Optimizing Mutation Testing by Discovering Dynamic Mutant Subsumption Relations

Marcio Augusto Guimarães* Leo Fernandes‡ Márcio Ribeiro* Marcelo d'Amorim† Rohit Gheyi*
*Federal University of Alagoas, Maceió-AL, Brazil
‡Federal Institute of Alagoas, Maceió-AL, Brazil
†Federal University of Pernambuco, Recife-PE, Brazil
*Federal University of Campina Grande, Campina Grande-PB, Brazil
{masg, marcio}@ic.ufal.br leonardo.oliveira@ifal.edu.br damorim@cin.ufpe.br rohit@dsc.ufcg.edu.br

***Abstract*—One recent promising direction on reducing costs of mutation analysis is to identify redundant mutations, i.e., mutations that are subsumed by some other mutations. Previous works found out redundant mutants manually through the truth table. Although the idea is promising, it can only be applied for logical and relational operators. In this paper, we propose an approach to discover redundancy in mutations through dynamic subsumption relations among mutants. We focus on subsumption relations among mutations of an expression or statement, named here as "mutation target." By focusing on targets and relying on automatic test generation tools, we define subsumption relations for dozens of mutation targets in which the MUJAVA tool can apply mutations. We then implemented these relations in a tool, named MUJAVA-M, that generates a reduced set of mutants for each target, avoiding redundant mutants. We evaluated MUJAVA and MUJAVA-M using classes of five open-source projects. As results, we analyze 2,341 occurrences of 32 mutation targets in 168 classes. MUJAVA-M generates less mutants (on average 64.43% less) with 100% of effectiveness in 20 out of 32 targets and more than 95% in 29 out of 32 mutation targets. MUJAVA-M also reduced the time to execute the test suites against the mutants in 52.53% on average, considering the full mutation analysis process.**

***Keywords*-Mutation Testing, Redundant Mutants, Minimal Mutants Set, Dynamic Subsumption**

I. INTRODUCTION

Usually, the costs of using mutation analysis are high, mainly due to the high number of generated mutants and the high computing time to execute the test suite against each mutant. However, some mutants, such as the redundant ones, may not be necessary for the effectiveness of mutation analysis and we may discard them [1]. A redundant mutant does not contribute to the test assessment process because they are killed when other mutants are also killed [1], [2]. In other words, redundant mutants are always subsumed by other mutants. Then, the generation of these mutants increases the total cost and does not help to improve the test suite. Ammann et al. [3] empirically identified that a number of the generated mutants are redundant. Also, Papadakis et al. [4] identified that such redundant mutants inflate the mutation score and that 68% of recent research papers are vulnerable to threats to validity due to the effect of these mutants.

To identify redundant mutants, we can take subsumption relations into account. Kaminski et al. [5] manually constructed subsumption hierarchies with the support of truth tables produced by the outcomes of mutants associated with the *Relational Operator Replacement* (ROR) mutation operator. This operator generates seven different mutations, but Kaminski et al. identified that only three mutations are sufficient to cover all input domains, yielding a reduction of 57%. Just et al. [6] expanded this idea with two more mutation operators. Both works use truth table to infer logical relationships across the operations. However, this only works for logical and relational operators. Although the idea is promising, we cannot apply it for non-logical operators.

In this paper we propose an approach to discover subsumption relations among mutants. Our approach uses *Dynamic Subsumption*, which relies on a set of tests. When applying an exhaustive set of tests, the dynamic subsumption tends to be stable and close to the true subsumption [7]. We focus on subsumption relations among mutations of a *mutation target*. A mutation target is a language expression or statement in which it is possible to apply a set of mutations of one or more mutation operators (e.g., `a + b`, `a > b`, `exp++` etc). Our hypothesis is that the subsumption relation found for one specific target might be less context-dependent, and in turn, generalizable to different programs in different domains.

We execute our approach in method-level mutation targets where the MUJAVA tool [8], [9] can apply mutations and define the subsumption relations for each target based on automatic test generation tools. We introduce a tool named MUJAVA-M to take these relations into account and avoid the generation of redundant mutants. We evaluate MUJAVA-M in terms of *number of generated mutants*, *effectiveness*, and *time savings*. To execute this evaluation, we use five open source programs, i.e., *joda-time*, *commons-math*, *commons-lang*, *h2*, and *javassist*. However, applying mutation analysis in all classes of these projects is not feasible. So, we selected a total of 168 class files from these projects to evaluate our approach. We count the *number of mutants generated* by MUJAVA and MUJAVA-M. We also verify *effectiveness* by checking whether MUJAVA-M has discarded the redundant mutants correctly. To calculate the *time savings* and again make our analysis feasible, we randomly selected five class files from each project. Then, we executed MUJAVA and MUJAVA-M with the original test suite of each subject and

measure the time to perform mutant generation and to execute the test suite against all mutants.

As results, we analyzed 2,341 occurrences of 32 mutation targets in 168 classes of the five subjects. MUJAVA generated 10,818 mutants whereas MUJAVA-M generated 3,848 mutants, representing a reduction of 64.43%. When considering the results for effectiveness, in 20 out of 32 mutation targets the effectiveness was 100%, and in 29 targets we achieved an effectiveness greater than 95%. Also, MUJAVA-M achieved promising time savings, i.e., it took 11 hours to perform the mutation analysis, representing a reduction of 52.53% when compared to MUJAVA.

The main contributions of this paper are:

- We propose an approach to discover dynamic mutant subsumption relations (Section III);
- We define subsumption relations for the mutation targets in which the MUJAVA mutation testing tool is able to create mutants and we embedded the minimal set of each target in a tool, i.e., MUJAVA-M (Section III);
- We evaluate our approach by executing MUJAVA-M against class files of open-source Java projects, showing promising results in terms of number of generated mutants and time savings (Section IV).

II. REDUNDANT MUTANTS

Mutation analysis uses mutation operators to introduce faults in the program to create mutants deliberately. In this context, there is a wide variety of mutation operators. Each mutation operator can implement a set of mutations. In this paper, we follow the same definition for "mutation" of previous work [10]: a *mutation* refers to a syntactic change (e.g., `a && b` $\mapsto$ `a || b`). For example, in a binary expression with a relational operator `lexp <op> rexp`, where `lexp` and `rexp` indicate expressions or literals and `<op>` is a relational operator (==, !=, >, >=, <, or <=), the *Relational Operator Replacement* (ROR) mutation operator performs seven mutations, replacing the original operator `<op>` with each of the other five relational operators and replacing the entire expression with `true` and `false`. Thus, for the binary expression `a > b`, the ROR operator performs the following seven mutations: `a > b` $\mapsto$ `a == b`, `a > b` $\mapsto$ `a != b`, `a > b` $\mapsto$ `a >= b`, `a > b` $\mapsto$ `a < b`, `a > b` $\mapsto$ `a <= b`, `a > b` $\mapsto$ `true`, and `a > b` $\mapsto$ `false`.

However, some mutations may not be necessary for the effectiveness of mutation analysis and are actually useless. In fact, they can yield *equivalent mutants* or *redundant mutants* [11]. In this paper, we focus on redundant mutants. To identify them, we rely on subsumption relations [7]. In this sense, a mutation m_1 *subsumes* another, say m_2, if whenever m_1 is detected, m_2 is also detected, i.e., detecting m_1 is a sufficient condition to detect m_2. The consequence of finding these subsumption relations for mutation tools is that subsumed mutations could not be applied, resulting in a polynomial, but relevant, speedup.

III. DEFINING SUBSUMPTION RELATIONS

We propose an approach to discover subsumption relations among mutations. It uses *Dynamic Subsumption*, which "*is computed relative to a specific set of tests. As the number of tests tends towards the entire domain of the artifact under test (not possible, of course), the dynamic subsumption approaches 'true' subsumption*" [7]. In other words, Kurtz et al. observed that if we apply an exhaustive set of tests, the dynamic subsumption tends to be stable and close to the true subsumption.

In our approach, we focus only on a mutation target at a time. *A mutation target is a language expression or statement in which it is possible to apply a set of mutations of one or more mutation operators.* For example, the binary expression `a + b` is a target because we can apply a set of mutations from one or more mutation operators. In case `a` and `b` are variables, examples of mutation operators and their respective mutations for the `a + b` target are described as follows. The *Arithmetic Operator Replacement* (AORB) mutation operator applies the following mutations: `a + b` $\mapsto$ `a - b`, `a + b` $\mapsto$ `a * b`, `a + b` $\mapsto$ `a / b`, and `a + b` $\mapsto$ `a % b`; two mutations are applied with the *Variable Deletion* (VDL); and two mutations are applied with the *Operator Deletion* (ODL) (e.g., `a + b` $\mapsto$ `a`, and `a + b` $\mapsto$ `b`).

A. Running Example

To better illustrate our approach, we use the `lexp && rexp` mutation target as a running example. The first step of our approach consists of isolating the target and implementing it in a tiny and trivial program (see Figure 1). Then, we create all possible mutations of all possible mutation operators for our target. Table I outlines all possible mutations applied to `lexp && rexp`.

```
1  public boolean and(boolean lexp, boolean rexp) {
2      return lexp && rexp;
3  }
```

Fig. 1: Program snippet containing the `lexp && rexp` target.

TABLE I: Mutations applied to the `lexp && rexp` mutation target.

Op.	Mutation	Short form
COR	`lexp && rexp` $\mapsto$ `lexp || rexp`	`COR ||`
COR	`lexp && rexp` $\mapsto$ `lexp == rexp`	`COR ==`
COR	`lexp && rexp` $\mapsto$ `lexp != rexp`	`COR !=`
COR	`lexp && rexp` $\mapsto$ `lexp ^rexp`	`COR ^`
COR	`lexp && rexp` $\mapsto$ `false`	`COR false`
COR	`lexp && rexp` $\mapsto$ `true`	`COR true`
COI	`lexp && rexp` $\mapsto$ `!(lexp && rexp)`	`COI !()`
ODL	`lexp && rexp` $\mapsto$ `lexp`	`ODL lexp`
ODL	`lexp && rexp` $\mapsto$ `rexp`	`ODL rexp`
VDL	`lexp && rexp` $\mapsto$ `lexp`	`VDL lexp`
VDL	`lexp && rexp` $\mapsto$ `rexp`	`VDL rexp`

Notice that eleven mutations of four different mutation operators can be applied to this single mutation target. Now, we need an exhaustive set of tests. Because it is a trivial program, automatic test generation tools can create tests that exercise the target with many different input values. However, for our running example, only four test cases are needed (see Figure 2). When executing all tests against all mutants, we end up with the kill matrix depicted in Table II.

```
1  public class TestSuite {
2      private Program p = new Program();
3      public void t1() {
4          assertEquals(true, p.and(true, true));
5      }
6      public void t2() {
7          assertEquals(false, p.and(true, false));
8      }
9      public void t3() {
10         assertEquals(false, p.and(false, true));
11     }
12     public void t4() {
13         assertEquals(false, p.and(false, false));
14     }
15 }
```

Fig. 2: Test suite generated for the `and` function.

TABLE II: Kill matrix of the `lexp && rexp` mutation target.

Mutation	t_1	t_2	t_3	t_4
COR ||		✗	✗	
COR ==				✗
COR !=	✗	✗	✗	
COR ^	✗	✗	✗	
COR true		✗	✗	✗
COR false	✗			
COI !()	✗	✗	✗	✗
ODL lexp		✗		
ODL rexp			✗	
VDL lexp		✗		
VDL rexp			✗	

In the kill matrix, each row represents a mutant, and each column represents a test case. The ✗ symbol indicates that the test of this column detects the mutation in the row.

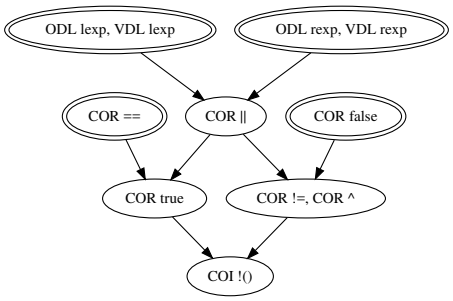


Fig. 3: DMSG of the `lexp && rexp` mutation target.

Kurtz et al. [7] depict the dynamic subsumption hierarchy with a directed graph. The *Dynamic Mutant Subsumption Graph* (DMSG) supports us to identify the minimal mutant set visually. Figure 3 illustrates the DMSG for our running example. Notice that there are four nodes (doubled line) subsuming all other nodes (single line). In case a non-subsumed node contains two or more mutations (e.g., see the node at the upper-left corner in Figure 3: `ODL lexp` and `VDL lexp`), we say that these mutations are *undistinguished*, and thus we can pick only one to compose the minimal set (either `ODL lexp` or `VDL lexp` in the abovementioned example). To determine the dynamic minimal mutant set, we need to get a mutation from each subsuming node in the graph.

So, we can state that by applying our approach to discover the dynamic subsumption relations for the mutation target `lexp && rexp`, the mutation tool should perform only four mutations (e.g., `COR ==`, `ODL lexp`, `ODL rexp`, and `COR false`) out of eleven possible mutations, yielding a reduction of 64% in the number of mutants. Notice that our approach yielded the same DMSG as the one presented by Just et al. [12].

B. Approach

In summary, our approach works as follows:

1) Given a target, we create a trivial and tiny program containing it;
2) We create all possible mutations of all possible mutation operators for the given target;
3) Then, we generate an exhaustive set of tests in order to exercise the mutation target;
4) We execute all tests against all mutants of the mutation target in order to obtain the *kill matrix*;
5) Finally, we use the kill matrix to determine the dynamic subsumption relations among the mutations applied to the mutation target.

Notice that we *isolate* a mutation target and try to determine subsumption relations considering only the mutations that can be applied to the target. Our hypothesis is that the subsumption relation found for one specific target might be less context-dependent, and in turn, generalizable to different programs in different domains. Therefore, we avoid subsumption relations from entire programs, classes, or functions, since they might be difficult to generalize.

Our approach to discover dynamic mutant subsumption relations of a mutation target is inherently empirical. We cannot guarantee that the minimal set of mutants—defined for a given mutation target—represents the true subsumption hierarchy. In addition, we cannot guarantee that the tiny and trivial program used to wrap the mutation target can be generalized to all possible programs. Next, we execute our approach to all method-level mutation targets where the MuJava tool can apply mutations.

C. Executing the Approach to Define Subsumption Relations

Table III illustrates all method-level mutation targets (53) in which the MuJava (version 4) mutation tool [8], [9] is able to apply mutations from one or more mutation operators. For each target, we list the set of mutation operators able to apply mutations into the target. For each operator, we provide the number of possible mutations (in parentheses) that such operator can apply into the target. For example, the *Logical Operator Replacement* (LOR) operator can apply two mutations into the `lexp | rexp` target.

When executing our approach to discover dynamic subsumption relations, we considered the parts of each target as:

- `exp`: unary expression, such as identifiers, variables, field access, array access, literals, or function calls;
- `lexp` and `rexp`: unary expressions, or binary expression;
- `lhs`: identifiers, variables, field access, or array access;
- `rhs`: unary expressions, or binary expression.

In this paper, we consider the type of expressions as arithmetic, (i.e., they have types `byte`, `char`, `short`, `int`, `long`, `float`, or `double`), boolean (tagged with `bool` in

TABLE III: 53 method-level mutation targets in which MUJAVA is able to apply mutations. We also present the mutation operators and the number of mutations that each operator is able to create in the corresponding target.

Mutation Target	Mutation Operators
`lexp + rexp`	AORB (4), VDL (2), CDL (2), ODL (2)
`lexp + rexp (obj)`	VDL (2), CDL (2), ODL (2)
`lexp - rexp`	AORB (4), VDL (2), CDL (2), ODL (2)
`lexp * rexp`	AORB (4), VDL (2), CDL (2), ODL (2)
`lexp / rexp`	AORB (4), VDL (2), CDL (2), ODL (2)
`lexp % rexp`	AORB (4), VDL (2), CDL (2), ODL (2)
`lexp > rexp`	ROR (7), COI (1)
`lexp >= rexp`	ROR (7), COI (1)
`lexp < rexp`	ROR (7), COI (1)
`lexp <= rexp`	ROR (7), COI (1)
`lexp == rexp`	ROR (7), COI (1)
`lexp == rexp (obj)`	ROR (1), COI (1)
`lexp == rexp (bool)`	ROR (1), COI (1), VDL (2), CDL (2), ODL (2)
`lexp != rexp`	ROR (7), COI (1)
`lexp != rexp (obj)`	ROR (1), COI (1)
`lexp != rexp (bool)`	ROR (1), COI (1), VDL (2), CDL (2), ODL (2)
`lexp && rexp`	COR (6), COI (1), VDL (2), CDL (2), ODL (2)
`lexp || rexp`	COR (6), COI (1), VDL (2), CDL (2), ODL (2)
`lexp & rexp`	LOR (2), VDL (2), CDL (2), ODL (2)
`lexp | rexp`	LOR (2), VDL (2), CDL (2), ODL (2)
`lexp ^ rexp`	LOR (2), VDL (2), CDL (2), ODL (2)
`lexp ^ rexp (bool)`	COR (5), COI (1), VDL (2), CDL (2), ODL (2)
`lexp >> rexp`	SOR (2), VDL (2), CDL (2), ODL (2)
`lexp << rexp`	SOR (2), VDL (2), CDL (2), ODL (2)
`lexp >>> rexp`	SOR (2), VDL (2), CDL (2), ODL (2)
`exp`	AOIS (4), AOIU (1), LOI (1)
`exp (bool)`	COI (1)
`+exp`	AODU (1), LOI (1), ODL (1)
`!exp`	COD (1), ODL (1)
`-exp`	AODU (1), LOI (1), ODL (1)
`~exp`	LOI (1), LOD (1), ODL (1)
`++exp`	AORS (1), AODS (1), LOI (1), ODL (1)
`exp++`	AORS (1), AODS (1), LOI (1), ODL (1)
`--exp`	AORS (1), AODS (1), LOI (1), ODL (1)
`exp--`	AORS (1), AODS (1), LOI (1), ODL (1)
`lhs += rhs`	ASRS (4), ODL (1), SDL (1)
`lhs -= rhs`	ASRS (4), ODL (1), SDL (1)
`lhs *= rhs`	ASRS (4), ODL (1), SDL (1)
`lhs /= rhs`	ASRS (4), ODL (1), SDL (1)
`lhs %= rhs`	ASRS (4), ODL (1), SDL (1)
`lhs <<= rhs`	ASRS (1), ODL (1), SDL (1)
`lhs >>= rhs`	ASRS (1), ODL (1), SDL (1)
`lhs >>>= rhs`	ASRS (2), ODL (1), SDL (1)
`lhs &= rhs`	ASRS (2), ODL (1), SDL (1)
`lhs |= rhs`	ASRS (2), ODL (1), SDL (1)
`lhs ^= rhs`	ASRS (2), ODL (1), SDL (1)
`stm_list`	SDL (1)
`while_stm`	SDL (1)
`if_stm`	SDL (1)
`for_stm`	SDL (1)
`return_stm`	SDL (1)
`case_stm`	SDL (1)
`try_stm`	SDL (1)

Table III), or object (tagged with `obj` in Table III) for other types and the null literal. For each mutation target, we manually create a tiny and trivial program to isolate it. The program is implemented with as few statements as possible. The reason for reducing the number of statements is to make the rest of the code to influence as little as possible the mutant subsumption behavior of the target analyzed. Figures 1, 4 and 5 illustrate three programs for the mutation targets `lexp && rexp`, `~exp`, and `lhs += rhs`. We used these programs to create subsumption relations for these targets.

Before executing our approach, we first disregard targets in which MUJAVA applies only one mutation, i.e., `exp (bool)`, `stm_list`, `while_stm`, `if_stm`, `for_stm`, `return_stm`, `case_stm`, and `try_stm`. Because they have only one mutation, we cannot achieve any reductions for these targets. We also disregard three targets due to MUJAVA compilation problems, i.e., `lexp >> rexp`, `lexp << rexp`, and `lexp >>> rexp`.

```
1  public int complement(int exp) {
2      return ~exp;
3  }
```

Fig. 4: Program snippet containing the `~exp` mutation target.

```
1  public int plusEquals(int lhs, int rhs) {
2      lhs += rhs;
3      return lhs;
4   }
```

Fig. 5: Program snippet containing the `lhs += rhs` mutation target.

After disregarding these 11 targets, we use MUJAVA to generate mutants for each program of the remaining 42 targets. We enable all method-level mutation operators available in the MUJAVA tool. To have an exhaustive test suite for each program, we use the EVOSUITE and RANDOOP automatic test generation tools. Despite the simplicity of the programs, we have configured these tools to generate as many inputs as possible. We set the generation timeout for both tools to 120 seconds. According to previous work, when considering more than 120 seconds, the coverage does not present significant variation [13]. Other settings were used to control the test case size (maximum of five statements per test) and quantity (1,000 tests per file). We used the default settings for the other parameters. Five test suites were generated with each tool to ensure a large number of test cases.

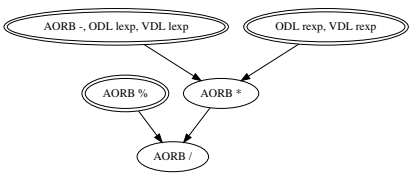
We then executed the tests, computed the kill matrix for each program, and subsequently captured the subsumption relation with the DMSG of each program. As a result, we defined a minimal set of mutations for each mutation target.

Table IV shows the minimal sets that our approach defined for the 42 mutation targets we studied. For instance, MUJAVA might perform up to eight mutations in the mutation target `lexp + rexp`. After the dynamic subsumption relation analysis, this mutation target needs only three out of eight possible mutations, yielding a reduction of 62.5%. The minimal set is composed by the following mutations: `AORB %`, `ODL lexp`, and `ODL rexp`. Figure 6 illustrates DMSGs our approach found for three targets, i.e., `lexp + rexp`, `lexp > rexp`, and `lhs <<= rhs`.

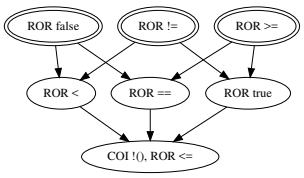
With the minimal sets of the mutation targets presented in Table IV, we implemented a new version of MUJAVA. We named this tool MUJAVA-M. The companion website [14] of this work has a link to download MUJAVA-M, as well as all the programs we developed, the generated test suite for each program, and the DMSGs.

IV. EVALUATION

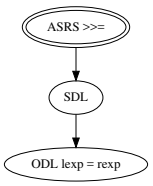
This section presents the evaluation of our approach.



(a) DMSG for the target `lexp + rexp`.



(b) DMSG for the target `lexp > rexp`.



(c) DMSG for the target `lhs <<= rhs`.

Fig. 6: Three DMSGs we found by applying our approach.

TABLE IV: Minimal mutants set for the 42 mutation targets analyzed.

Target	Minimal Set	Reduction
`lexp + rexp`	`AORB %, ODL lexp, ODL rexp`	62.5%
`lexp + rexp (obj)`	`ODL lexp, ODL rexp`	50.0%
`lexp - rexp`	`AORB %, ODL lexp, ODL rexp`	62.5%
`lexp * rexp`	`AORB /, ODL lexp, ODL rexp`	62.5%
`lexp / rexp`	`AORB %, AOIS *, ODL rexp`	62.5%
`lexp % rexp`	`AORB +, AORB -, AORB /, ODL lexp`	50.0%
`lexp > rexp`	`ROR false, ROR !=, ROR >=`	62.5%
`lexp >= rexp`	`ROR true, ROR ==, ROR >`	62.5%
`lexp < rexp`	`ROR false, ROR !=, ROR <=`	62.5%
`lexp <= rexp`	`ROR true, ROR ==, ROR <`	62.5%
`lexp == rexp`	`ROR false, ROR <=, ROR >=`	62.5%
`lexp == rexp (obj)`	`ROR !=`	50.0%
`lexp == rexp (bool)`	`ROR !=, ODL lexp, ODL rexp`	50.0%
`lexp != rexp`	`ROR true, ROR <, ROR >`	62.5%
`lexp != rexp (obj)`	`ROR ==`	50.0%
`lexp != rexp (bool)`	`ROR ==, ODL lexp, ODL rexp`	50.0%
`lexp && rexp`	`COR false, COR ==`	81.8%
`lexp || rexp`	`COR true, COR !=`	81.8%
`lexp & rexp`	`ODL lexp, ODL rexp`	66.7%
`lexp | rexp`	`ODL lexp, ODL rexp, LOR ^`	66.7%
`lexp ^ rexp`	`LOR |`	83.3%
`lexp ^ rexp (bool)`	`COR false, COR ||`	80.0%
`exp`	`AOIU -exp`	83.3%
`+exp`	`LOI  exp`	75.0%
`!exp`	`COD exp`	50.0%
`-exp`	`AODU exp`	66.7%
`~exp`	`LOD exp`	75.0%
`++exp`	`AODS exp`	75.0%
`exp++`	`LOI  exp`	75.0%
`--exp`	`AODS exp`	75.0%
`exp--`	`LOI  exp`	75.0%
`lhs += rhs`	`ASRS -=, ASRS %=, ODL =`	50.0%
`lhs -= rhs`	`ASRS +=, ASRS %=, ODL =`	50.0%
`lhs *= rhs`	`ASRS /=, ASRS %=, ODL =`	50.0%
`lhs /= rhs`	`ASRS *=, ASRS %=, ODL =`	50.0%
`lhs %= rhs`	`ASRS +=, ASRS -=, ASRS *=, ASRS /=`	33.3%
`lhs <<= rhs`	`ASRS >>=,`	66.7%
`lhs >>= rhs`	`ASRS <<=, ODL =, SDL`	0.0%
`lhs >>>= rhs`	`ASRS >>=, ASRS <<=`	50.0%
`lhs &= rhs`	`ODL =, SDL`	50.0%
`lhs |= rhs`	`ODL =, ASRS ^=, SDL`	25.0%
`lhs ^= rhs`	`ASRS |=`	75.0%

A. Definition

To better structure our evaluation, we rely on the Goal, Question, Metrics methodology [15]. The goal of our experiment consists of analyzing our approach, implemented by MUJAVA-M, with the purpose of evaluating the dynamic subsumption relations we found with respect to the number of mutants discarded, the correctness of this discard, and time savings, from the point of view of testers in the context of applying mutation testing to Java open source programs.

We address the following research questions:

- RQ_1*: How many mutants are likely-subsumed?*

To answer this question, we count the number of mutants generated by MUJAVA and MUJAVA-M for each mutation target. Notice that answering RQ_1 is important because it allows us to estimate the amount of computational effort saved. However, the dynamic subsumption relations we embedded in MUJAVA-M must be effective in the sense that they should not discard important mutants that would be in the minimal set. To better understand this point, we formulate the following research question:

- RQ_2*: How many mutants are incorrectly discarded from the minimal set?*

To answer RQ_2, we rely on the definition of minimal tests set [3]. The minimal tests set necessary to kill the minimal mutants set must also kill all the mutants in the full mutants set. Thus, we generate this minimal tests set and execute against the full mutants set. If a mutant from the full mutants set survives, this means that we incorrectly discarded this mutant. We compute the frequency of these cases and manually analyze them to understand the reasons of why the mutant was discarded incorrectly.

Our focus is on mutant subsumption relations with respect to the mutation targets (instead of entire programs, classes, or functions). This makes MUJAVA-M potentially scalable. However, we need empirical evidence to determine the degree to which it scales. To better understand the real gain in reducing the total time of the mutation analysis, we elaborate the following research question:

- RQ_3*: What are the time savings of eliminating likely-subsumed mutants?*

We answer this question by executing MUJAVA and MUJAVA-M as full mutation analysis tools. That is, we not only use the tools to generate the mutants, but we also use them to execute the test suite against the mutants. Then, we measure the time required to perform mutant generation and mutation analysis on both tools.

B. Planning

We use five large open source programs to carry out our evaluation. Table V illustrates the programs studied, i.e., *joda-time*, *commons-math*, *commons-lang*, *h2*, and *javassist*. These programs varies in size and application domain. We performed the evaluation on Intel Core i5-7400 with 8 GB of RAM equipped with Linux 3.10.0 operating system. We used MUJAVA and MUJAVA-M command-line version. In both, all method-level mutation operators were enabled.

After generating mutants in both tools, we need to calculate the mutants incorrectly discarded by MUJAVA-M. Thus, we need to execute a minimal test set—necessary to kill

TABLE V: Programs we used in our evaluation.

Project	Version	Lines of Code (LOC)
joda-time	2.10.1	28,790
commons-math	3.6.1	100,364
commons-lang	3.6	27,267
h2	1.4.199	134,234
javassist	3.20	35,249

the MUJAVA-M mutants—against the mutants generated by MUJAVA. To find out the minimal test set, we rely on EVOSUITE's [16] Regression test suite generation (EVOSUITER) version 1.0.6. EVOSUITER is a specialization of EVOSUITE that tries to generate one test revealing the difference between two versions of a Java class. For instance, given two Java classes with a small syntactic difference in code, say a mutant, EVOSUITER tries to find a test case that exposes this behavioral difference between the two files. We set up 60 seconds as the time limit to EVOSUITER generate tests. We used the default values for the other parameters.

In case the mutant survives the test generated by EVOSUITER, we need to know if it is an equivalent mutant. Detecting equivalent mutants is a well-known undecidable problem [17]. Besides, equivalent mutants contribute negatively to the confidence assessment of the reduction applied. To minimize this problem, we avoid some equivalent mutants by using the Trivial Compiler Equivalence (*TCE*) [18]. *TCE* is a sound tool because it checks whether the bytecodes of the original program and the mutant are the same. This eliminates the possibility of false positives. However, *TCE* cannot identify equivalent mutants that have different bytecodes, which may yield false negatives.

In summary, to answer RQ_1 and RQ_2 the planning is the following: generate the mutants with MUJAVA and MUJAVA-M, then generate the minimal tests set with EVOSUITER, execute the test generated against the MUJAVA mutants, detect equivalence with *TCE*, and calculate the surviving mutants. Because it is a computationally costly experiment, we leave the programs running for seven days for each subject. Consequently, the number of randomly selected files of each subject varied. In total, 168 class files were evaluated.

Regarding RQ_3, applying full mutation analysis to all possible mutants of the programs presented in Table V is infeasible. So, we randomly selected five classes from each subject to compute the time savings. To perform the analysis, we not only generated the mutants with MUJAVA and MUJAVA-M, but also executed the subjects test suite against each mutant.

C. Procedure

To carry out our evaluation we follow two different procedures. The first procedure is used to answer the research questions RQ_1 and RQ_2. Next, we explain how we proceed:

1) A Java class is the MUJAVA unity of work, thus we need to generate the mutants for the whole class. Applying all possible mutants to all files in a large program is clearly infeasible. This way we randomly selected a set of Java class files for each subject.
2) With the classes selected, we executed MUJAVA and MUJAVA-M against these classes to generate the full set and the minimal set of mutants, respectively. We enabled all method-level mutation operators in both tools.
3) Next, we added up the mutants of MUJAVA and MUJAVA-M grouped by target. For instance, for each target t in the class file, MUJAVA generated the full set $M = \{m_1, m_2, m_3, m_4\}$ containing all mutants, and MUJAVA-M generated the minimal set $\bar{M} = \{m_1, m_2\}$ containing only the sufficient mutants according to the dynamic subsumption relations found previously by our approach (Section III).
4) We now proceed to create a minimal test set. As explained, the minimal test set necessary to kill the minimal mutants set must also kill the full mutants set [3]. Thus, we use EVOSUITER to create a test case for each mutant in the minimal set ($\bar{M}$). We provide the original program and a mutant from $\bar{M}$, and EVOSUITER generates a test containing only one test case to kill the mutant. We repeat this process for all mutants in $\bar{M}$. At the end, we group the generated tests to create a minimal test suite $\bar{T}$ for the minimal set of mutants $\bar{M}$.
5) To validate if the mutants of $\bar{M}$ indeed represent the minimal mutants set for the target, we execute $\bar{T}$ against M. In case all mutants of M get killed, we confirm that $\bar{M}$ is a reliable representation of M. But if a mutant of M survives, it represents a fail in our approach. For example, if only the m_1, m_2, and m_3 mutants of M are killed by suite $\bar{T}$, only 75% of the mutants in the full set were killed. This means that m_4 is a useful mutant and should not be discarded from the minimal set. An exception occurs when m_4 is an equivalent mutant. In this case m_4 is useless to the mutation test. This way, we executed *TCE* against the mutants of M that survived to $\bar{T}$. If *TCE* identifies a mutant as equivalent, we take this mutant out of the analysis. If *TCE* does not mark a mutant as equivalent, then we understand that this mutant represents an error in our reduction and it should be part of the minimal mutant set.
6) To understand if our approach has eliminated important mutants, we verified the number of mutants not generated by MUJAVA-M but should be part of the minimal set. We also manually verified a subset of these incorrectly deleted mutants and did a qualitative analysis where we discuss the possible causes that led to the approach failing.

To automate the process described above, we create a script that executes all the steps. In some exceptional scenarios we discard the target. Below we list these scenarios:

- In case EVOSUITER cannot identify a test case to distinguish the original program and a mutant in a limit of 60 seconds, we did not proceed with the analysis of the target.
- We execute the minimal test suite against the original

program to confirm they are passing. We repeat this process three times to reduce the presence of *flaky* tests [19]. In case we identify *flaky* tests, or the test suite does not pass in the original program, we do not proceed with the analysis of the target.

To answer RQ_3, we proceed as follows:

1) We randomly selected five class files from each subject.
2) Next, we executed the MUJAVA and MUJAVA-M to generate the mutants. Again, we enable all method-level mutation operators.
3) Then, we complete the mutation analysis executing original test suites of the subjects against all the mutants generated by MUJAVA and MUJAVA-M.

D. Replication Package

We provide a comprehensive report of our evaluation in our companion website [14]. All data, setups, scripts, spreadsheets are also available for replication and reproduction purposes.

E. Results

1) RQ_1*: How many mutants are likely-subsumed?:* Table VI presents the number of mutants generated by MUJAVA and MUJAVA-M for each subject. In particular, we analyzed 2,341 occurrences of mutation targets in 168 classes. MUJAVA generated 10,818 mutants, which gives an average of 4.62 mutants per target. MUJAVA-M, in its turn, generated 3,848 mutants for the same set of mutation targets, i.e., an average of 1.64 mutants per target. This way, MUJAVA-M achieved a reduction of 64.43% in the number of generated mutants when compared to the original version of MUJAVA.

TABLE VI: Number of mutants per subject.

Project	Classes	Occurrences of mut. targets	MUJAVA	MUJAVA-M
joda-time	51	929	4,712	1,691
commons-math	35	425	1,924	654
commons-lang	33	631	2,500	943
h2	19	74	327	117
javassist	35	282	1,355	443
Total	168	2,341	10,818	3,848

Table VII illustrates the occurrences of 32 mutation targets we analyzed in the 168 classes. Although we provide subsumption relations for 42 targets (see Table IV), we could only analyze 32 basically for two reasons. First, we did not find occurrences of some targets in the 168 classes (e.g., `lhs >>>= rhs`); second, for some targets occurrences, EVOSUITER could not generate the minimal test set, which is a prerequisite to check the effectiveness of our minimal mutation sets. Table VII also presents the reductions per target applied by MUJAVA-M. The most common target is `exp`. We identified 1,307 `exp` occurrences. The reduction was 71.15% on average.

Notice that the reduction percentage must be equal to or lower than the minimal sets that our approach defined in Table IV. This is because the full set of mutations may not always be applied, so the reduction percentage may vary

TABLE VII: General results for RQ_1 and RQ_2.

Mutation Target	Occurrences	Reduction	Effectiveness
`lexp + rexp`	111	58.89%	98.88%
`lexp + rexp (obj)`	86	40.07%	100.00%
`lexp - rexp`	74	60.57%	99.64%
`lexp * rexp`	55	58.54%	99.50%
`lexp / rexp`	35	59.62%	99.23%
`lexp % rexp`	13	47.47%	96.97%
`lexp > rexp`	19	62.50%	100.00%
`lexp >= rexp`	26	62.50%	100.00%
`lexp < rexp`	35	62.50%	100.00%
`lexp <= rexp`	16	62.20%	100.00%
`lexp == rexp`	34	62.50%	100.00%
`lexp == rexp (obj)`	234	51.72%	98.53%
`lexp == rexp (bool)`	4	40.00%	100.00%
`lexp != rexp`	14	62.50%	100.00%
`lexp != rexp (obj)`	85	51.15%	98.85%
`lexp != rexp (bool)`	1	40.00%	100.00%
`lexp && rexp`	13	55.56%	100.00%
`lexp || rexp`	25	55.56%	100.00%
`lexp & rexp`	33	64.32%	100.00%
`lexp | rexp`	6	40.00%	100.00%
`lexp ^ rexp`	6	66.67%	100.00%
`exp`	1,307	71.15%	76.31%
`!exp`	27	50.00%	100.00%
`-exp`	38	58.70%	96.74%
`~exp`	13	58.06%	100.00%
`exp++`	4	71.43%	100.00%
`--exp`	1	80.00%	100.00%
`exp--`	4	83.33%	83.33%
`lhs += rhs`	11	37.93%	96.55%
`lhs -= rhs`	8	38.10%	100.00%
`lhs /= rhs`	2	33.33%	100.00%
`lhs ^= rhs`	1	66.67%	66.67%

downwards. For example, consider the `lexp * rexp` target. According to the minimal set applied by MUJAVA-M, the reduction for this mutation target can reach 62.5% (Table IV). However, when analyzing this target in real programs, this percentage dropped to 58.54%. Notice that eight possible mutations can be applied by MUJAVA for this mutation target: `AORB +`, `AORB -`, `AORB /`, `AORB %`, `ODL lexp`, `ODL rexp`, `VDL lexp` or `CDL lexp`, and `VDL rexp` or `CDL rexp`. Figure 7 presents an instance of the `lexp * rexp` mutation target from the *joda-time* subject. Here, `VDL lexp`, `CDL lexp`, `VDL rexp`, and `CDL rexp` are not applicable in this context and thus these mutants are dropped from the full set generated by MUJAVA.

```
1  (wrappedValue - thisValue) * getUnitMillis();
```

Fig. 7: Code snippet with an instance of the target `lexp * rexp`.

In general, even though we are focusing only on mutation targets, we achieve significant reductions when considering the total number of generated mutants (see column "Reduction" in Table VII). However, we may have been discarded important mutants for the mutation analysis. In this sense, to better understand to what extent our reductions are indeed focusing only on redundant mutants, we now answer RQ_2.

2) RQ_2*: How many mutants are incorrectly discarded from the minimal set?:* Table VII also presents numbers with respect to the effectiveness of MUJAVA-M, i.e., we check whether the mutants discarded by our tool were indeed discarded correctly. Column "Effectiveness" presents these results. This percentage represents the number of mutants

generated by MUJAVA that were killed by the minimal test set. In the ideal scenario, the minimal test set should kill all MUJAVA mutants.

According to Table VII, we achieve 100% of effectiveness in 20 targets (62.5%). In 29 out of 32 targets we achieve effectiveness greater than 95%. On the other hand, we achieved only 76.31% of effectiveness for the `exp` target (the one more common in the subjects we studied, i.e., 1,307 occurrences). Notice that `exp` is a very generic constructor that can be, for example, a variable that stores the index of an array, i.e., `arr[i]`. Thus, according to the context in which the mutation target is placed, the dynamic subsumption behavior can vary significantly.

To better understand the reasons of why MUJAVA-M wrongly discarded some mutants, we performed a manual analysis in a sample of them. Now we discuss two examples and explain why our approach was not effective in these cases.

a) Example 1: In the first example, we present an error in a minimal set with the `lexp + rexp` mutation target. According to Table VII, this target occurred 111 times and had an effectiveness of 98.88%. Figure 8 presents a code snippet of the `InnerClassesAttribute` class in the *javassist* project. In the `append` method, we have the expression `len + 8` (Line 4). This expression is used to define the length of an array in a local variable. The minimal mutation set defined for this target would be: `ODL lexp` (`len + 8` $\mapsto$ `len`), `ODL rexp` (`len + 8` $\mapsto$ `8`), and `AORB %` (`len + 8` $\mapsto$ `len % 8`). However, the minimal test set did not kill the `AORB *` (`len + 8` $\mapsto$ `len * 8`) mutation.

Initially, we thought that the mutant resulting from the `AORB *` mutation was equivalent, because it simply increases the array storage space and, in principle, would not change the behavior of the program. However, in Line 8 of Figure 8 the `newdata` reference is passed as a parameter to a parent class method that uses `newdata` to change a byte array field. Notice that in the parent class there is also a public method that returns the size of this field. Thus, a test that checks the length of this field could kill the `AORB *` mutation, which in this context turns this mutation not subsumed by any mutant of the minimal mutant set proposed.

b) Example 2: In the second example, we discuss an error when defining the minimal set for an instance of the target `lhs ^= rhs`. This target occurred only once in the subjects studied. Figure 9 presents a code snippet of the `BooleanUtils` class of the project *commons-lang*. At Line 5 of the `xor` method there is the following statement: `result ^= element`. This statement applies an exclusive disjunction logic operation among all elements of the array. The minimal mutation set for this target is made up of just one mutation: `ASRS |=` (`result ^= element` $\mapsto$ `result |= element`). However, the minimal test set did not kill the `ASRS &=` (`result ^= element` $\mapsto$ `result &= element`) mutation.

Lindström and Márki [20] suggest that the subsumption relations cannot hold when the mutated statements are re-executed (in the context of *strong mutation* [21]). If the mutated instruction is executed more than once by any test execution, we cannot determine the future state of the program. Since our subsumption relations were obtained from a program where the final state is known, they are not sufficient to represent the mutation within a repeating context.

Although evaluating only the mutation target decreases the influence of the context of the program, we found that, in some cases, we indeed need some context information in order to define subsumption relations for targets within more specific contexts, such as array initialization and field access.

```
1  public void append(int i, int o, int n, int f) {
2      byte[] data = get();
3      int len = data.length;
4      byte[] newData = new byte[len + 8];
5      for (int i = 2; i < len; ++i)
6          newData[i] = data[i];
7      ...
8      set(newData);
9  }
```

Fig. 8: Code snippet of from *javassist* project

```
1  public static boolean xor(final boolean... array) {
2      ...
3      boolean result = false;
4      for (final boolean element : array) {
5          result ^= element;
6      }
7      return result;
8  }
```

Fig. 9: Code snippet of from *commons-lang* project

3) RQ_3*: What are the time savings of eliminating likely-subsumed mutants?:* To answer RQ_3, we applied the full mutation analysis with MUJAVA and MUJAVA-M (differently from RQ_1 and RQ_2, where we focused only on the mutants generation). By full mutation analysis, we mean the mutants generation and the execution of the original test suite of each subject against the mutants. We randomly selected five class files of each project. However, due to several crashes during the test suite execution, we are not able to provide numbers regarding the *h2* subject. This way, our analysis regards 20 class files instead of 25.

Table VIII presents the results by each class file analyzed. The full set part presents the number of generated mutants ("Mut."), the time necessary to generate these mutants ("Time(G)"), and the time necessary to execute the test suite against all mutants ("Time(E)"). The minimal set part presents the same columns for the minimal mutation set. Table VIII also presents the reductions with respect to the number of mutants ("Mut. %"), time to generate the mutants ("Time(G) %"), and time to execute the test suite ("Time(E) %").

In total, MUJAVA generated 5,053 mutants and MUJAVA-M generated 2,234 mutants. This represents a reduction of 55.79% in the number of mutants. The time to generate all mutants on MUJAVA was two minutes and 30 seconds. In MUJAVA-M, this number dropped to one minute and 35 seconds, which represents a gain of 36.03% in mutant generation time.

TABLE VIII: Results for RQ_3.

Class	Project	Full set			Minimal set			Reductions		
		Mut.	Time (G)	Time (E)	Mut.	Time (G)	Time (E)	Mut. %	Time (G) %	Time (E) %
MutableDateTime	joda-time	844	31.372 s	0:29:44	432	19.848 s	0:16:41	48.82%	36.73%	43.89%
MutableInterval	joda-time	127	5.054 s	0:05:13	71	3.986 s	0:03:33	44.09%	21.13%	31.95%
GJMonthOfYearDateTimeField	joda-time	13	1.577 s	0:00:41	11	1.540 s	0:00:35	15.38%	2.35%	14.47%
ISOYearOfEraDateTimeField	joda-time	138	3.908 s	0:06:21	85	3.013 s	0:04:11	38.41%	22.90%	34.12%
UnsupportedDateTimeField	joda-time	108	7.508 s	0:04:01	59	4.482 s	0:02:45	45.37%	40.30%	31.54%
ClassPathUtils	commons-lang	24	1.985 s	0:06:29	20	1.894 s	0:05:20	16.67%	4.58%	17.74%
ConstantInitializer	commons-lang	24	1.890 s	0:04:17	13	1.725 s	0:03:29	45.83%	8.73%	18.68%
MutableInt	commons-lang	235	5.528 s	0:48:40	131	4.519 s	0:31:20	44.26%	18.25%	35.62%
MutableObject	commons-lang	41	2.207 s	0:06:11	18	1.934 s	0:04:51	56.10%	12.37%	21.56%
IEEE754rUtils	commons-lang	456	7.212 s	1:18:00	216	4.761 s	0:38:25	52.63%	33.99%	50.75%
Subroutine	javassist	45	2.269 s	0:08:10	27	2.105 s	0:04:56	40.00%	7.23%	39.59%
TypeAnnotationsWriter	javassist	110	3.285 s	0:19:56	86	3.029 s	0:15:35	21.82%	7.79%	21.82%
MapMaker	javassist	2,018	42.581 s	5:18:00	717	19.983 s	2:04:00	64.47%	53.07%	61.01%
Keyword	javassist	21	1.686 s	0:02:23	8	1.351 s	0:00:42	61.90%	19.87%	70.87%
Handler	javassist	137	4.532 s	0:19:54	79	3.745 s	0:10:23	42.34%	17.37%	47.82%
DefaultIterativeLinearSolverEvent	commons-math	47	2.819 s	0:30:31	24	2.365 s	0:22:52	48.94%	16.11%	25.07%
MaxIter	commons-math	39	2.084 s	0:59:23	17	1.756 s	0:24:56	56.41%	15.74%	58.01%
SimpleValueChecker	commons-math	190	4.885 s	3:08:00	61	2.914 s	1:29:00	67.89%	40.35%	52.66%
LeastSquaresConverter	commons-math	276	7.476 s	5:47:00	105	4.520 s	2:38:00	61.96%	39.54%	54.47%
FunctionUtils	commons-math	160	9.736 s	4:18:00	54	5.571 s	1:37:00	66.25%	42.78%	62.40%
Total		5,053	0:02:30	24:00:54	2,234	0:01:35	10:58:33	55.79%	36.47%	54.30%

It is well-known that a critical problem of mutation testing analysis is the computational time to execute the test suite. Here, MUJAVA-M achieves a promising reduction. Whilst MUJAVA took one day to execute the test suite against all mutants, MUJAVA-M was able to reduce this time by 52.53%, i.e., it took 11 hours on this task.

F. Discussion

Approximated DMSGs. The subsumption relations found by our approach are dependent on the test suite's quality. We use automatic test generation tools to create suites that exhaustively exercise the mutation target. However, automatic test generation tools have limitations. To better illustrate this scenario, consider the DMSG showed in Figure 6a. This DMSG presents the subsumption relations among the mutations of the `lexp + rexp` target. For this mutation target, the test case where `lexp` and `rexp` are equal to two—i.e., $(2, 2)$—does not detect the `AORB *` mutation, since $2+2 = 4$, as well as $2 * 2 = 4$. However, the test case $(2, 2)$ detects the `AORB -` mutation, since $2+2 = 4$ and $2-2 = 0$. The test case $(2, 0)$ does not detect the `AORB -` mutation, since $2 + 0 = 2$ and $2 - 0 = 2$. On the other hand, this test case detects the `AORB *` mutation, i.e., $2 + 0 = 2$ and $2 * 0 = 0$. Therefore, we can conclude that `AORB -` does not subsume `AORB *` and vice-versa, differently from what is suggested in the DMSG in Figure 6a. We found this subsumption relation (`AORB -` subsuming `AORB *`) because the automatic test generation tools did not generate the test case $(2, 2)$. Notice that there is a higher probability of generating test cases that do not detect `AORB -`, e.g., $(x, 0)$ for any value of x than generating test cases that do not detect `AORB *`, i.e., $(0, 0)$ and $(2, 2)$. This way, detecting `AORB -` is more difficult than detecting `AORB *`. This is why we found that `AORB -` subsumes `AORB *`.

Still, we obtained 98.88% of effectiveness for the `lexp + rexp` mutation target. Therefore, the subsumption relation we found between `AORB -` and `AORB *` does not significantly reduce the effectiveness of the minimal set in representing the full mutation set.

Last but not least, although the DMSGs obtained are approximations to the true subsumption relations, in most cases, they were close enough to obtain 100% of effectiveness in almost all mutation targets we analyzed.

Comparison with weak mutation. In our approach, we use tiny programs to isolate the mutation targets and obtain approximations for subsumption relations. In most cases, these programs contain public methods that only return the result of the instruction with the mutation target. In these programs, the test cases assert the program's output immediately after the mutation is executed, similarly to the definition of weak mutation, where the difference in the state after the mutation execution is sufficient to determine whether the mutant was detected. However, our proposal differs from weak mutation because we can take the entire program context into account (not only the context right after the execution of the mutated target that infected the program's state).

Generalizability of the approach. Our approach allows us to obtain the subsumption relations of any mutation target covered by the tests generated by the RANDOOP and EVOSUITE tools. We isolated the targets in tiny programs to obtain subsumption relations with the least influence of the context. In this sense, our hypothesis is that the subsumption relations found in this way are maintained in most contexts where the target can be found. The proposed approach allows us to find subsumption relations in any context, for example, in a mutation target within the scope of a repetition structure.

G. Threats to Validity

The set of projects we used represents a threat to external validity. Also, we did not evaluate all files of all projects. To increase diversity, we consider projects of different sizes and domains. As another threat to external validity, we focused only on method-level operators of only one tool, i.e., MUJAVA.

In some cases MUJAVA generates mutants that do not compile or fails to generate some mutants, representing a threat to internal validity. So, the DMSGs that we identified may change in case we use another mutation testing tool.

Subsumption relations were determined in isolated targets in tiny programs. These tiny programs do not reflect all the contexts in which the targets could be inserted. For example, our programs consider only the `int` type for arithmetic expressions. Thus, the absence of targeting programs in different contexts is a threat to internal validity. To reduce this threat, we have verified the effectiveness of the minimal mutation sets to represent the full mutation set.

We only considered in this study mutation targets that did not generate flaky tests and that EVOSUITER could generate the minimal test sets. This represents a threat to internal validity. This decision was necessary to assess the effectiveness of the reductions. The minimal test sets also poses a threat to internal validity. This is because computing minimal mutant sets for all possible test sets is computationally hard [3]. Thus, the EVOSUITER can generate the test set which is minimal but not *minimum* [3].

The mutants that survived the minimal test sets also represents a threat. Despite running *TCE* to identify equivalent mutants, *TCE* cannot detect all equivalent mutants due to the undecidability of the *Equivalent Mutant Problem* [22].

Some targets did not appear frequently in our evaluation. For instance, the mutation targets `--exp` and `lhs ^= rhs`, occurred only once. So, the effectiveness of the minimal set defined for these targets may not hold for general cases. We intend to perform other studies to evaluate these targets.

V. RELATED WORK

Zhu et al. [23] investigated different compression techniques to speed up mutation testing. They have introduced six compression strategies based on two clustering algorithms and three mutant selection strategies. Just et al. [24] described three methods to: (i) detect test-equivalent mutants by monitoring infected execution states, (ii) detect test-equivalent mutants by monitoring propagation of infected execution states in com-pound expressions during execution on the unmutated program, and (iii) reduce the number of partitioning mutants with identically infected state. They implemented the optimizations in the Major tool and empirically evaluated them on 14 open source programs. The optimizations reduced the mutation analysis time by 40% on average. We derive a number of dynamic subsumption relationships using our approach to avoid the generation of redundant mutants.

Kaminski et al. [5] defined a sufficient replacements for ROR mutations. Just et al. [6] present sufficient sets of non-redundant mutations for the COR and UOI operators. These subsumption hierarchies are defined by manually analyzing the combinations of all possible input situations. However, we discuss in this paper that, in several other cases, analyzing all possible combinations is prohibitive due to the high costs.

Just and Schweiggert [10] presented a study that analyzes the effect of redundant mutants on mutation analysis efficiency, mutation score, and mutation coverage ratio. The authors show that the mutants generated by COR, ROR, and UOI have a mean ratio of 45% of the total mutants generated. Using the sufficient set of non-redundant mutations for these operators, the number of mutants was reduced by 27% overall. Just and Schweiggert also show that redundant mutants worsen the accuracy of the mutation score. In this paper, we show approximations of the subsumption hierarchy for 32 mutation targets. We also conducted a study to show the effectiveness of a test suite that detects all mutants in the sufficient set of non-redundant mutants to detect all generated mutants.

Several other approaches in the literature suggest cost reduction strategies for mutation analysis [25]. Offutt et al. [26] presented an empirical approach to define an appropriate set of selective mutation operators. The idea was to randomly select a subset of mutation operators [27], [28]. Perez et al. [29] explored Evolutionary Mutation Testing to reduce the number of mutants to be executed. Namin et al. [30] formulated the selective mutation problem as a statistical problem. They applied linear statistical approaches to identify a subset of 28 mutation operators for *C*. However, in a recent empirical study Gopinath et al. [31] found no differences in effectiveness between selective mutation and random selection. The main challenge in reducing the mutants set is not losing useful information. Just et al. [32] state that existing approaches to selective mutation take no account of program context and this is fundamental to avoid losing useful information. Our approach might also lose useful information, i.e., an useful mutant might be discarded. In this context, we achieved 100% of effectiveness in 20 (62.5%) out of 32 targets.

VI. CONCLUSIONS AND FUTURE WORKS

We proposed an approach to discover dynamic mutant subsumption relations among mutants in a specific mutation target. We isolate a mutation target, generate all possible mutations for this target, and apply an exhaustive set of tests to discover the dynamic mutant subsumption relation, i.e., a minimal mutant set for such target. We empirically defined the minimal mutant sets for 42 method-level mutation targets of the MUJAVA mutation testing tool. Then, we implemented a new version of the tool which we call MUJAVA-M. Then, we executed MUJAVA-M against 168 classes of five open-source projects and evaluate the tool in terms of reducing the number of mutants, effectiveness, and time speedup. Our findings indicate that even focusing only on a mutation target at a time, the reductions in the number of generated mutants can be significant. We analyzed 2,341 occurrences of 32 mutation targets and MUJAVA-M achieved 100% of effectiveness in 20 out of 32 targets, and more than 95% in 29 mutation targets. Also, MUJAVA-M achieved a reduction of 52.53% in the time to execute the mutation analysis.

Future work include carrying out new experiments to use the targets not found in the projects we studied in this paper. We also intend to use other mutation tools such as *Major* [33] and *PIT* [34], as well as class-level mutation operators [35].

REFERENCES

[1] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. Le Traon, "Threats to the validity of mutation-based test assessment," in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2016, pp. 354–365.

[2] M. Kintis, M. Papadakis, and N. Malevris, "Evaluating mutation testing alternatives: A collateral experiment," in *Proceddings of the 17th Asia Pacific Software Engineering Conference (APSEC)*. IEEE, 2010, pp. 300–309.

[3] P. Ammann, M. E. Delamaro, J. Offutt *et al.*, "Establishing theoretical minimal sets of mutants," in *Proceedings of the International Conference on Software Testing, Verification, and Validation*. IEEE, 2014, pp. 21–30.

[4] M. Papadakis and N. Malevris, "An empirical evaluation of the first and second order mutation testing strategies," in *Proceedings of the 3rd International Conference on Software Testing, Verification, and Validation workshop*. IEEE, 2010, pp. 90–99.

[5] G. Kaminski, P. Ammann, and J. Offutt, "Better predicate testing," in *Proceedings of the 6th International Workshop on Automation of Software Test*. ACM, 2011, pp. 57–63.

[6] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Do redundant mutants affect the effectiveness and efficiency of mutation analysis?" in *Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2012, pp. 720–725.

[7] B. Kurtz, P. Ammann, M. E. Delamaro, J. Offutt, and L. Deng, "Mutant subsumption graphs," in *Proceedings of the seventh international conference on software testing, verification and validation workshops*. IEEE, 2014, pp. 176–185.

[8] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "MuJava: an automated class mutation system," *Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 97–133, 2005.

[9] ——, "MuJava: a mutation system for Java," in *Proceedings of the 28th International Conference on Software Engineering*. ACM, 2006, pp. 827–830.

[10] R. Just and F. Schweiggert, "Higher accuracy and lower run time: efficient mutation analysis using non-redundant mutation operators," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 490–507, 2015.

[11] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[12] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis," in *Proceedings of the 23rd International Symposium on Software Reliability Engineering*. IEEE, 2012, pp. 11–20.

[13] G. Soares, R. Gheyi, E. Murphy-Hill, and B. Johnson, "Comparing approaches to analyze refactoring activity on software repositories," *Journal of Systems and Software*, vol. 86, no. 4, pp. 1006 – 1022, 2013.

[14] "Replication package repository." 2020. [Online]. Available: https://easy-software-ufal.github.io/subsumption-relations/

[15] V. R. Basili, G. Caldiera, and H. D. Rombach, "The Goal Question Metric approach," in *Encyclopedia of Software Engineering*. Wiley, 1994.

[16] G. Fraser and A. Arcuri, "EvoSuite: automatic test suite generation for object-oriented software," in *Proceedings of the Foundations of Software Engineering*. ACM, 2011, pp. 416–419.

[17] T. Budd and D. Angluin, "Two notions of correctness and their relation to testing," *Acta Informatica*, vol. 18, no. 1, pp. 31–45, 1982.

[18] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. L. Traon, and M. Harman, "Detecting trivial mutant equivalences via compiler optimisations," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2017.

[19] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 643–653.

[20] B. Lindström and A. Márki, "On strong mutation and the theory of subsuming logic-based mutants," *Software Testing, Verification and Reliability*, vol. 29, no. 1-2, p. e1667, 2019.

[21] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[22] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Jozala, "Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation," *IEEE Transactions on Software Engineering*, vol. 40, no. 1, pp. 23–42, 2014.

[23] Q. Zhu, A. Panichella, and A. Zaidman, "An investigation of compression techniques to speed up mutation testing," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, 2018, pp. 274–284.

[24] R. Just, M. D. Ernst, and G. Fraser, "Efficient mutation analysis by propagating and partitioning infected execution states," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 315–326.

[25] A. V. Pizzoleto, F. C. Ferrari, 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, p. To appear, 2019.

[26] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 5, no. 2, pp. 99–118, 1996.

[27] A. Mathur, "Performance, effectiveness, and reliability issues in software testing," in *Proceedings of the 15th Annual International Computer Software & Applications Conference*. IEEE, 1991, pp. 604–605.

[28] W. E. Wong and A. P. Mathur, "Reducing the cost of mutation testing: An empirical study," *Journal of Systems and Software*, vol. 31, no. 3, pp. 185–196, Dec. 1995.

[29] P. Delgado-Pérez, S. Segura, and I. Medina-Bulo, "Assessment of C++ object-oriented mutation operators: A selective mutation approach," *Software Testing, Verification and Reliability*, vol. 27, no. 4-5, 2017.

[30] A. Siami Namin, J. H. Andrews, and D. J. Murdoch, "Sufficient mutation operators for measuring test effectiveness," in *Proceedings of the 30th International Conference on Software Engineering*. ACM, 2008, pp. 351–360.

[31] R. Gopinath, I. Ahmed, M. A. Alipour, C. Jensen, and A. Groce, "Mutation reduction strategies considered harmful," *IEEE Transactions on Reliability*, vol. 66, pp. 854 – 874, 2017.

[32] R. Just, B. Kurtz, and P. Ammann, "Inferring mutant utility from program context," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2017, pp. 284–294.

[33] R. Just, "The Major mutation framework: Efficient and scalable mutation analysis for Java," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 433–436.

[34] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "PIT: a practical mutation testing tool for Java," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 449–452.

[35] J. Offutt, Y.-S. Ma, and Y.-R. Kwon, "The class-level mutants of MuJava," in *Proceedings of the 2006 International Workshop on Automation of Software Test*, 2006, pp. 78–84.