<\} \wedge op1 \neq op2\}$
	LOR	$\{(op1, op2) \mid op1, op2 \in \{\&,\mid,\wedge\} \wedge op1 \neq op2\}$
	COR	$\{(op1, op2) \mid op1, op2 \in \{\&\&,\mid\mid,\wedge\} \wedge op1 \neq op2\}$
Major	AOR	$\{(op1, op2) \mid op1, op2 \in \{+,-,*,/,\%\} \wedge op1 \neq op2\}$
	SOR	$\{(op1, op2) \mid op1, op2 \in \{>>,>>>,<<\} \wedge op1 \neq op2\}$
	LOR	$\{(op1, op2) \mid op1, op2 \in \{\&,\mid,\wedge\} \wedge op1 \neq op2\}$
	COR	$\{(\&\&, op2), (\mid\mid, op2) \mid op1 \in \{==, LHS, RHS, false\}$, $op2 \in \{!=, LHS, RHS, true\}$

These differences in mutation operators may mean that effective tests for mutants created by one tool may not be effective for mutants created from the same operators in a different tool. Thus, the tools used to create the mutants in the program population must be carefully considered.

A. Limitations

SiMut is a conceptual framework that currently has one proof-of-concept implementation. The *SiMut* framework has three variables: (1) the program abstraction, (2) the similarity calculation, and (3) the cost reduction technique. Our implementation and pilot study used internal metrics for (1), clustering for (2), and one-op mutation for (3). Other configurations can be used for more extensive experimentation.

SiMut can also be expensive to use. Specific costs include executing *SiMut*'s sub-processes to generate the reference group of programs G, and computing the cost reduction technique for G. However, computing cost reduction for an entire set of programs, and obtaining cost reduction parameters to apply to a new, untested program may be even more costly.

VII. Related Work

Despite a comprehensive analysis of the literature on software testing [21, 40, 42], we were not able to find prior approaches that use historical information to reduce the cost of applying mutation testing. This section describes recent papers that explore program similarity to automatically classify equivalent and redundant mutants [23, 24, 39], use internal program characteristics to reduce the cost of mutation testing [6, 8], and to correlate such characteristics with program testability [11].

Several authors used program similarity to automatically classify equivalent [23, 24, 39] and redundant mutants [24, 39]. At the binary code level, Papadakis et al. [39] developed *Trivial Compiler Equivalence* (TCE), which uses a simple diff program to check equivalence between an original program and its mutants, and to check for redundancy among mutants. If the binary code is identical, the mutant is confirmed to be equivalent or redundant.

A recent evaluation of TCE [24] considered programs written in Java and C. For each language, two sets of programs were analyzed (6 small, and 6 medium or large Java programs; and 18 small, and 6 medium and large C programs). On average, for the first sets of programs, for which ground-truth information was available (that is, all equivalent mutants were manually classified), TCE identified 30% of the equivalent mutants for C and and 54% for Java. For medium and large programs, considering all mutants, TCE classified 5.7% of Java mutants and 7.4% of C mutants as equivalent, and 5.4% of Java mutants and 21% of C mutants as duplicate. The study did not report the true percent of equivalent and duplicate mutants for medium and large programs, as that would have taken extensive (and time-consuming) hand analysis.

At the source code level, Kintis and Malevris [23] explored the concept of *mirrored mutants*; if two or more mutants are present in similar code fragments, and particularly in identical code locations inside these fragments, defining one of those mutants as equivalent to the original program may imply that the others are also equivalent. The authors used a clone detection tool to detect the mirrored mutants. An evaluation with six small Java programs resulted in average 49% automatically classified equivalent mutants, with precision ranging from 88% to 100%. Also at the source code level, Ji et al. [20] used a domain reduction technique to compute a distance value between two mutants to classify mutants into specific clusters. They experimented with 12 mutation operators from the muJava tool [28] and one small

Java program. The predefined number of algorithm executions did not allow for precise extraction of results. In our work, sub-processes 1.1, 1.2, and 1.3 in Figure 2 can handle either source code or binary code as program abstractions to calculate similarity. As opposed to previous research [20, 23], *SiMut* does not compute the similarity between mutants of a single program; instead, it computes the similarity among various programs.

Chekam et al.'s FaRM approach [6, 8] relies on internal program information and supervised ML algorithms to automatically classify mutants that lead to more effective test cases in terms of probability of revealing real unknown faults. FaRM uses 28 program features to train the classifier. The features are derived from the control-flow graph, the abstract syntax tree, and the mutation type. Chekam et al. found that FaRM outperforms random mutation and selective mutation in terms of fault revealing ability, ranging from 23% to 34% more revealed faults. When compared with our work, FaRM uses program features that encompasses complexity elements, which are also reflected by internal metrics. As described in Section III-C, our approach could be combined with FaRM; from a large set of input programs, *SiMut* could find the most similar programs to be used during training, with the expectation of achieving more precise mutant classification.

Cruz and Eler [11] explored program clustering based on CK metrics to estimate program testability. They studied Java programs and generated clusters using the Expectation Maximization (EM) algorithm, based on the CBO, LCOM, RFC, and WMC metrics. The testability of a class was characterized as high or low, according to the structural coverage and mutation score produced by the associated test sets. The precision of the EM algorithm was assessed with the kNN (*k Nearest Neighbors*) algorithm [1]. In particular, Cruz and Eler used $k = 3$ and in 81.5% of cases, kNN confirmed the clusters produced by the EM algorithm precisely grouped classes with the same level of testability.

VIII. Conclusions and Future Work

This paper introduced the *SiMut* conceptual framework to support cost reduction of mutation testing based on historical data and program similarity. Although numerous papers have reported success using cost reduction techniques when averaged over many programs, Kurtz et al. [26] demonstrated that, although the techniques work well on average, they do not work well on individual programs. They suggested the idea of *individualized* cost reduction, where the cost reduction technique is tailored to individual programs. This leaves the problem: How do we know which cost reduction technique will work well with a given program before we test it? Thus, the key goal of *SiMut* is to support the application of mutation testing to a new, untested program u based on knowledge obtained from testing programs that are similar to u. The framework has three main variation points: the program abstraction that represents the programs, the approach to be used to calculate similarity among programs, and the mutation cost reduction technique that to apply to u.

We demonstrated the feasibility of implementing and running *SiMut* with a proof-of-concept implementation and a pilot study. We used internal program metrics as the program abstraction, clustering as the similarity calculation approach, and one-op mutation as the cost reduction technique. We also presented some usage scenarios, and discussed some issues related to the use of the framework, focusing on the composition of a representative dataset to properly explore the potential of *SiMut*.

For future work, the implementation can be reused and customized for other experiments, including replications, and for properly supporting the evolution of *SiMut*. We also plan to analyze how the mechanisms *SiMut* uses can be used in other contexts. Examples are the metrics collector and the clustering calculator, which may also be used for program analysis purposes. Last but not least, we would also like to compare programs at the method level.

Acknowledgements

Alessandro Pizzoleto and Lucas Dallilo were supported by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, and Federal University of São Carlos, Brazil. Fabiano Ferrari was partly supported by Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP, Brazil) - grant #2016/21251-0, Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq, Brazil) - grant #306310/2016-3, and Federal University of São Carlos, Brazil. Jeff Offutt was partly supported by KKS (The Knowledge Foundation), by project 20130085, Testing of Critical System Characteristics (TOCSYC); and partly by a Google educational grant, Self-Paced Learning Increases Retention and Capacity (SPARC).

References

[1] D. W. Aha, D. Kibler, and M. K. Albert, "Instance-based Learning Algorithms," *Machine Learning*, vol. 6, no. 1, 1991.

[2] P. Ammann, M. E. Delamaro, and A. J. Offutt, "Establishing Theoretical Minimal Sets of Mutants," in *Proc. the 7th Internl. Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2014.

[3] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is Mutation an Appropriate Tool for Testing Experiments?" in *Proc. the 27th Internl. Conf. on Software Engineering (ICSE)*. ACM, 2005.

[4] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi, "Toward the Determination of Sufficient Mutant Operators for C," *Soft. Testing, Verification and Reliability*, vol. 11, no. 2, 2001.

[5] T. Budd and D. Angluin, "Two Notions of Correctness and their Relation to Testing," *Acta Informatica*, vol. 8, no. 1, 1982.

[6] T. Chekam, M. Papadakis, T. Bissyandé, and Y. Le Traon, "Predicting the Fault Revelation Utility of Mutants," in *Proc. the 40th Internl. Conference on Software Engineering (ICSE) - Posters Track*. IEEE, 2018.

[7] T. T. Chekam, M. Papadakis, Y. Le Traon, and M. Harman, "An Empirical Study on Mutation, Statement and Branch Coverage Fault Revelation That Avoids the Unreliable Clean Program Assumption," in *Proc. the 39th Internl. Conference on Software Engineering (ICSE)*. IEEE, 2017.

[8] T. T. Chekam, M. Papadakis, T. F. Bissyandé, Y. Le Traon, and K. Sen, "Selecting Fault Revealing Mutants," SnT Centre, University of Luxembourg; and University of California, Joint Technical Report arXiv:1803.07901, 2018.

[9] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. on Soft. Eng.*, vol. 20, no. 6, 1994.

[10] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *Proc. the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, 1977.

[11] R. C. Cruz and M. M. Eler, "Using a Cluster Analysis Method for Grouping Classes According to their inferred Testability: An Investigation of CK Metrics, Code Coverage and Mutation Score," in *Proc. the 36th Internl. Conf. of the Chilean Computer Science Society (SCCC)*. Chilean Comp. Science Society, 2017.

[12] L. D. Dallilo, A. V. Pizzoleto, and F. C. Ferrari, "An Evaluation of Internal Program Metrics as Predictors of Mutation Operator Score," in *Proc. the 4th Brazilian Symposium on Systematic and Automated Software Testing (SAST)*. ACM, 2019.

[13] M. Daran and P. Thévenod-Fosse, "Software error analysis: A real case study involving real faults and mutations," in *Proc. the 1996 ACM SIGSOFT Internl. Symposium on Software Testing and Analysis (ISSTA)*. ACM, 1996.

[14] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *IEEE Computer*, vol. 11, no. 4, 1978.

[15] L. Deng, A. J. Offutt, and N. Li, "Empirical Evaluation of the Statement Deletion Mutation Operator," in *Proc. the 6th Internl. Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2013.

[16] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, 2005.

[17] E. Frank, M. A. Hall, and I. H. Witten, *Data Mining: Edition Practical Machine Learning Tools and Techniques*, 4th ed. Morgan Kaufmann, 2016.

[18] P. G. Frankl, S. N. Weiss, and C. Hu, "All-uses vs Mutation Testing: An Experimental Comparison of Effectiveness," *Journal of Systems and Software*, vol. 38, no. 3, 1997.

[19] M. Harman, Y. Jia, P. Reales, and M. Polo, "Angels and Monsters: An Empirical Investigation of Potential Test Effectiveness and Efficiency Improvement from Strongly Subsuming Higher Order Mutation," in *Proc. the 29th Internl. Conference on Automated Software Engineering (ASE)*. ACM, 2014.

[20] C. Ji, Z. Chen, B. Xu, and Z. Zhao, "A Novel Method of Mutation Clustering Based on Domain Analysis," in *Proc. the 21th Internl. Conference on Software Engineering and Knowledge Engineering (SEKE)*. Knowledge Systems Institute Graduate School, 2009.

[21] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Trans. on Soft. Eng.*, vol. 37, no. 5, 2011.

[22] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are Mutants a Valid Substitute for Real Faults in Software Testing?" in *Proc. the 22nd Internl. Symposium on Foundations of Software Engineering (FSE)*. ACM, 2014.

[23] M. Kintis and N. Malevris, "Identifying More Equivalent Mutants via Code Similarity," in *Proc. the 20th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2013.

[24] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon, and M. Harman, "Detecting Trivial Mutant Equivalences via Compiler Optimisations," *IEEE Trans. on Soft. Eng.*, vol. 44, no. 4, 2017.

[25] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, N. Malevris, and Y. Le Traon, "How Effective are Mutation Testing Tools? An Empirical Analysis of Java Mutation Testing Tools with Manual Analysis and Real Faults," *Empirical Software Engineering*, vol. 23, no. 4, 2018.

[26] B. Kurtz, P. Ammann, A. J. Offutt, M. E. Delamaro, M. Kurtz, and N. Gökçe, "Analyzing the Validity of Selective Mutation with Dominator Mutants," in *Proc. the 24th Internl. Symposium on Foundations of Software Engineering (FSE)*. ACM, 2016.

[27] N. Li, U. Praphamontripong, and A. J. Offutt, "An Experimental Comparison of Four Unit Test Criteria: Mutation, Edge-Pair, All-Uses and Prime Path Coverage," in *Proc. the 4th Internl. Workshop on Mutation Analysis (Mutation)*. IEEE, 2009.

[28] Y. S. Ma, A. J. Offutt, and Y. R. Kwon, "MuJava: An Automated Class Mutation System," *Soft. Testing, Verification and Reliability*, vol. 15, no. 2, 2005.

[29] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Józala, "Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation," *IEEE Trans. on Soft. Eng.*, vol. 40, no. 1, 2014.

[30] A. P. Mathur, *Foundations of Software Testing*, 1st ed. Addison-Wesley Professional, 2007.

[31] A. P. Mathur and W. E. Wong, "An Empirical Comparison of Data Flow and Mutation Based Test Adequacy Criteria," *Soft. Testing, Verification and Reliability*, vol. 4, no. 1, 1994.

[32] T. J. McCabe, "A Complexity Measure," *IEEE Trans. on Soft. Eng.*, vol. SE-2, no. 4, 1976.

[33] MuJava, "muJava Home Page," Online, 2015, https://cs.gmu.edu/~offutt/mujava/ - accessed in December, 2019.

[34] G. J. Myers, T. Badgett, and C. Sandler, *The Art of Software Testing*, 3rd ed. John Wiley & Sons, 2011.

[35] A. J. Offutt and J. Pan, "Detecting Equivalent Mutants and the Feasible Path Problem," in *Proc. the 11th Annual Conference on Computer Assurance (COMPASS)*. ACM, 1996.

[36] A. J. Offutt, G. Rothermel, and C. Zapf, "An Experimental Evaluation of Selective Mutation," in *Proc. the 15th Internl. Conference on Software Engineering (ICSE)*. IEEE, 1993.

[37] A. J. Offutt, Y.-S. Ma, and Y.-R. Kwon, "The Class-Level Mutants of MuJava," in *Proc. the Internl. Workshop on Automation of Software Test (AST)*. ACM, 2006.

[38] A. A. L. Oliveira, C. G. Camilo-Junior, and A. M. R. Vincenzi, "A Coevolutionary Algorithm to Automatic Test Case Selection and Mutant in Mutation Testing," in *Proc. the 2013 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2013.

[39] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon, "Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique," in *Proc. the 37th Internl. Conference on Software Engineering (ICSE)*. ACM, 2015.

[40] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation Testing Advances: An Analysis and Survey," in *Advances in Computers*, A. M. Memon, Ed. Elsevier, 2019, vol. 112.

[41] D. Pelleg and A. W. Moore, "X-means: Extending K-means with Efficient Estimation of the Number of Clusters," in *Proc. the 17th Internl. Conference on Machine Learning*. Morgan Kaufmann Publishers, 2000.

[42] A. V. Pizzoleto, F. C. Ferrari, A. J. Offutt, L. Fernandes, and M. Ribeiro, "A Systematic Literature Review of Techniques and Metrics to Reduce the Cost of Mutation Testing," *Journal of Systems and Software*, vol. 157, 2019.

[43] M. Polo, M. Piattini, and I. García-Rodríguez, "Decreasing the Cost of Mutation Testing with Second-Order Mutants," *Soft. Testing, Verification and Reliability*, vol. 19, no. 2, 2009.

[44] P. Reales, "Bacterio Mutation Test System - User Manual," Online, 2011, https://alarcos.esi.uclm.es/per/preales/bacterio/BacterioManual.pdf - accessed in February, 2020.

[45] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach," *Science of Computer Programming*, vol. 74, no. 7, 2009.

[46] A. Siami-Namin and S. Kakarla, "The use of mutation in testing experiments and its sensitivity to external threats," in *Proc. the 20th Internl. Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2011.

[47] A. Terceiro, J. Costa, J. Miranda, P. Meirelles, L. R. Rios, L. Almeida, C. Chavez, and F. Kon, "Analizo: An Extensible Multi-Language Source Code Analysis and Visualization Toolkit," in *Proc. the 1st Brazilian Conference on Software: Theory and Practice (CBSoft) – Tools Session*, 2010.

[48] R. H. Untch, "On Reduced Neighborhood Mutation Analysis Using a Single Mutagenic Operator," in *Proc. the 47th Annual Southeast Regional Conference (ACM-SE)*. ACM, 2009.

[49] A. Walenstein, M. El-Ramly, J. R. Cordy, W. S. Evans, K. Mahdavi, M. Pizka, G. Ramalingam, and J. W. Von-Gudenberg, "Similarity in Programs," in *Duplication, Redundancy, and Similarity in Software*, ser. Dagstuhl Seminar Proceedings, 2007.