Assessing and Improving the Mutation Testing Practice of PIT

Thomas Laurent†‡, Anthony Ventresque∗, Mike Papadakis†, Christopher Henard†, and Yves Le Traon‡
∗Lero@UCD, School of Computer Science, University College Dublin, Ireland
†Ecole Centrale de Nantes, France
‡Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg

In summary, the contributions of this paper are:

• We describe and implement a comprehensive mutant set for Java that reflects the beliefs of the mutation testing community as it has been recorded in the literature [9], [4] and discussed during the Mutation 2014 and 2015 workshops, e.g., [17].
• We provide empirical evidence that the comprehensive mutant set is superior to the one often used by mutation testing tools. This set is statistically significant superior from 11% to 62% of the studied program classes.
• To support future research, we will submit our code to the PIT repository to became available. Our new version of PIT that supports the comprehensive mutants is also available or request it.

II. TERMINOLOGY & BACKGROUND

This section introduces the terminology and the concepts that are used throughout the paper. First, II-A presents the mutation testing process. Then, Section II-B describes the selection of mutants. Finally, the notion of disjoint mutants is introduced in Section II-C.

†For inquiries please contact Anthony Ventresque, antony.ventresque@ucd.ie
A. Mutation Testing

Mutation analysis operates by injecting defects in the software under investigation. Thus, given a program, several variants of this program are produced, each variant containing a defect. These are called mutants and they are made by altering (mutating) the code, either source code or executable binary code, of the program under test. The creation of mutants is based on syntactic rules, called mutant operators, that transform the syntax of the program. For example, an arithmetic mutant operator changes an instance of an arithmetic language operator such as the ‘+’ to another one, such as ‘−’.

Mutants are produced by syntactic changes introduced by mutant operators. These are the instances that are produced by applying an operator on every point of the code under investigation that matches their respective rule. Thus, every mutant has a single and specific syntactic difference from the original program. For reasons that we will discuss in the related work (see Section VI), mutation testing uses mutants produced by simple syntactic changes.

Mutants are used to measure how good the employed test cases are in checking the software under assessment. This is done by observing the runtime behavior of the original, non-mutated, and the mutated programs. When comparing the program outputs of the original with the mutated programs, and found differences, we exhibit behavior discrepancies [18]. Such differences are attributed to the ability of the used test to project the syntactic program changes to its behavior, i.e., to show a semantic difference. When mutants exhibit such differences in their behavior, they are called “killed”. Those that do not exhibit such differences are called “live”. Mutants might not exhibit any difference in their behavior either because the employed test cases were not capable of revealing them or because they are functionally equivalent with the original program. Mutants belonging to the latter case are called equivalent [18].

Mutation testing refers to the process of using mutation analysis as a means of quantifying the level of thoroughness of the test process. Thus, it measures the number of mutants that are killed and calculates the ratio of those over the total number of mutants. This ratio represents the adequacy metric and is called mutation score. Ideally, to have an accurate metric, equivalent mutants must be removed from the calculation of the score. However, this is not possible since judging programs’ equivalence is an undecidable problem [19].

B. Mutant Selection

Selective mutation was shown to be valid in several studies involving programs written in Fortran [9] and in C [20]. As a result, Java mutation tools were built based on the findings of these studies. To address the scalability issues of the method, tool developers made further reductions. Thus, popular tools like PIT [11] support a very small and restrictive set of mutants that neither follows any suggestion from previous studies nor practical experience.

PIT, even in its latest version that supports an extended mutant set, has several shortcomings. One such example is the relational operator for which PIT replaces one instance of the operator by only another one, i.e., mutates < only to <=, or <= only to <, or > only to >=, or >= only to >. However, this practice is not sufficient. Indeed, previous studies have shown that three mutants are needed to avoid a reduced effectiveness of the method [21, 22].

Although practical, mutant reduction should not be at the expense of the method effectiveness. Almost all previous studies were based on the assumption that mutants are equal [23]. However, this does not hold in practice and has the potential to bias the conducted research as recent studies show [24], [23], [25]. Therefore, when using mutation for research purposes, it is mandatory to make sure that a representative mutant set is employed.

C. Disjoint Mutants

In literature, mutation testing is extensively used to support experimentation [3], [8], i.e., it is used to measure the level of test thoroughness achieved by various testing methods. Mutation score serves as a comparison basis between testing techniques and hence, as a yardstick to judge the winning one. This practice is quite popular, and introduce severe problems that can threaten the validity of the conducted research.

The problem is that not all mutants are of equal power [23], which means that some are useful and some are not. Indeed, mutants cover the full spectrum of cases, including trivial ones, very easy to kill, duplicated, equivalent ones and also hard to kill ones. Those of the last category are of particular interest since they lead to strong tests [23]. Hard to kill, trivial and easy to kill mutants are defined with respect to the employed test suite [23]. Thus, mutants killed by a small percentage of tests that exercise them are hard to kill, while, those killed by a large one are easy to kill.

When using mutation as a basis for comparing testing methods, a filtering process that sweeps out the duplicated and equivalent mutants is needed [26]. However, this process might not be adequate since in most cases many mutants tend to be killed jointly [25]. Thus, they do not contribute to the test process despite being considered. This has an inflation effect on the mutation score computation since only a very small fraction of mutants contribute to the test process.

This issue was initially raised by Kintis et al. [25] who introduced the concept of disjoint mutants, i.e., minimum number of mutants that contribute to mutation score. Their use is motivated by the same study which demonstrated that hard to kill mutants also suffer from the inflation problem. Later Amman et al. [24] formalized this concept, name it as “minimum mutants”, and suggested using it as a way to bypass the mutation score inflation problem.

In this paper we follow an analysis based on both all and disjoint mutants. Disjoint mutants have the advantage to cover the whole spectrum of mutants and suffer less from the mutant inflation effect. Their identification is an NP-complete
Input: A set \( S \) of mutants
Input: A set \( T \) of test cases
Input: A matrix \( M \) of size \(|T| \times |S|\) such as \( M_{ij} = 1 \) if test \( i \) kills mutant \( j \)
Output: The disjoint mutant set \( D \) from \( S 

Algorithm 1: Disjoint Mutants

1. \( D = \emptyset \) /* Remove live mutants */
2. \( S = S \setminus \{m \in S | \forall i \in 1..|T|, M_{ij} \neq 1\} \)
3. \( S = S \setminus \{m \in S | \exists m' \in S | \forall i \in 1..|T|, M_{ij}(m) = M_{ij}(m')\} \)
4. while \(|S| > 0\) do
    5.     maxSubsumed = 0
    6.     subsumedMut = null
    7.     maxMutSubsuming = null
    8.     foreach \((m \in S)\) do
        9.         sub_m = \{m' \in S | \forall i \in 1..|T|, \( M_{ij}(m) = 1 \Rightarrow (M_{ij}(m') = 1) \}\)
        10.        if \((|sub_m|) > maxSubsumed) then
            11.            maxSubsumed = |sub_m|
            12.            maxMutSubsuming = m
            13.            subsumedMut = sub_m
        14.     end
    15. end
    16. \( D = D \cup \{\text{maxMutSubsuming}\} \)
    17. \( S = S \setminus \text{subsumedMut} \)
18. end
19. return \( D \)

In this paper, we deal with this issue by investigating the extent to which the mutant sets employed by popular mutation testing tools meet the standards as expressed by the mutation testing literature and community. We call the first set as the “common” mutant set and the second one as the “comprehensive” one.

Our goal is to validate the use of the popular mutant set which was introduced by the developers of popular mutation testing tools. We seek to investigate this issue since modern tools like PIT\(^{\text{11}}\) have been extensively used in the recent years.\(^{\text{12}}\) Thus, a possible issue with their adopted mutants can question the effectiveness of the mutation method and hence the conducted research. We therefore compare the extent to which the popular mutant set conforms to the test requirements possessed by the mutation testing literature. To validate this practice we use large open source projects written in Java with mature test suites.

IV. EXPERIMENTAL STUDY

This section first states the Research Questions (RQs) under investigation. Then, the subjects, settings and tools used for the experiments are described. Finally, the last subsections detail the studied mutant operators and the analysis procedure followed to answer the RQs.

A. Definition of the Experiment and Research Questions

Current research on software engineering has largely focused on using mutation analysis as supported by the existing mutation testing tools. However, a central role in mutation testing is played by the mutants that are used; meaning that the effectiveness of the method is sensitive to the employed mutants\(^{\text{12}}\). Therefore, it is important to know whether the commonly used mutants, as supported by these tools, are suitable. In other words we seek to determine the degree to which the commonly used mutants are representative of those suggested by the literature, i.e., the comprehensive mutant set. This leads us to our first research question:

**RQ1 (Effectiveness).** Is there any effectiveness difference between the commonly used mutants and the comprehensive ones?

Since we are interested in testing, we seek to identify the mutants that are more effective at measuring the ability of test cases to exercise each point of the program under investigation. Mutation score measurements can differ when different mutant sets are employed. The measurements are affected by the number of mutants, of equivalent ones, of trivial and hard to kill ones. To deal with this issue, we perform an objective comparison, i.e., we measure the extent to which one method covers the requirements of the other, between the two examined mutant sets. Thus, we seek to measure the ratio of mutants, of the one set, that are found by the tests that are selected based on the other mutant set. Thus, the “weaker” mutants will lead to “weaker” tests and hence kill a smaller fraction of the “stronger” mutants. Here, it should be noted that objective comparisons form a common practice in mutation testing literature, e.g.,\(^{\text{4}, \text{6}, \text{9}}\).

As discussed previously, in Section II-C there is a potential problem with this practice due to the inflation effect of the trivial mutants. We thus measure the ratios of the disjoint mutants that are killed. Therefore, to answer RQ1 we report results based on two effectiveness measures; the percentage of all mutants killed and the percentage of the disjoint ones that are killed.

So far, our investigations focus on whether mutants of one set can capture all the faults introduced by the other set. However, this analysis tells us nothing about the difficulty of exposing mutants. Thus, mutant easiness is another important attribute of mutants\(^{\text{4}, \text{23}}\). This is due to the fact that hard to kill mutants indicate a relatively small semantic
change difference \cite{23}, \cite{27} that is often easy to overlook when testing. Thus, through our second research question, we investigate whether the comprehensive mutant set introduces harder to kill mutants than the commonly used one:

**RQ2 (Easiness).** What is the difference, in terms of difficulty to expose mutants, between the common and the comprehensive mutants?

Mutation testing has a widespread reputation of being computationally demanding. In the past, practitioners believed that it cannot scale to real-world systems mainly due to the large number of mutants. However, tools like PIT and Javalanche proved that this belief was incorrect \cite{28}. This ability of the tools can be attributed to the restricted mutant set they employ and to the advanced mutant generation and execution techniques used. Therefore, it is possible that the comprehensive mutant set is too expensive to be used in practice. Hence, we investigate:

**RQ3 (Scalability).** What is the execution time differences of the comprehensive mutants when compared with the common ones?

We measure execution time since it forms a direct measure of the application cost of the method. We do not consider other parameters that can influence the application cost, such as the number of equivalent mutants, the test generation cost, since they fall outside the scope of the present paper. Here, we focus on the effectiveness of the method as conducted by recent studies and thus, leaving the issue of its application cost open for future research.

### B. Subject Programs

The experiments are conducted on the 5 Java projects recorded in Table I. For each of them, the version, lines of code (calculated with the JavaNCSS tool \cite{29}), number of classes (for which test suites exist) and number of tests are reported.

<table>
<thead>
<tr>
<th>Subjects</th>
<th>Version</th>
<th>LoC</th>
<th>Classes</th>
<th>Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>joda-time</td>
<td>2.8.1</td>
<td>18,611</td>
<td>210</td>
<td>4,129</td>
</tr>
<tr>
<td>jfreechart</td>
<td>1.0.19</td>
<td>46,986</td>
<td>290</td>
<td>1,320</td>
</tr>
<tr>
<td>jaxen</td>
<td>1.1.6</td>
<td>6,790</td>
<td>152</td>
<td>646</td>
</tr>
<tr>
<td>commons-lang</td>
<td>3.3.4</td>
<td>16,286</td>
<td>199</td>
<td>3,373</td>
</tr>
<tr>
<td>commons-collections</td>
<td>4.4.0</td>
<td>11,281</td>
<td>243</td>
<td>2,210</td>
</tr>
</tbody>
</table>

Table III presents some descriptive statistics (minimum median, mean and maximum values) about the employed mutants as they appear in the classes of the studied projects. Thus, the table records details about the number of mutants, the number of killable mutants (determined based on the available test suite), and mutation score for the common and comprehensive operators for each project.

### F. Analysis Procedure for Answering the Research Questions

To answer RQ1, we constructed test suites using the common set. This was performed by incrementally adding random tests in the suites and keeping only those that increase mutation score. So, if the randomly selected tests failed to kill any additional mutant, i.e., it is redundant with respect to the employed mutants, the test was not included. This is a typical process followed by many previews studies, e.g., \cite{4}, \cite{2}. Thus, we measured (a) the number of mutants of the comprehensive set that are killed by the tests selected based on the common set and (b) the number of mutants of the comprehensive sets found when using all available tests. Since the tests were selected at random, this process was repeated 30 times. As a result, we obtain 30 instances for each one of the two measures for every class of each project. We compared them with Wilcoxon test using the R statistical computing project \cite{31}. From this test, we obtain a p-value which represents the probability that measure (a) is higher than measure (b). Following the usual statistical inference procedures we consider the differences as statistically significant if they provide a p-value lower than
As a result, when 100% of the test cases kill a mutant, the latter is denoted as very easy to kill.

To answer RQ3, we applied mutation analysis as it is supported by the current version of the tool and record the time required.

V. RESULTS & ANSWERS TO THE RESEARCH QUESTIONS

This section reports on the experimental results and answers the RQs stated in previous section.

A. RQ1 - Effectiveness

For this question, we first consider the number of comprehensive mutants killed by test cases selected based on the common mutants, and the number of comprehensive mutants killed by test cases targeting them. This forms two mutation scores for each Java class of the considered projects. The comparison has been performed 30 times, thus yielding 30 scores for each Java class of the considered projects. The values of Table III imply a significant difference.

The ratio (a)/(b) forms the objective comparison time required.

V. RESULTS & ANSWERS TO THE RESEARCH QUESTIONS

This section reports on the experimental results and answers the RQs stated in previous section.

A. RQ1 - Effectiveness

For this question, we first consider the number of comprehensive mutants killed by test cases selected based on the common mutants, and the number of comprehensive mutants killed by test cases targeting them. This forms two mutation scores for each Java class of the considered projects. The comparison has been performed 30 times, thus yielding 30 mutation scores of the two types per class.

Table IV records the classes for which there is a statistical significance between the two measures. As can be seen, there

0.05, i.e., 5% is our significance level. In RQ1, we record the number of classes for which there is a statistically significance difference. The ratio (a)/(b) forms the objective comparison score when using all mutants. The values of Table III imply that the projects have classes with only 1 mutant and 0 killable ones. Also, there are classes where all mutants are killed. Thus, we base the results of the objective comparison at the class granularity level and present them according the first three quartiles, i.e., according to the ordered 25%, median (50%) and 75% values. Finally, we compute the ratio of the disjoint mutants of the comprehensive set that are found by the tests selected based on the common set. The distance from value 1 on both the objective comparison and disjoint mutants scores quantify the effectiveness differences between the examined mutants.

To answer RQ2, we measure the easiness of killing mutants. The easiness of killing a mutant is defined as the number of test cases that kill a mutant, towards the total number of test cases. As a result, when 100% of the test cases kill a mutant, the latter is denoted as very easy to kill.

To answer RQ3, we applied mutation analysis as it is supported by the current version of the tool and record the time required.

V. RESULTS & ANSWERS TO THE RESEARCH QUESTIONS

This section reports on the experimental results and answers the RQs stated in previous section.

A. RQ1 - Effectiveness

For this question, we first consider the number of comprehensive mutants killed by test cases selected based on the common mutants, and the number of comprehensive mutants killed by test cases targeting them. This forms two mutation scores for each Java class of the considered projects. The comparison has been performed 30 times, thus yielding 30 mutation scores of the two types per class.

Table IV records the classes for which there is a statistical significance between the two measures. As can be seen, there
is a significant difference for 11% of the classes for commons-collections, for 62% of them for joda-time.

We now evaluate the percentage of mutants of the comprehensive set that are killed by the test cases selected based on the common set. This percentage is denoted as objective comparison score. Figure 1 records the objective comparison scores for the 5 programs. The objective comparison has been performed 30 times per class (represented on the x-axis), thus yielding 30 different scores for each class of each program.

The 4 colors of the plot represent the distribution of the 30 ordered scores according to the quartiles. Thus, from the lightest to the darkest color, the first, second, third and fourth 25% of the resulting scores are represented. For instance, a light gray bar (the first quartile reaching 0.6) means that the
lowest 25% of the 30 scores obtained are below or equal to 0.6. The black area represents the values above the third quartile, i.e., last 25% of the 30 scores. In that, if a bar is completely light gray, it means that most of the mutants killed by the test cases are the same on both sets, while the presence of darker colors on the graph indicates that there are mutants that are missed by the test cases.

The plots on the left part of Figure 1 represent the scores when considering all the mutants while the plots on the right side depict the results when considering the disjoint mutants. On the left side of the figure, i.e., when all the mutants are considered, we can see that in most of the cases the scores are close to 1, which means that there are only a few mutants missed by the test cases. This is especially the case

Fig. 2. Easiness of killing mutants (RQ2 - easiness). The plots on the left side display the easiness of all mutants while on the right side they display the easiness of the disjoint mutants. The y-axis represents the median mutant easiness, while x-axis represents the Java classes of the programs.
for jfreechart and commons-collections. For the other projects, there is a larger proportion of classes with lower scores, indicating that there are more mutants of the comprehensive sets missed by the test cases. Considering now the right part of Figure 1 that concerns the disjoint mutants, we can see that the proportion of missed mutants is even more important, in particular for joda-time where mutants are missed in more than half of the classes.

To conclude, there is a significant difference between the common mutants and the comprehensive ones. This shows that the test cases miss many killable mutants from the comprehensive set. The difference is even more significant when considering disjoint mutants.

### B. RQ2 - Easiness

Here, we evaluate whether mutants are difficult to kill or not. The easiness of killing a mutant is the percentage of the test cases that kill this mutant. Thus, if all the test cases kill a given mutant, the easiness is 1. By contrast, an easiness close to 0 means that the mutant is very difficult to kill, since only few test cases are able to identify it.

Figure 2 shows the easiness, median values, of the common mutants against the comprehensive ones for each class. A greater surface indicates that the corresponding mutants are easier to kill. The plots on the left part consider all the mutants while the right side are for the disjoint mutants. From these results, we can observe that the comprehensive mutants (represented by the gray bars) are harder to kill. For all the mutants, the difference in terms of easiness compared to the common mutants range from 2-5% for the 50% of the commons-lang program classes. In the case of jaxen, the easiness difference is 12% in almost the 60% of the program classes. With respect to the disjoint mutants, the difference goes beyond 20% for approximately 75% of the jaxen classes.

To summarize, the comprehensive mutants are harder to kill than the common ones when considering either all the mutants or the disjoint ones only. It means that the comprehensive set introduces faults which are more difficult to expose.

### C. RQ3 - Scalability

To evaluate the cost of using the comprehensive mutants, we measure the execution time of both the common and comprehensive sets. Table V records the number of mutants, killable mutants, execution time in seconds and execution time per mutant in seconds for both sets of mutants.

Considering the time in seconds, the comprehensive mutants require more time than the common ones. This is natural since the number of mutants is also much higher. The highest execution time is for jaxen with 31,077 seconds, i.e., approximately 8 hours and a half. Focusing on the time per mutants, the execution times between the two sets become rather similar, except for jaxen where the execution time is approximately 10 times longer.

Overall, the comprehensive set of mutants has an overhead in terms of execution time, but this is not unbearable.

### VI. Related Work

The following sections present studies with respect to mutation testing (VI-A), mutant reduction techniques (VI-B) and the accuracy of mutation score (VI-C).

#### A. Mutation Analysis

Mutation analysis was initially introduced as a test method helping the generation of test cases [32]. However, in recent years it has been proven to be quite powerful and capable of supporting various software engineering tasks [8]. In particular mutants have been used to guide test generation [22], [33], test oracle generation [34], to assist the debugging activities [18], [34], to evaluate fault detection ability [4] and to support regression testing activities like test selection and prioritization [16], [35]. The method has also been applied to models [36], software product lines [37] and combination strategies [38].

One of the main issues of the method is equivalent mutants [39], [40]. Despite the efforts from the community, e.g., [8], and recent advances [41], [26], [42], the problem remains open [26]. Similar situation arises when considering mutation-based test generation [22], [33].

Mutation has become popular thanks to its ability to represent real faults [4]. Also, many modern mutation tools are integrated with build systems and development tools, thus making their application easy [10]. However, such tools usually support mutants that are more restrictive than the popular set, and hence, overestimate or underestimate the employed measures, as shown by the present paper. Previous research suggested that the comprehensive mutant set has the ability to prod-subsume and to reveal more faults than most of the other white-box testing criteria [5], [7].

#### B. Mutant Reduction

From the early days of mutation testing, it was evident that mutants were far too numerous to be used in practice. Therefore, researchers have tried to identify subsets of them that are representative. The first reduction was made towards the coupling effect hypothesis, which states that tests revealing simple mutants can also reveal complex ones [32], [43].

A straight way to reduce the number of mutants is to randomly sample them [44]. Although, sampling can provide a range of trade-offs, Papadakis and Malevris [44] provided evidence that mutant sampling ratios of 10% to 60% have a loss on fault detection from 26% to 6%, respectively. Similar results have been shown in the study of Wong et al. [45].
Other mutant reduction strategies fall in the category of selective mutation [9]. Selective mutation tries to reduce the arbitrariness of random sampling by using only specific types of mutants.

C. Duplicated, Trivial and Redundant Mutants

The presence of redundant mutants has long been recognized, i.e., since 1993 [46], but only recently, received attention. The studies of Tai [21], [46] were focused on reducing the application cost of fault-based testing strategies. This was based on constraints that restrict the mutants introduced by the relational and logical operators so that they only consider non-redundant ones. Thus, their suggestion was to restrict the mutant instances of the relational and logical operators to the minimum possible number. In a later study, Kaminski et al. [47], [48] came to a similar conclusion about the relational operator. Thus, they suggested that mutation testing tools should reduce the number of redundant mutants by restricting the mutant instances of the relational operator.

Papadakis and Malevris [49] suggested using minimized constraint mutant instances to efficiently generate mutation-based test cases. Thus, when aiming at generating test cases there is no point in aiming at non-redundant mutants. On the same lines, Just et al. [50], demonstrated that by constraining the relational and logical operators, it is possible to reduce some of the redundancy between the mutants.

All the approaches discussed so far were based on an analysis at the predicate level, specifically designed for “weak” mutation. Thus, when applied to strong mutation, the analysis may not hold. This can be due to the following two reasons: a) constructs like loops and recursion can make a statement to be equivalent while the non-redundant ones are not, thus, resulting in degradation in the effectiveness of the method. These issues motivated the use of disjoint mutants that do not suffer from these problems.

The first study suggesting the use of non-redundant mutants when comparing testing techniques is that of Kintis et al. [25]. As discussed in Section III-C, Kintis et al. introduced the notion of disjoint mutants and demonstrated that the majority of mutants produced by the MuJava mutation testing tool is redundant. In the same lines, Amman et al. [24] defined algorithms for generating a minimum set of mutants based on the notion of dynamic subsumption. Their results confirmed the findings of Kintis et al. by demonstrating that the majority of mutants used by the MuJava and Proteum mutation testing tools are redundant. Later, Kurtz et al. [50] used static analysis techniques, such as symbolic execution to identify the minimum set of mutants.

Recently Papadakis et al. [26] used compilers to eliminate duplicated mutants, a special form of mutant redundancy. Duplicated mutants are those that are mutually equivalent but differ from the original program. In the study of Papadakis et al. [26] it is reported that 21% of all mutants is duplicated and can be removed by using compiler optimization techniques.

All the approaches discussed in this section either identified the problem of trivial/redundant mutants or used some form of redundancy elimination. However, none of them studied the differences of the commonly used operators from those suggested by the literature. Additionally, none of them uses disjoint mutants on real-world programs.

VII. Conclusions

This paper investigates the extent to which mutants used by popular mutation testing tools like PIT conform to mutation testing standards. Our study revealed a large divergence in the effectiveness of the popular mutants from the comprehensive ones. Comprehensive mutants are not only harder to kill but also score considerably higher than the common ones in most of the examined cases. Additionally, we report results by considering both concepts of disjoint and mutant easiness. Thus, we point out the importance of the problems introduced by both trivial and redundant mutants to be considered in future evaluations.

References


