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 \( O = \{o_1, o_2, \ldots, o_n\} \) 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 [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 experimentation 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 Similarity to support Cost Reduction of Mutation). SiMut supports the cost reduction of mutation testing based on historical data and on program similarity as a core element. It 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.
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 (untested) 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 \( C \) 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 \( u \), obtaining results \( R_u \).
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\footnote{http://www.analizo.org/} to calculate metrics. Analizo computes 10 project level metrics and 16 class level metrics, including the CK metrics \cite{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 \cite{41}. Third, we use the X-means algorithm, as implemented in the Weka API \cite{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 \cite{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
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 \[3] \[44\] 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 Kemere proposed six metrics for object-oriented software \[9\]. Table I defines the metrics, and Table II shows the values gathered using the Analizol tool for some of the programs. 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.

1Metrics 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 \(B = \{P_1, P_2, P_3, P_4\}\). If the LOI and AOR operators were applied to \(P_1\), \(P_2\) and \(P_3\), and the AOR and ROR operators were applied to \(P_1\) and \(P_4\), we created two sets of programs \(B_1 = \{P_1, P_2, P_4\}\) 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 \(B_1\), \(B_2\), and \(B_3\). 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. \(R_1\) and \(R_2\), 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.

- **\(R_1\):** We compute \(R_1\) 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 \(OC_u\) is the set of best operators for each cluster \(i\). \(R_1\) evaluates the intersection of \(O_u\) and \(OC_u\), considering the cluster that \(u\) is associated with, and the remaining clusters formed from the program set.

- **\(R_2\):** We next compute \(R_2\) 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. \(R_2\) evaluates the intersection between the sets \(O_u\) and \(OP_u\), then compares this result

---

**Table I**

<table>
<thead>
<tr>
<th>Source</th>
<th># Programs</th>
<th># Tests</th>
<th># Mutants</th>
<th># Operators</th>
<th>Operators</th>
<th>Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>[15]</td>
<td>29</td>
<td>18</td>
<td>4,161</td>
<td>15</td>
<td>LOI; SDL; AOS; VDL; COE; AOR; ROR; ODL; CDE; COR; ASR5; AODU; AODS; COD</td>
<td>muJava</td>
</tr>
<tr>
<td>[15][3]</td>
<td>9</td>
<td>303</td>
<td>1,547</td>
<td>10</td>
<td>ADIS; ADIU; AOR; ROR; LOI; ADDS; COR; ADDU; COE; COJ</td>
<td>muJava</td>
</tr>
<tr>
<td>[16]</td>
<td>13</td>
<td>11</td>
<td>2,418</td>
<td>18</td>
<td>JID; JSF; JSI; AOS; AOR; ASR3; COI; LOI; ROR; EAM; JDC; JTD; JTI; IOD; ADDU; COD; COR</td>
<td>muJava</td>
</tr>
<tr>
<td>[17]</td>
<td>21</td>
<td>105</td>
<td>10,147</td>
<td>30</td>
<td>EAM; EOC; IOD; ROR; IPC; JSI; JSF; JCL; OMIR; PCC; ISD; PRV; OAN; PMD; PPD; EMM; IOP; CDC; JID; ROR; UOI; SWA; LCR; NUF; ABS; TEX; AOR; DEC; INC</td>
<td>muJava</td>
</tr>
<tr>
<td>[19]</td>
<td>211</td>
<td>934</td>
<td>19,851</td>
<td>10</td>
<td>DEC; SWA; AOR; ABS; INC; UOI; NUF; ROR; LCR; TEX</td>
<td>Bacterio</td>
</tr>
</tbody>
</table>
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 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 \textit{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) \textit{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] [33]. 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 (\(\mathbb{M}\)) 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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIT M ((\langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle))</td>
<td></td>
</tr>
<tr>
<td>AORB ((\langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle))</td>
<td></td>
</tr>
<tr>
<td>SOR ((\langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle))</td>
<td></td>
</tr>
<tr>
<td>LOR ((\langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle))</td>
<td></td>
</tr>
<tr>
<td>COR ((\langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle, \langle op1, op2\rangle))</td>
<td></td>
</tr>
<tr>
<td>((\langle op3, op4\rangle, \langle op3, op4\rangle, \langle op3, op4\rangle, \langle op3, op4\rangle, \langle op3, op4\rangle, \langle op3, op4\rangle, \langle op3, op4\rangle, \langle op3, op4\rangle, \langle op3, op4\rangle))</td>
<td></td>
</tr>
</tbody>
</table>

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 Malvavis [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 3 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. [20] 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.

ACKNOWLEDGMENTS

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 #306337/2016-3, 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
