Information and Software Technology 132 (2021) 106496

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Identifying method-level mutation subsumption relations using Z3[☆]

Rohit Gheyi[a,*], Márcio Ribeiro[b], Beatriz Souza[a], Marcio Guimarães[b], Leo Fernandes[c], Marcelo d'Amorim[d], Vander Alves[e], Leopoldo Teixeira[d], Baldoino Fonseca[b]

[a] *Department of Computing and Systems, UFCG, Campina Grande-PB, Brazil*
[b] *Computing Institute, UFAL, Maceió-AL, Brazil*
[c] *IFAL, Maceió-AL, Brazil*
[d] *Informatics Center, Universidade Federal de Pernambuco, Recife-PE, Brazil*
[e] *Computer Science Department, UnB, Brasília-DF, Brazil*

ARTICLE INFO

Keywords:
Mutation analysis
Redundant mutants
Theorem proving

ABSTRACT

Context: Mutation analysis is a popular but costly approach to assess the quality of test suites. One recent promising direction in reducing costs of mutation analysis is to identify redundant mutations, i.e., mutations that are subsumed by some other mutations. A previous approach found redundant mutants manually through truth tables but it cannot be applied to all mutations. Another work derives them using automatic test suite generators but it is a time consuming task to generate mutants and tests, and to execute tests.
Objective: This article proposes an approach to discover redundant mutants by proving subsumption relations among method-level mutation operators using weak mutation testing.
Method: We conceive and encode a theory of subsumption relations in the Z3 theorem prover for 37 mutation targets (mutations of an expression or statement).
Results: We automatically identify and prove a number of subsumption relations using Z3, and reduce the number of mutations in a number of mutation targets. To evaluate our approach, we modified MuJava to include the results of 24 mutation targets and evaluate our approach in 125 classes of 5 large open source popular projects used in prior work. Our approach correctly discards mutations in 75.93% of the cases, and reduces the number of mutations by 71.38%.
Conclusions: Our approach offers a good balance between the effort required to derive subsumption relations and the effectiveness for the targets considered in our evaluation in the context of strong mutation testing.

1. Introduction

Mutation analysis is a popular technique to assess quality of test suites [1–3]. The technique introduces variations in code and checks if those variations are observable through test execution. Applying a mutation to a program yields a mutant. A mutant is said to be killed if a test case in the test suite fails on a given mutant; a mutant is said to survive otherwise. The intuition is that a test suite that kills more mutants is more adequate to detect defects when they actually occur [4].

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 are redundant, that is, they may not be necessary for the effectiveness of mutation analysis and thus we may discard them [5]. We can speed up execution time using multi-execution, parallel execution, and so on. But reducing cost is still important. Redundant mutants do not contribute to the test assessment process because they are killed when other mutants are also killed [5,6]. Redundant mutants are always subsumed by other mutants. The generation of these mutants increases the total cost and does not help to improve effectiveness of the test suite. Ammann et al. [7] empirically identified that a number of the generated mutants are redundant. Also, Papadakis et al. [8] identified that such redundant mutants inflate the mutation score and that a number of recent research papers are vulnerable to threats to validity due to the effect of these mutants.

☆ This work was partially supported by CNPq and CAPES grants.
* Corresponding author.
E-mail addresses: rohit@dsc.ufcg.edu.br (R. Gheyi), marcio@ic.ufal.br (M. Ribeiro), beatriz.souza@ccc.ufcg.edu.br (B. Souza), masg@ic.ufal.br (M. Guimarães), leonardo.oliveira@ifal.edu.br (L. Fernandes), damorim@cin.ufpe.br (M. d'Amorim), valves@unb.br (V. Alves), lmt@cin.ufpe.br (L. Teixeira), baldoino@ic.ufal.br (B. Fonseca).

https://doi.org/10.1016/j.infsof.2020.106496
Received 16 April 2020; Received in revised form 11 November 2020; Accepted 18 November 2020
Available online 4 December 2020
0950-5849/© 2020 Elsevier B.V. All rights reserved.

To identify redundant mutants, we can take subsumption relations into account. Kaminski et al. [9] 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. [9] identified that only three mutations are sufficient to cover all input domains, yielding a reduction of 57% of redundant mutants. Just et al. [10] expanded this idea with two more mutation operators. Both works use truth tables to infer logical relationships across the operations. Although the idea is promising, we cannot apply it for non-logical operators. For instance, a binary expression with two numeric variables `a + b` has a very large set of input possibilities, which turns the manual and logical approach more difficult. Guimarães et al. [11] proposed an approach to yield dynamic subsumption relations among method-level mutants by using automatic test suite generators, such as Randoop [12] and EvoSuite [13] in the context of strong mutation testing. However, the approach is time consuming since it needs to generate mutants, compile them, generate test suites, and execute them.

In this article, we propose an approach consisting of six steps to discover subsumption relations among method-level mutations using theorem proving in the context of weak mutation testing [14]. We encode a theory of subsumption relations in Z3 and use its theorem prover [15] to automatically identify redundant mutations (Section 4). We consider most of the method-level mutation operators available in the MuJava tool [16,17]. We reduce the number of mutations in a number of mutation targets (mutations of an expression or statement). 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 [11].

To evaluate our approach, we modify MuJava to include the results of 24 mutation targets and evaluate our approach in 125 classes of 5 real projects. Our approach achieves an effectiveness (the percentage of mutants correctly discarded by our technique) of 75.93% and a reduction rate (the percentage of mutants discarded by our technique) of 71.38%. We achieve a good cost-benefit ratio between the effort required to derive the mutation subsumption relations and the effectiveness for the targets considered in our evaluation in the context of strong mutation testing. Moreover, we show that the random sampling strategy requires a sampling rate greater than 60% to achieve a similar effectiveness of our approach. So, our reduction mutation strategy is not considered harmful.

We organize this article as follows. We explain mutant subsumption relations in Section 2, and present a motivating example in Section 3. Section 4 describes our approach to identify subsumption relations using Z3. Section 5 presents the evaluation of our approach. Finally, we relate our approach to others (Section 6), and present concluding remarks (Section 7).

2. Mutation subsumption relations

Mutation analysis uses mutation operators to introduce faults in the program to create mutants deliberately [1]. In this context, there is a wide variety of mutation operators. Each mutation operator can implement a set of mutations. In this work, we follow the same definition for "mutation" of previous work [18]: a *mutation* refers to a syntactic change (e.g., `a&&b` $\mapsto$ `a || b`).

Subsumption relations identify redundancy in sets of mutations and hence can be used to optimize approaches to both mutant and test generation [19]. The subsumed mutants do not need to be generated, and test generation methods can target subsuming mutants.

We now define the `kills` relation.

Definition 1. Consider a program `p`. We apply a mutation `M` to `p` and yield one or more mutants. Let `m` be one of them. Both `p` and `m` always terminate when running any test case. We define the `kills(p,m)` function that yields all test cases that have different return values in `p` and `m`.

For example, consider the `p = x+y` program. Suppose we apply a mutation `M` converting the arithmetic operator `+` to the arithmetic operator `−`. We yield the `m = x−y` mutant. In this example, `kills(p,m)` yields a non empty set of test cases. A test case assigns values to all variables in `p`. It contains the following test case `t=(x=1,y=1)` that yields different values in `p` (2) and `m` (0).

We define the subsumption relation in Definition 2.

Definition 2. Consider a program `p` and two distinct mutations, M_1 and M_2, that are applied on the same mutation target. We say that M_2 subsumes M_1 iff for all targets `tgt` in `p` and all mutants m_1 and m_2 generated from M_1 and M_2, respectively, on `tgt`:

1. $\texttt{kills}(p,m_2) \neq \emptyset$
2. $\texttt{kills}(p,m_2) \subseteq \texttt{kills}(p,m_1)$

The first condition of Definition 2 guarantees that m_2 is not an equivalent mutant [20]. The program and the mutant have at least one test case that yields different values. In the second condition of Definition 2, the set of test cases that kills m_2 is a subset of the set of test cases that kill m_1. Notice that we can have more test cases that kill m_1 but cannot kill m_2. In this way, it is easier to kill m_1 than m_2. So, we say that m_2 subsumes m_1. We do not need to generate m_1 during mutation testing. Studying mutation subsumption relation can help us build more efficient mutation testing tools, significantly improving the applicability of mutation testing in industry by helping to minimize one of the challenges [21].

3. Motivating example

Consider 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 [11]:

1. `a > b` $\mapsto$ `a == b`;
2. `a > b` $\mapsto$ `a != b`;
3. `a > b` $\mapsto$ `a >= b`;
4. `a > b` $\mapsto$ `a < b`;
5. `a > b` $\mapsto$ `a <= b`;
6. `a > b` $\mapsto$ `true`;
7. `a > b` $\mapsto$ `false`.

However, some mutations may not be necessary for the effectiveness of mutation analysis and are actually useless. An equivalent mutant is syntactically different from the original program but has the same semantics [20]. In this work, we focus on redundant mutants. To identify them, we rely on subsumption relations, as defined in Section 2.

For instance, consider the binary expression `a > b` and two mutants: `a >= b` and `a <= b`. Notice that both mutants are not equivalent to the original binary expression using weak mutation testing. If `a` is different from `b` in a test case, we kill `a <= b` but we cannot kill `a >= b`. If `a` is equal to `b` in a test case, we kill both mutants. Since (i) all test cases that kill `a >= b` also kill `a <= b`, and (ii) there are some test cases that kill `a <= b` but cannot kill `a >= b`, we conclude that ROR (>=) subsumes ROR (<=) for the mutation target `a > b`. As a consequence, we must not apply ROR (<=) in this mutation target if we apply ROR (>=) using weak mutation testing, hence reducing the number of redundant mutants.

Previous works manually found redundant mutants through the truth table [9,10]. Although the idea is promising, it can only be applied for logical and relational operators. Guimarães et al. [11] used automatic test generation to identify subsumption relations using

Table 1
It presents the mutation targets, method-level mutations that each operator is able to create in the corresponding target, a minimal set of mutations for each target identified in our approach, and the size of a minimal set of mutations compared to the original one. OP_1: select CDL, ODL, or VDL. We use the following variables. `exp`: unary expression, such as identifiers, variables, literals; `lexp` and `rexp`: unary expressions, or binary expression; `lhs`: identifiers, or variables used in statements; `rhs`: unary expressions, or binary expression used in statements.

Mutation target	Mutation operators	Minimal set of mutations	Size
lexp + rexp (for Z^+)	AORB (2), VDL (2), CDL (2), ODL (2)	AORB(*)	12.5%
lexp + rexp (for Z)	AORB (2), VDL (2), CDL (2), ODL (2)	All	100%
lexp - rexp (for Z^+)	AORB (2), VDL (2), CDL (2), ODL (2)	OP_1(lexp)	12.5%
lexp - rexp (for Z)	AORB (2), VDL (2), CDL (2), ODL (2)	All	100%
lexp * rexp (for Z^+)	AORB (2), VDL (2), CDL (2), ODL (2)	AORB(+), OP_1(lexp), OP_1(rexp)	87.5%
lexp * rexp (for Z)	AORB (2), VDL (2), CDL (2), ODL (2)	All	100%
lexp ^ rexp (bool)	COR (4), ROR(2), COI (3), VDL (2), CDL (2), ODL (2)	COR(False), COR(||)	13.3%
lexp && rexp	COR (4), ROR(2), COI (3), VDL (2), CDL (2), ODL (2)	OP_1(lexp), OP_1(rexp), ROR(==), COR(False)	26.7%
lexp || rexp	COR (4), ROR(2), COI (3), VDL (2), CDL (2), ODL (2)	OP_1(lexp), OP_1(rexp), ROR(!=), COR(True)	26.7%
lexp == rexp (bool)	ROR (1), COI (3), VDL (2), CDL (2), ODL (2)	OP_1(lexp), OP_1(rexp)	20%
lexp != rexp (bool)	ROR (1), COI (3), VDL (2), CDL (2), ODL (2)	OP_1(lexp), OP_1(rexp)	20%
lexp == rexp	ROR (7), COI (1)	ROR(False), ROR(>=), ROR(<=)	37.5%
lexp != rexp	ROR (7), COI (1)	ROR(<), ROR(True), ROR(>)	37.5%
lexp > rexp	ROR (7), COI (1)	ROR(False), ROR(!=), ROR(>=)	37.5%
lexp >= rexp	ROR (7), COI (1)	ROR(True), ROR(==), ROR(>)	37.5%
lexp < rexp	ROR (7), COI (1)	ROR(False), ROR(!=), ROR(<=)	37.5%
lexp <= rexp	ROR (7), COI (1)	ROR(True), ROR(==), ROR(<)	37.5%
lexp != rexp (obj)	ROR (7), COI (1)	ROR(True), ROR(>), ROR(<)	37.5%
lexp & rexp	LOR (2), VDL (2), CDL (2), ODL (2)	OP_1(lexp), OP_1(rexp)	25%
lexp | rexp	LOR (2), VDL (2), CDL (2), ODL (2)	OP_1(lexp), OP_1(rexp), LOR(^)	37.5%
lexp ^ rexp	LOR (2), SOR (2), CDL (2), ODL (2)	LOR(|)	12.5%
lexp >> rexp	LOR (3), SOR (1), VDL (2), CDL (2), ODL (2)	OP_1(lexp), OP_1(rexp), LOR(^), LOR(|), LOR(&), SOR(<<)	60%
lexp << rexp	LOR (3), SOR (1), VDL (2), CDL (2), ODL (2)	LOR(^), LOR(&), SOR(≫)	30%
exp	AOIS (4), AOIU (1), LOI (1)	AOIU(-exp)	16.7%
+exp	AODU (1), LOI (1), ODL (1)	LOI(~exp)	33.3%
-exp	AODU (1), LOI (1), ODL (1)	AODU(exp)	33.3%
++exp	AORS (1), AODS (1), LOI (1), ODL (1)	AODS(exp), LOI(~exp)	50%
exp++	AORS (1), AODS (1), LOI (1), ODL (1)	LOI(~exp)	25%
--exp	AORS (1), AODS (1), LOI (1), ODL (1)	AODS(exp), LOI(~exp)	50%
exp--	AORS (1), AODS (1), LOI (1), ODL (1)	LOI(~exp)	25%
!exp	COD (1), ODL (1)	COD(exp)	50%
~exp	AODU (1), LOD (1), ODL (1)	LOD(exp)	33.3%
lhs += rhs (for Z^+)	ASRS (2), ODL (1), SDL (1)	ASRS(*=)	25%
lhs -= rhs (for Z^+)	ASRS (2), ODL (1), SDL (1)	ODL(lhs=rhs)	25%
lhs *= rhs (for Z^+)	ASRS (2), ODL (1), SDL (1)	ODL(lhs=rhs), ASRS(+=), SDL	75%
lhs <<= rhs	ASRS (1), ODL (1), SDL (1)	ASRS(≫=)	33.3%
lhs >>= rhs	ASRS (1), ODL (1), SDL (1)	All	100%
lhs &= rhs	ASRS (2), ODL (1), SDL (1)	ODL(lhs=rhs), SDL	50%
lhs | = rhs	ASRS (2), ODL (1), SDL (1)	ODL(lhs=rhs), ASRS(^=), SDL	75%
lhs ^= rhs	ASRS (2), ODL (1), SDL (1)	ASRS(|=)	33.3%

strong mutation testing. However, we cannot have full confidence in the results derived using automatic test suite generators for some types of variables, such as integer numbers. Moreover, it is a time consuming approach to derive the subsumption relations using automatic test suite generators. In this work, we focus on proposing an approach to automatically derive sound method-level mutation subsumption relations in a theorem prover using weak mutation testing.

4. Encoding and proving subsumption relations

In this section, we propose a technique to prove subsumption relations using weak mutation testing. We focus on code fragments. We use the Z3 [15] API for Python, which has a theorem prover. We consider most MuJava method-level mutation operators [17], such as operators that mutate arithmetic, relational, and logical expressions, and variable assignment statements. We do not focus on the object-oriented ones, i.e., the class-level mutation operators.

Table 1 illustrates a number of method-level mutation targets (code fragments) in which MuJava is able to apply a set of mutations from one or more mutation operators. Consider the first column of the table. The first row of the table focuses on the mutation target `lexp + rexp (for` Z^+`)`, where `lexp` and `rexp` are positive integer expressions. In the second row of Table 1, `lexp` and `rexp` are integer expressions. For other mutation targets, we also consider boolean expressions or objects in other targets. In the second column of the table, we present the mutation operators that can be applied to each mutation target. We can apply four mutation operators in MuJava to the mutation target presented in the first row of the table: AORB, VDL, CDL, and ODL. Table 2 describes the mutation operators considered in our work [22].

The mutation operators can generate eight mutants, two for each operation. We provide the number of possible mutations (in parentheses) that such operator can apply into the target. So, we have in the second column of Table 1: AORB (2), VDL (2), CDL (2), ODL (2). By using our technique explained in Section 4.2, we yield a minimal set of mutations presented in the third column of Table 1. For the mutation target `lexp + rexp` (for Z^+), we only have one mutation: AORB using the operator `*`. The other seven mutants presented in the second column of Table 1 are redundant or equivalent. Since the redundant mutants do not contribute to the test assessment process because they are killed when other mutants are also killed [5,6], our technique detects and removes them. So, we define a minimal set of mutations. We may have more than one minimal sets, but all of them have the same set cardinality. A developer can select any of them. Finally, the last column of Table 1 indicates the size of a minimal set of mutations compared to the original one. Since our minimal set for this mutation target contains one out of eight mutations, the size is 12.5%.

This section is organized as follows. Section 4.1 presents some auxiliary functions. Section 4.2 defines the main steps of our technique. Section 4.3 encodes our technique in Z3. Finally, we present our lessons learned in Section 4.4.

4.1. Auxiliary functions

Listing 1 specifies how to prove a theorem using the Z3 Python API. It can yield three answers: the theorem is valid, invalid, or it does not know the answer. The command `Solver` creates a general purpose solver in Z3 [15]. Constraints can be added using the `add` function. The `Solver.check` method solves the constraints. The result is `sat` (satisfiable) if a solution was found. The result is `unsat` (unsatisfiable) if no solution exists. Finally, a solver may fail to solve a system of constraints and `unknown` is returned.

Listing 1: Proving a theorem in Z3.

```
def prove(theorem):
  s = Solver()
  s.add(Not(theorem))
  r = s.check()
  if r == unsat:
    return 1 # theorem is valid
  elif r == unknown:
    return 2 # Z3 doesn't know the answer
  else:
    return 0 # theorem is invalid
```

Listing 2 presents two functions checking whether constraints are satisfiable (`isSat`) or unsatisfiable (`isUnsat`).

Listing 2: Checking constraints in Z3.

```
def check(f):
  s = Solver()
  s.add(f)
  r = s.check()
  if r == unknown:
    print(``unexpected unknown result for '', f)
  return r

def isSat(f):
  return check(f) == sat

def isUnsat(f):
  return check(f) == unsat
```

Before specifying the subsumption relation, we encode the `kills` function in Listing 3. It defines a formula stating that `p` and `m` have different values.

Listing 3: The function kills.

```
def kills(p, m):
  return p != m
```

The `subsumption` function presented in Listing 4 checks whether a mutation subsumes another one, when considering the input program `p`. We may add some conditions (`conds`) when checking a theorem, such as restricting that all integer numbers are positive. The first part of Definition 2 states that kills(`p`,`m`$_2$) $\neq \emptyset$. To encode it in Z3, we define the `isNonEmpty` function, which tries to find a test case for `kills(p, m)`. To check the second condition presented in Section 2, we define the `isSubset` function, which checks whether there is no test case that is valid for `kills(p, m2)` but it is not valid for `kills(p, m1)`.

Listing 4: Defining a theorem in Z3.

```
def isNonEmpty(p, cond, m):
  return isSat(And(cond, kills(p, m)))

def isSubset(p, cond, m1, m2):
  return isUnsat(And(cond, kills(p, m2),
    Not(kills(p, m1))))

def subsumption(p,m1,m2,conds):
  t1 = isNonEmpty(p, conds, m2)
  t2 = isSubset(p, conds, m1, m2)
  if t1 == 1 and t2 == 1:
    return (m2,m1) # m2 subsumes m1
  else:
    return None
```

To make it easier to compare all mutations, we define the `identifySubsumptions` function (see Listing 5) that compares all possible combinations to identify whether a mutation subsumes another one. `muts` represents a list of mutants.

Listing 5: Identifying all subsumption relations in Z3.

```
def identifySubsumptions(p, muts, conds):
  result = []
  for i in range(len(muts)):
    for j in range(len(muts)):
      if i != j:
        s = subsumption(p,muts[i], muts[j], conds)
        if (s is not None):
          result.append(s)
  return result
```

Moreover, we also declare the `keepNonEquivalentMutants` function that keeps only non-equivalent mutants (see Listing 6). In this way, we discard equivalent mutants from our analysis, hence satisfying the first condition of Definition 2.

Listing 6: Identifying equivalent mutants in Z3.

```
def keepNonEquivalentMutants(p, muts):
  return [m for m in muts if prove(p==m)!=1]
```

Finally, we define the `keepNonRedundantMutants` function that keeps only non redundant mutants (see Listing 7). A mutant is duplicated to another mutant when both of them have the same semantics. In this way, we discard redundant mutants.

Table 2
Description of mutation operators.

Operator	Description
AORB	Binary Arithmetic Operator Replacement
AORS	Short-Cut Arithmetic Operator Replacement
AOIU	Unary Arithmetic Operator Insertion
AOIS	Short-Cut Arithmetic Operator Insertion
AODU	Unary Arithmetic Operator Deletion
AODS	Short-Cut Arithmetic Operator Deletion
ROR	Relational Operator Replacement
COR	Conditional Operator Replacement
COI	Conditional Operator Insertion
COD	Conditional Operator Deletion
SOR	Shift Operator Replacement
LOR	Logical Operator Replacement
LOI	Logical Operator Insertion
LOD	Logical Operator Delete
ASRS	Short-Cut Assignment Operator Replacement
SDL	Statement DeLetion
VDL	Variable DeLetion
CDL	Constant DeLetion
ODL	Operator DeLetion

Listing 7: Detecting redundant mutants in Z3.

```
def redundantMutants(m1,m2):
  if prove(m1==m2) == 1:
    return True
  else:
    return False

def keepNonRedundantMutants(muts):
  non_redundant = []
  for m1 in muts:
    if not any(redundantMutants(m1, m2)
     for m2 in non_redundant):
      non_redundant.append(m1)
  return non_redundant
```

4.2. Steps

For each mutation target, the main steps of our approach are the following:

1. Declare variables and conditions;
2. Specify a program;
3. Specify a list of mutants;
4. Identify and remove equivalent mutants;
5. Identify and remove redundant mutants;
6. Identify subsumption relations.

Only Steps 4–6 do not change for all mutation targets.

4.3. Encoding

Next we follow the steps presented in Section 4.2, use the auxiliary functions presented in Section 4.1, and identify some subsumption relations for some mutation targets presented in Table 1 using weak mutation testing.

4.3.1. Boolean expressions

Next we prove subsumption relations for boolean expressions. For a boolean expression `lexp && rexp` (the eighth row of Table 1), we simplify it to `x && y`. We declare `x` and `y` as boolean variables in the Z3 Python API (Step 1) as shown in Listing 8. We can declare other types of variables in Z3 [15]: `Int` (integer numbers), `Bool` (boolean variables), `BitVec` (bit-vector variables), `Real` (real numbers), and so on. In our work, we use `Bool`, `Int`, and `BitVec` with 32 bits. For the `x && y` target, we do not impose any condition (see first column of Table 1). So, we declare `conds=True` in our example.

In Step 2, we specify our program. In Z3, we have the following boolean operators: `And`, `Or`, `Not`, `Implies` (implication), `If`, and so on. In our example, we use the declared variables and specify our program in Z3: `And(x,y)` (see Listing 8).

After declaring variables and a program, we specify all mutants in Step 3. According to Table 1, the binary expression representing the input program `x and y` can derive the following mutants using ODL, VDL, CDL, COR, and COI operators:

- `True` (COR `true`);
- `False` (COR `false`);
- `x or y` (COR ||);
- `x` (`VDL/CDL/ODL(rexp)`);
- `y` (`VDL/CDL/ODL(lexp)`);
- `not(x and y)` (COI `!()`);
- `x == y` (COR `==`);
- `x != y` (COR `!=`);
- `x xor y` (COR `^`);
- `not(x) xor y` (COR COI `(!x && y)`);
- `x xor not(y)` (COR COI `(x && !y)`).

The last two of them are higher-order mutants derived from COR and COI mutant operators [23]. We consider them to show how to encode them in our approach. Next we manually specify them using the Z3 boolean operators (see Listing 8), but this process can be automated.

Listing 8: Identify Subsumption Relations for `lexp && rexp` target.

```
# Step 1
x = Bool('x')
y = Bool('y')
conds = True
# Step 2
p = And(x,y)
# Step 3
muts = [True, False, Or(x,y), x, y,
        Not(p), equals(x,y),
        Not(equals(x,y)), xor(x,y),
        xor(Not(x),y), xor(x,Not(y))]
# Step 4
muts = keepNonEquivalentMutants(p,muts)
# Step 5
muts = keepNonRedundantMutants(muts)
# Step 6
subsumptions = identifySubsumptions(p, muts, conds)
```

Next we identify non-equivalent mutants in some targets using the `keepNonEquivalentMutants` function (Step 4). For instance, consider the `exp` mutation target. Some mutants (`exp++`, and `exp--`) are equivalent to the program `exp` in our encoding using weak mutation testing.

We can further reduce the number of mutations by checking whether there are some mutants that are redundant to other ones in Step 5. We can check this by calling the `keepNonRedundantMutants` function passing the set of mutants yielded in Step 4. For the `lexp && rexp`, all four dominant nodes are not redundant. We find some redundant mutants for other targets, such as `--exp` target. Consider the following set of mutations: `AODS(exp)`, `AORS(exp++)`, and `ODL(exp)`. The three mutants are redundant. Since they are redundant, for the `--exp` target, we can select one of them (`AODS(exp)`, `AORS(exp++)`, and `ODL(exp)`), instead of selecting all of them.

Finally, to identify all subsumption relations in Step 6, we have to call the `identifySubsumptions` function passing `p`, `muts`, and

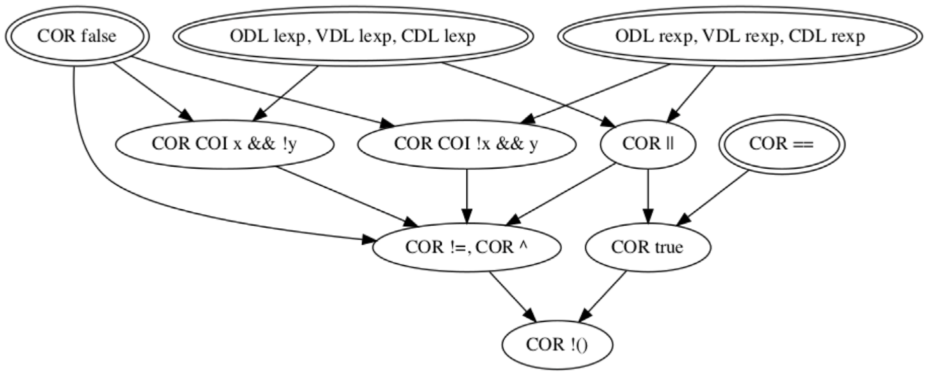

Fig. 1. Mutation subsumption graph for the `lexp && rexp` mutation target. Mutations CDL/VDL/ODL(lexp), CDL/VDL/ODL(rexp), COR ==, and COR false dominate the other mutations.

`conds` as parameters. Based on the output, our script automatically derives the following subsumption graph presented in Fig. 1 for the mutation target `lexp && rexp`. We create a node for each mutation, and an arrow between two nodes, when a mutation subsumes another one. For example, since `COR ||` subsumes `COR true`, we specify this subsumption relation by including an arrow between the nodes. For the `lexp && rexp` mutation target, our results indicate that we only need to use the following ones: `ODL lexp`, `VDL lexp`, `CDL lexp`, `ODL rexp`, `VDL rexp`, `CDL rexp`, `COR ==`, and `COR false`. These nodes dominate the others since they do not have incoming arrows. It is important to mention that `ODL exp`, and `VDL exp` or `CDL exp` yield syntactic equivalent mutants when we are dealing with variables or constants. We only need to select one of them. So, we only need to use four mutations for the following target `x and y`.

We also automated the process of creating the graph presented in Fig. 1 by using the graphviz Python library. Listing 9 declares the `createSubsRelationGraph` function that receives the subsumptions relations identified by `identifySubsumptions`, a list of mutations and a dictionary specifying the names for each mutation.

Listing 9: Creating the subsumption relation graph.

```
def createSubRelationGraph(subsumptions,
 muts, mutNames):
  graph = Digraph('G')
  for m in muts:
    graph.node(mutNames[str(m)])
  for s in subsumptions:
    x = mutNames[str(s[0])]
    y = mutNames[str(s[1])]
    graph.edge(x,y)
  return graph
```

It is important to mention that `prove` (see Listing 1) does not yield unknown as a result in any of the results presented in Table 1. But this scenario can happen for other mutation targets. When Z3 yields unknown, we cannot identify subsumed relations. We recommend adding some conditions (`conds`) to the variables to avoid unknown in `prove`. This way, we may identify some useful subsumption relations for a restricted domain. So, we have confidence in the results given by the Z3 theorem prover. It takes a few seconds to prove all relations on a MacBook Pro 2,3 GHz Intel Core i5 with 8 GB RAM memory.

4.3.2. Integer expressions

Consider the `lexp + rexp` target (the second row of Table 1). In Step 1, we declare integer variables `x` and `y` (see Listing 10). First, we will not impose any condition to identify subsumption relations (`conds=True`), since we do not have any constraint to this mutation target. We declare the `x + y` program in Python using its arithmetic operator. Then, we specify the following mutations (`AORB (2)`, `VDL (2)`, `CDL (2)`, `ODL (2)`). Since `VDL`, `CDL`, and `ODL` yield the same result, we only declare one mutation for them. In this example, we have four mutations:

- `x * y` (`AORB(*)`);
- `x - y` (`AORB(-)`);
- `x` (`VDL/CDL/ODL(rexp)`);
- `y` (`VDL/CDL/ODL(lexp)`).

Listing 10: Identify Subsumption Relations for the `lexp + rexp` target.

```
# Step 1
x = Int('x')
y = Int('y')
conds = True
# Step 2
p = x+y
# Step 3
muts = [x*y, x-y, x, y]
# Step 4
muts = keepNonEquivalentMutants(p,muts)
# Step 5
muts = keepNonRedundantMutants(muts)
# Step 6
subsumptions = identifySubsumptions(p, muts, conds)
```

We do not find equivalent and redundant mutants in Steps 4 and 5. Step 6 does not identify any subsumption relation for the `lexp + rexp` target. For instance, for the test case `x=2, y=2`, we can kill the `AORB(-)` mutation, but we cannot kill the `AORB(*)` mutation. On the other hand, for the test case `x=2, y=0`, we can kill the `AORB(*)` mutation, but we cannot kill the `AORB(-)` mutation. So, all mutations are dominant (see Fig. 2) different from the result obtained by Guimarães et al. [11]. All mutation targets containing integer numbers have different subsumption graphs from Guimarães et al. [11].

However, in case there are some restrictions in the developers' domain, we can reduce the number of generated mutants. For instance, suppose that all numbers are positive (for Z^+) in the developers' domain (see first row of Table 1). In Step 1, we can specify it (`conds = And(x>0,y>0)`) using Z3 and Python operators. We can execute all steps again, and now it yields the subsumption relation presented in Fig. 3. In this setting, the `AORB(*)` mutation dominates all other mutations for the `lexp + rexp` mutation target. Even considering this condition, the subsumption graph is different from the result obtained by Guimarães et al. [11], which has a limitation in their technique due to limitations in using automatic test suite generators.

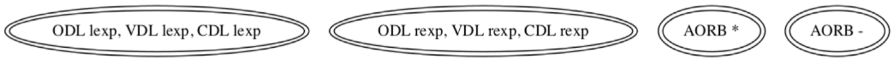

Fig. 2. Mutation subsumption graph for the `lexp + rexp` mutation target. All mutations are dominant.

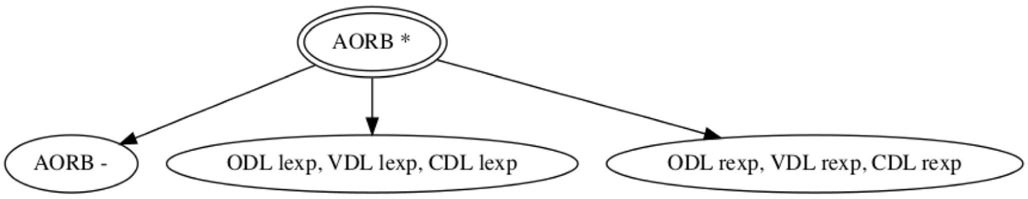

Fig. 3. Mutation subsumption graph for the `lexp + rexp` mutation target considering positive integer numbers (for Z^+).

4.3.3. *Expressions containing bitwise operators*

For expressions using bits, we follow a similar approach. We declare the variables x and y as a BitVec with 32 bits. Then we follow the same steps. It is important to mention that all bitwise operators presented in Table 1 (Mutation target column) have an equivalent bitwise operator in Python.

4.3.4. *Expressions containing assignment operators*

For expressions containing assignment operators, consider the `lhs &= rhs` target (see second to the last row of Table 1). The encoding is equivalent to the one presented in Section 4.3.2. In Step 1, we declare BitVec variables x and y containing 32 bits (see Listing 11). We will not impose any condition to identify subsumption relations (`conds=True`), since we do not have any constraint to this mutation target. The only difference between encoding expressions and commands is that we update the variable `lhs`. Since in our program and in all mutants we only have one variable `lhs` being updated, we do not need to specify it in our encoding. In summary, we encode commands in the same way we encode expressions.

Listing 11: Identify Subsumption Relations for the `lhs &= rhs` target.

```
# Step 1
x = BitVec('x',32)
y = BitVec('y',32)
conds = True
# Step 2
p = x&y
# Step 3
muts = [y,x|y,x^y,x]
# Step 4
muts = keepNonEquivalentMutants(p,muts)
# Step 5
muts = keepNonRedundantMutants(muts)
# Step 6
subsumptions = identifySubsumptions(p, muts, conds)
```

We declare the x &= y program in Python using its bitwise operator. Then, we specify all mutations using the Python bitwise operators:

- y (ODL (lhs=rhs));
- x|y (ASRS(|=));
- x^y, (ASRS(^=));
- x (SDL).

For the Statement Deletion mutation operator (SDL), it always yields x. Finally, we execute Steps 4–6 and it yields the subsumption relation graph presented in Fig. 4.

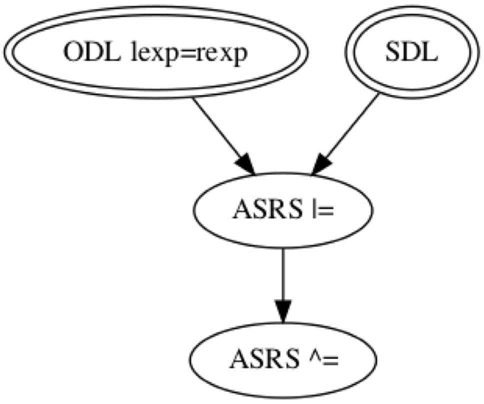

Fig. 4. Mutation subsumption graph for the `lhs &= rhs` mutation target.

4.4. *Lessons learned*

Guimarães et al. [11] used automatic test generators to derive subsumption relations using strong mutation testing. In this work, we use theorem proving in the context of weak mutation testing. Since all proofs are automatically done by the Z3 theorem prover, it is easier and faster to derive the subsumption relations using the approach presented here. In the approach proposed by Guimarães et al. [11], we have to generate a number of mutants using MuJava, compile all of them, generate a number of tests for them, and then analyze all results. It is a time consuming activity. It takes hours to yield dynamic subsumption relations. Moreover, we have to rely on good automatic test suite generators. However, tests only improve confidence in the previous results since we do not have a proof [24].

In the approach presented here, we only need to encode the program and mutants (see Listing 8) to prove subsumption relations in few seconds. Analyzing the values given by the Z3 theorem prover for invalid theorems can help in this process to better understand why a mutation does not subsume another one.

By using our approach, we find some differences in the dynamic mutant subsumption graphs derived by Guimarães et al. [11] that contain integer expressions. All mutation subsumption relation graphs are different. We find that we cannot reduce the number of mutations for targets containing integer numbers. Since Guimarães et al. [11] rely on the test suite generators that do not consider all integer values, they find some subsumption relations different from our work.

All mutant subsumption relation graphs, proof scripts, and reproducibility instructions can be found in our notebook [25].

5. Evaluation

In our previous section, we analyze code fragments using weak mutation testing to derive a minimal set of mutants (see Table 1).

This section evaluates our subsumption relations in the context of strong mutation testing by considering a complete program. Offutt and Lee [26] evaluated the effectiveness versus the efficiency of weak mutation testing. They found that weak mutation testing can be applied in a manner that is almost as effective as strong mutation testing and with significant computational savings. However the results using weak mutation testing do not always hold for strong mutation testing. For instance, Lindström and Márki [24] found that their subsumption relations for ROR identified using weak mutation testing do not hold for strong mutation testing.

To analyze to what extent our results hold for complete programs, first we change MuJava to include the results presented in Table 1 for 24 mutation targets. This tool is called MuJava-M [11]. Then we compare the results for MuJava and MuJava-M for a number of mutants generated from real projects in this section.

This section is organized as follows. First we present our research questions in Section 5.1. Section 5.2 presents the experimental planning. Section 5.3 explains the experimental procedure. Section 5.4 shows our results. We compare our technique to random sampling in Section 5.5. Finally, we discuss some threats to validity in Section 5.6. All data, setups, scripts, and MuJava-M are available in our companion website [25].

5.1. Research questions

To better structure our evaluation, we rely on the Goal, Question, Metrics methodology [27]. The goal of our experiment consists of analyzing our approach, implemented by MuJava-M, with the purpose of evaluating the subsumption relations we found in Z3 with respect to the number of mutants discarded (effort reduction), and the correctness of this reduction (effectiveness) from the point of view of testers in the context of applying mutation testing to Java open source programs (strong mutation testing).

To achieve this goal, we address the following research questions:

RQ$_1$: *How many mutants are subsumed (effort reduction)?*

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. The subsumption relations we embedded in MuJava-M must be effective. They should not discard important mutants that would be in a minimal set. To better understand this point, we formulate the following complementary research question:

RQ$_2$: *How many mutants are incorrectly discarded from a minimal set (effectiveness)?*

To answer RQ$_2$, we rely on the definition of minimal test set [7]. According to Amman et al. [7], a minimal test set necessary to kill a minimal mutants set must also kill all the mutants in the full mutants set. Thus, we generate this minimal test 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.

5.2. Planning

We use five large open source programs to carry out our evaluation. Table 3 illustrates the studied programs, i.e., *joda-time*, *commons-math*, *commons-lang*, *h2*, and *javassist*. These programs vary in size and application domain. *joda-time* is a time manipulation library. *commons-math* is a library of mathematics and statistics components. *commons-lang* is a package of Java utility classes for the classes that are in java.lang's hierarchy. *h2* is a Java SQL-based database. *javassist* is class library for editing bytecodes. 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 cases, all method-level mutation operators were enabled.

Table 3
Programs 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

After generating mutants with each tool, we need to calculate the incorrectly discarded mutants by MuJava-M. Thus, we need to execute a minimal test set – necessary to kill the MuJava-M mutants – against the mutants generated by MuJava. To find out the minimal test set, we rely on EvoSuite's [13] 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 s 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 try to discard equivalent mutant. Equivalent mutants contribute negatively to the confidence assessment of the reduction applied. Unfortunately, detecting equivalent mutants is a well-known undecidable problem [28]. To minimize this problem, we avoid some equivalent mutants by using the Trivial Compiler Equivalence (*TCE*) [29]. *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 plan 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, we evaluate 125 class files (see Table 4).

5.3. Procedure

We explain how we proceed to answer the research questions RQ$_1$ and RQ$_2$. 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. With the classes selected, we executed MuJava and MuJava-M against these classes to generate the full set and a minimal set of mutants, respectively. We enabled all method-level mutation operators in both tools.

Next, we added the mutants of MuJava and MuJava-M grouped by target. For instance, for each target t in a given class file, MuJava generated the full set $M = \{m_1, m_2, m_3, m_4\}$ containing all mutants, and MuJava-M generated a minimal set $\bar{M} = \{m_1, m_2\}$ containing only the sufficient mutants according to the subsumption relations found previously by our approach (Section 4).

We now proceed to create a minimal test set. As explained, a minimal test set necessary to kill the minimal mutants set must also kill the full mutants set [7]. Thus, we use EvoSuiteR to create a test case for each mutant in a 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 a minimal set of mutants $\bar{M}$.

Table 4
Number of mutants per subject.

Project	Classes	MuJava	MuJava-M
joda-time	38	2755	666
commons-math	34	1282	368
commons-lang	22	1737	537
h2	11	231	63
javassist	20	893	216
Total	125	6898	1850

To validate if the mutants of $\bar{M}$ indeed represent a 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 a 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.

To understand if our approach has eliminated important mutants, we verified the number of mutants not generated by MuJava-M that should be part of a minimal set. We also manually verified a subset of these incorrectly deleted mutants.

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

- If EvosuiteR cannot identify a test case to distinguish the original program and a mutant in a limit of 60 s, 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 [30]. 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.

5.4. Results

Next we answer our research questions.

5.4.1. RQ$_1$: *How many mutants are subsumed (effort reduction)?*

Table 4 presents the number of mutants generated by MuJava and MuJava-M for each subject. In particular, we analyzed 1,403 occurrences of mutation targets in 125 classes. MuJava generated 6,898 mutants, which gives an average of 4.92 mutants per target. MuJava-M, in its turn, generated 1,850 mutants for the same set of mutation targets, i.e., an average of 1.32 mutants per target. This way, MuJava-M achieved an average reduction of 71.38% in the number of generated mutants when compared to the original version of MuJava.

Table 5 illustrates the occurrences of 18 mutation targets we analyzed in the 125 classes. The most common target is `exp`. We identified 1,089 `exp` occurrences. The effort reduction rate is 75.80% on average for this target, respectively.

We achieve significant reductions when considering the total number of generated mutants (see column "Reduction" in Table 5). However, we may have 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$_2$.

Table 5
General results for some targets.

Mutation Target	Occurrences	Projects	Reduction	Effectiveness
`lexp > rexp`	19	4	62.50%	100.00%
`lexp >= rexp`	26	4	62.50%	100.00%
`lexp < rexp`	35	4	62.50%	100.00%
`lexp <= rexp`	16	3	62.20%	100.00%
`lexp == rexp`	34	2	62.50%	100.00%
`lexp != rexp`	14	4	62.50%	100.00%
`lexp && rexp`	13	2	55.56%	100.00%
`lexp || rexp`	25	4	55.56%	100.00%
`lexp & rexp`	33	2	64.32%	100.00%
`lexp | rexp`	6	1	40.00%	100.00%
`lexp ^ rexp`	6	1	83.33%	100.00%
`exp`	1089	5	75.80%	47.20%
`!exp`	27	4	50.00%	100.00%
`-exp`	38	4	58.70%	92.68%
`~exp`	13	2	58.06%	100.00%
`exp++`	4	3	71.43%	100.00%
`exp--`	4	2	83.33%	66.67%
`lhs ^= rhs`	1	1	66.67%	50.00%

5.4.2. RQ$_2$: *How many mutants are incorrectly discarded from a minimal set (effectiveness)?*

Table 5 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 5, we achieve 100% of effectiveness in 14 targets. On the other hand, we achieved only 47.20% of effectiveness for the `exp` target (the one more common in the subjects we studied, i.e., 1,089 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]`.

There are some reasons why the results presented in Table 1 for mutation targets in isolation do not hold for some projects (Table 5). There are scenarios in which we infect the program state but the infection is not propagated [31]. So we cannot kill the mutants. The changes are not observable for users. It is an internal change. This is a limitation of weak mutation testing [14]. However, we also have limitations in the procedure presented in Section 5.3. There are some challenges in using automatic test suite generators [32,33]. These challenges negatively impact on the effectiveness of our approach. For instance, they may not generate some input values to kill some mutants. So, the test suite generator cannot infect the program state [31]. There are some limitations in the automatic test suite generators related to defining oracles [32]. We have scenarios in which the program state is infected, the infection is propagated, but we cannot reveal it since we do not have a good oracle. The automatic test suite generators may generate flaky (unstable) tests [32]. As future work, we intend to manually analyze our sample, and also consider projects with test suites created by developers.

Next, we discuss an error when defining a minimal set for an instance of the target `lhs ^= rhs` [11]. This target occurred only once in the subjects studied (see Table 5). Listing 12 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(|=)` as presented in Table 1. However, the minimal test set did not kill the `ASRS(&=)` mutation.

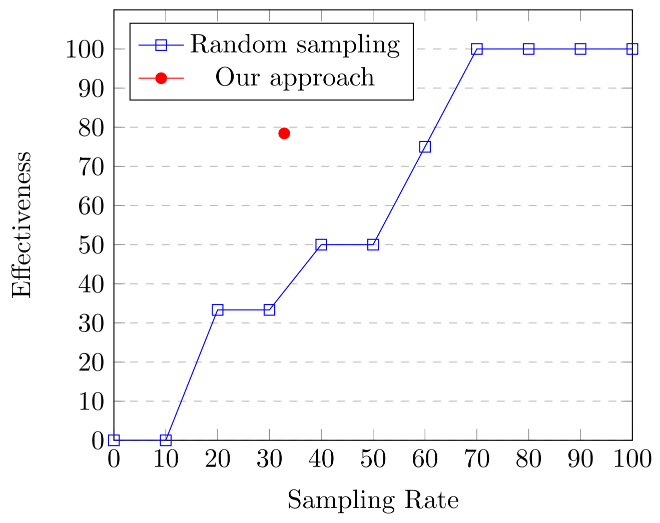

Fig. 5. Comparing our approach to random sampling strategy.

Listing 12: Code snippet from *commons-lang* project.

```
public static boolean xor(final boolean... array) {
  ...
  boolean result = false;
  for (final boolean element : array) {
    result ^= element;
  }
  return result;
}
```

Lindström and Márki [24] suggested that the subsumption relations cannot hold when the mutated statements are re-executed (in the context of strong mutation [1]). If the mutated instruction is executed more than once by any test execution, we cannot determine the future state of the program. Notice that the mutation target is inside the for loop (see Listing 12). Since our subsumption relations were obtained using weak mutation testing, they are not sufficient to represent the mutation within a repeating context.

In summary, we show that our approach to identify subsumption relations in Z3 using weak mutation testing (see Section 4) has a good balance between the effort (sampling rate of 28.62%) required to derive them and the effectiveness (75.93%) for the targets considered in our evaluation in the context of strong mutation testing.

5.5. Random sampling

Gopinath et al. [34] compared the effectiveness of some mutation reduction strategies to random sampling. In their evaluation, none of the mutation reduction strategies evaluated produced an effectiveness advantage larger than 5% in comparison with random sampling. In summary, they argue that mutation reduction strategies are considered harmful. In this section, we compare our approach to random sampling.

We consider the programs presented in Table 3 and targets presented in Table 5 to evaluate the random sampling strategy. The baseline minimal set of mutants is defined by joining a minimal set of mutants identified by our approach, and the set of nonequivalent mutants not killed by our approach. For each target, we randomly select the mutant set and count the number of mutants in the baseline minimal set of mutants. To avoid bias, we repeat this process 100 times and yield the median value.

We use 10 sampling rates from 0 to 100%. Fig. 5 presents our results. The random sampling approach yields an average effectiveness of 75% in correctly identifying the baseline minimal set of mutants when it uses a sampling rate of 60%. Our approach presented in Section 4 yields an effectiveness of 75.93% when it uses a sampling rate of 28.62%. This way, different from the results obtained by Gopinath et al. [34] in their study, using the random sampling strategy is not a good approach compared to ours. Our reduction mutation strategy is not considered harmful.

5.6. 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 for Java. In some cases MuJava generates mutants that do not compile or fails to generate some mutants, representing a threat to internal validity.

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 [7]. Thus, the EvosuiteR can generate the test set which is minimal but not minimum [7].

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 [20].

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

6. Related work

There are some strategies to reduce costs for mutation analysis in the literature [35]. Kaminski et al. [9] defined the mutant subsumption graphs for six targets: `lexp > rexp`, `lexp >= rexp`, `lexp < rexp`, `lexp <= rexp`, `lexp == rexp`, and `lexp != rexp`. We yield the same minimal set for them. Moreover, we encode more targets, presented in Table 1, using the Z3 theorem prover. Using a similar strategy, Just et al. [10] presented 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, in several other cases, analyzing all possible combinations is prohibitive due to the high costs. Our approach encodes a theory in Z3 and uses the Z3 theorem prover to automatically deduce the subsumption relations.

Guimarães et al. [11] proposed an approach to identify subsumption relations using automatic test suite generators in the context of strong mutation testing. In contrast, we propose an approach that is simpler to derive subsumption relations. Indeed, we do not need to generate and compile a number of mutants. We do not need to automatically generate tests, nor execute them. Instead by using our theory, we have to encode the program and mutation operators. Then the Z3 theorem prover automatically proved a number of subsumption relations for weak mutation testing.

Just and Schweiggert [18] presented a study that analyzes the effect of redundant mutants on mutation analysis efficiency, mutation score, and mutation coverage ratio. They 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.

Papadakis and Malevris [8] showed that random selection of subsets containing 10%–60% of the generated mutants reduces the ability to detect failures by 26%–6%, respectively. Offutt et al. [36] presented an empirical approach to define an appropriate set of selective mutation operators. The idea was to randomly select a subset of mutation operators [37,38]. Perez et al. [39] explored Evolutionary Mutation Testing to reduce the number of mutants to be executed. Namin et al. [40] formulated the selective mutation problem as a statistical problem. They applied linear statistical approaches to identify a subset of 28 mutation operators for `C`. Some techniques used clustering algorithms to reduce the number of mutants by selecting only a subset of mutants from each cluster [41,42]. Other strategies [43,44] for reducing costs uses the idea of higher order mutants (mutants with more than one syntactic change), which subsume the behavior of two or more mutants with only one syntactic change, also known as first order mutants. We show how to encode two higher order mutants in our approach.

However, in another study, Gopinath et al. [34] found no differences in effectiveness between selective mutation and random selection. The main challenge in reducing the mutants set is not losing useful information. We show that our approach has a better effectiveness than the random sampling strategy for the same sampling rate. Just et al. [45] stated that existing approaches to selective mutation do not take program context into account, and this is fundamental to avoid losing useful information.

The high cost of mutation testing creates an entry barrier to its use in the software industry, but the effectiveness of mutation testing in assessing the quality of the test suite makes it attractive. Therefore, there is an incentive to carry out cost-saving studies and alternative ways to use mutation, such as the approach used by Google, where only one mutant per target is chosen by a software engineer manually during the code quality inspection [46].

In our work, we propose to use subsumption relationships to reduce costs for mutation testing. Our approach is related to the selective mutation strategy, as we use the subsumption relationships found to select the most representative mutants among all generated mutants. Moreover, we encode a theory of subsumption relations in Z3, and use its theorem prover to identify a number of subsumption relations. We focus on identifying subsumption relations using weak mutation testing. We have to be careful when leveraging our results for strong mutation testing. Lindström and Márki [24] studied the subsumption relations between `ROR` mutants. They showed that `ROR` fault hierarchies identified using weak mutation testing do not hold for strong mutation testing. The problem may be mitigated by avoiding loop structures. We evaluate our approach in the context of strong mutation testing in Section 5, and show that our approach has a good balance between the effort required to derive the mutant subsumption relations and the effectiveness for the targets considered in our evaluation.

Previous approaches focused on proposing approaches to detect equivalent mutants [20,47]. Baldwin and Sayward used compiler optimizations [48] to detect equivalent mutants by checking whether the original program and the optimized program are identical. Kintis et al. [29] proposed the Trivial Compiler Equivalence for C and Java, and mutation tools (Milu and MuJava).

Offut and Pan [49,50] developed a technique to detect equivalent mutants based on mathematical constraints that introduce a set of strategies to formulate the killing conditions of the mutants. If these conditions are not feasible, the mutant is equivalent. Voas and McGraw [31] and Hierons et al. [51] suggested to use program slicing to help with equivalence identification. These approaches suffer from inherent limitations in the scalability of constraint handling and slicing technology. Grun et al. [52] and Shuller and Zeller [53,54] proposed that changes in coverage can be used to detect non-equivalents mutants. Shuler et al. [55] used invariants violation as a way to classify killable mutants. In our approach, we defined the function `keepNonEquivalentMutants` and used the Z3 theorem prover to identify equivalent mutants using weak mutation testing.

7. Conclusion

In this work, we automatically identify and prove a number of subsumption relations for method-level mutations using the Z3 theorem prover. Developers only need to specify the types and mutations in our encoding to identify subsumption relations (see Listing 8). In few seconds, the Z3 theorem prover automatically proves a number of subsumption relations for 37 mutation targets. We reduce the number of mutations in a number of mutation targets containing integer and boolean expressions. We show some examples on how to encode them and identify subsumption relations. We can extend our theory to consider other types of expressions. To evaluate our approach, we extend MuJava with some of our results and evaluate it in 125 classes of 5 real projects. Our tool achieves an effectiveness of 75.93% and a sampling rate of 28.62% in the context of strong mutation testing. Moreover, we show that our approach is better than the random sampling strategy.

The results may help to build better mutation testing tools that will allow to reduce the mutation testing costs. We recommend the community to follow a similar approach presented here before proposing new mutations. We must propose new mutations that subsume the previous ones. In this way, developers can use a minimal set of mutations, hence reducing mutation testing costs. Overall, our work leverages lightweight formal methods to mutant analysis, resulting in effective gains for developers.

As future work, we intend to prove more subsumption relations by considering real numbers, other language constructs and mutations, and encoding more higher order mutants. It will require to encode the Java semantics of some constructions, such as classes and fields. We may encode the Java Featherweight semantics in Z3 [56] or in other systems in which we can interactively perform the proofs, such as PVS [57]. In this case, we may address some proofs that cannot be done (when `prove` yields `unknown`) using the Z3 theorem prover. Finally, the expressions and commands considered in this work for Java have a similar semantics in other languages, such as Python and C#. We intend to check whether the subsumption relations found also hold for other languages in the context.

CRediT authorship contribution statement

Rohit Gheyi: Formal analysis, Methodology, Investigation, Conceptualization, Data curation, Supervision, Writing - original draft. **Márcio Ribeiro:** Methodology, Investigation, Conceptualization, Supervision, Writing - original draft. **Beatriz Souza:** Software, Data curation, Validation, Writing - review & editing. **Marcio Guimarães:** Software, Data curation, Validation, Writing - review & editing. **Leo Fernandes:** Software, Data curation, Validation, Writing - review & editing. **Marcelo d'Amorim:** Methodology, Investigation, Validation, Writing - review & editing. **Vander Alves:** Formal analysis, Investigation, Writing - review & editing. **Leopoldo Teixeira:** Formal analysis, Investigation, Writing - review & editing. **Baldoino Fonseca:** Methodology, Writing - review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful suggestions. This work was partially supported by CNPq (465614/2014-0, 426005/2018-0, 421306/2018-1, 309844/2018-5, 311442/2019-6, 427787/2018-1), CAPES (117875, 175956, and 88887), FACEPE (APQ-0570-1.03/14, PRONEX-APQ-0388-1.03/14, and APQ-0399-1.03/17), and FAPEAL (60030.0000000462/2020) grants.

References

[1] R.A. DeMillo, R.J. Lipton, F.G. Sayward, Hints on test data selection: Help for the practicing programmer, Computer 11 (4) (1978) 34–41.

[2] J. Offutt, A mutation carol: Past, present and future, Inf. Softw. Technol. 53 (10) (2011) 1098–1107.

[3] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y.L. Traon, M. Harman, Chapter six - mutation testing advances: An analysis and survey, Adv. Comput. 112 (2019) 275–378.

[4] R. Just, D. Jalali, L. Inozemtseva, M.D. Ernst, R. Holmes, G. Fraser, Are mutants a valid substitute for real faults in software testing? in: Proceedings of the Foundations of Software Engineering, 2014, pp. 654–665.

[5] M. Papadakis, C. Henard, M. Harman, Y. Jia, Y. Le Traon, Threats to the validity of mutation-based test assessment, in: Proceedings of the International Symposium on Software Testing and Analysis, ACM, 2016, pp. 354–365.

[6] M. Kintis, M. Papadakis, N. Malevris, Evaluating mutation testing alternatives: A collateral experiment, in: Proceddings of the Asia Pacific Software Engineering Conference, IEEE, 2010, pp. 300–309.

[7] 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.

[8] M. Papadakis, 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.

[9] G. Kaminski, P. Ammann, J. Offutt, Better predicate testing, in: Proceedings of the International Workshop on Automation of Software Test, ACM, 2011, pp. 57–63.

[10] R. Just, G.M. Kapfhammer, F. Schweiggert, Do redundant mutants affect the effectiveness and efficiency of mutation analysis? in: Proceedings of the International Conference on Software Testing, Verification and Validation, IEEE, 2012, pp. 720–725.

[11] M. Guimarães, L. Fernandes, M. Ribeiro, M. d'Amorim, R. Gheyi, Optimizing mutation testing by discovering dynamic mutant subsumption relations, in: Proceedings of the International Conference on Software Testing, IEEE, 2020, pp. 98–208.

[12] C. Pacheco, S. Lahiri, M. Ernst, T. Ball, Feedback-directed random test generation, in: Proceedings of the International Conference on Software Engineering, IEEE, 2007, pp. 75–84.

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

[14] W. Howden, Weak mutation testing and completeness of test sets, IEEE Trans. Softw. Eng. 8 (4) (1982) 371–379.

[15] L.M. de Moura, N. Bjørner, Z3: An efficient SMT solver, in: Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, 2008, pp. 337–340.

[16] Y.-S. Ma, J. Offutt, Y.-R. Kwon, MuJava: an automated class mutation system, Softw. Test. Verif. Reliab. 15 (2) (2005) 97–133.

[17] Y.-S. Ma, J. Offutt, Y.-R. Kwon, MuJava: a mutation system for Java, in: Proceedings of the International Conference on Software Engineering, ACM, 2006, pp. 827–830.

[18] R. Just, F. Schweiggert, Higher accuracy and lower run time: efficient mutation analysis using non-redundant mutation operators, Softw. Test. Verif. Reliab. 25 (5–7) (2015) 490–507.

[19] B. Kurtz, P. Ammann, M.E. Delamaro, J. Offutt, L. Deng, Mutant subsumption graphs, in: Proceedings of the International Conference on Software Testing, Verification and Validation Workshops, IEEE, 2014, pp. 176–185.

[20] L. Madeyski, W. Orzeszyna, R. Torkar, M. Jozala, Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation, IEEE Trans. Softw. Eng. 40 (1) (2014) 23–42.

[21] G. Petrovic, M. Ivankovic, B. Kurtz, P. Ammann, R. Just, An industrial application of mutation testing: Lessons, challenges, and research directions, in: Proceedings of the International Conference on Software Testing, Verification and Validation Workshops, 2018, pp. 47–53.

[22] Y.-S. Ma, J. Offutt, Description of MuJava's method-level mutation operators, At https://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf.

[23] Y. Jia, M. Harman, Higher order mutation testing, Inf. Softw. Technol. 51 (10) (2009) 1379–1393.

[24] B. Lindström, A. Márki, On strong mutation and the theory of subsuming logic-based mutants, Softw. Test. Verif. Reliab. 29 (1–2) (2019) e1667.

[25] R. Gheyi, M. Ribeiro, B. Sousa, M. Guimarães, M. d'Amorim, V. Alves, L. Teixeira, B. Fonseca, Identifying method-level mutant subsumption relations using Z3 (artifacts), 2020, At https://github.com/rohitgheyi/ist-subsumption-relation-z3.

[26] A.J. Offutt, S.D. Lee, An empirical evaluation of weak mutation, IEEE Trans. Softw. Eng. 20 (5) (1994) 337–344.

[27] V.R. Basili, G. Caldiera, H.D. Rombach, The Goal Question Metric approach, in: Encyclopedia of Software Engineering, Wiley, 1994, pp. 528–532.

[28] T. Budd, D. Angluin, Two notions of correctness and their relation to testing, Acta Inform. 18 (1) (1982) 31–45.

[29] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y.L. Traon, M. Harman, Detecting trivial mutant equivalences via compiler optimisations, IEEE Trans. Softw. Eng. 44 (4) (2017) 308–333.

[30] Q. Luo, F. Hariri, L. Eloussi, D. Marinov, An empirical analysis of flaky tests, in: Proceedings of the Foundations of Software Engineering, 2014, pp. 643–653.

[31] J. Voas, G. McGraw, Software Fault Injection: Inoculating Programs Against Errors, John Wiley & Sons, Inc., 1997.

[32] S. Shamshiri, R. Just, J.M. Rojas, G. Fraser, P. McMinn, A. Arcuri, Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges, in: Proceedings of the Automated Software Engineering, 2015, pp. 201–211.

[33] G. Soares, R. Gheyi, E. Murphy-Hill, B. Johnson, Comparing approaches to analyze refactoring activity on software repositories, J. Syst. Softw. 86 (4) (2013) 1006–1022.

[34] R. Gopinath, I. Ahmed, M. Alipour, C. Jensen, A. Groce, Mutation reduction strategies considered harmful, IEEE Trans. Reliab. 66 (2017) 854–874.

[35] A. Pizzoleto, F. Ferrari, J. Offutt, L. Fernandes, M. Ribeiro, A systematic literature review of techniques and metrics to reduce the cost of mutation testing, J. Syst. Softw. 157 (2019).

[36] A.J. Offutt, A. Lee, G. Rothermel, R.H. Untch, C. Zapf, An experimental determination of sufficient mutant operators, ACM Trans. Softw. Eng. Methodol. 5 (2) (1996) 99–118.

[37] A. Mathur, Performance, effectiveness, and reliability issues in software testing, in: Proceedings of the Annual International Computer Software and Applications Conference, IEEE, 1991, pp. 604–605.

[38] W. Wong, A. Mathur, Reducing the cost of mutation testing: An empirical study, J. Syst. Softw. 31 (3) (1995) 185–196.

[39] P. Delgado-Pérez, S. Segura, I. Medina-Bulo, Assessment of C++ object-oriented mutation operators: A selective mutation approach, Softw. Test. Verif. Reliab. 27 (4–5) (2017).

[40] A. Namin, J. Andrews, D. Murdoch, Sufficient mutation operators for measuring test effectiveness, in: Proceedings of the International Conference on Software Engineering, ACM, 2008, pp. 351–360.

[41] C. Ji, Z. Chen, B. Xu, Z. Zhao, A novel method of mutation clustering based on domain analysis, in: Proceedings of the International Conference on Software Engineering and Knowledge Engineering, Vol. 9, 2009, pp. 422–425.

[42] S. Hussain, Mutation Clustering (Master's thesis), Kings College London, 2008.

[43] Y. Jia, M. Harman, Constructing subtle faults using higher order mutation testing, in: Proceedings of the International Working Conference on Source Code Analysis and Manipulation, IEEE, 2008, pp. 249–258.

[44] W.B. Langdon, M. Harman, Y. Jia, Multi objective higher order mutation testing with genetic programming, in: Testing: Academic and Industrial Conference-Practice and Research Techniques, IEEE, 2009, pp. 21–29.

[45] R. Just, B. Kurtz, P. Ammann, Inferring mutant utility from program context, in: Proceedings of the International Symposium on Software Testing and Analysis, 2017, pp. 284–294.

[46] G. Petrovic, M. Ivankovic, State of mutation testing at google, in: Proceedings of the International Conference on Software Engineering—Software Engineering in Practice, 2018, pp. 163–171.

[47] Y. Jia, M. Harman, An analysis and survey of the development of mutation testing, IEEE Trans. Softw. Eng. 37 (5) (2011) 649–678.

[48] D. Baldwin, F. Sayward, Heuristics for Determining Equivalence of Program Mutations, Technical Report, DTIC Document, 1979.

[49] J. Offutt, J. Pan, Detecting equivalent mutants and the feasible path problem, in: Proceedings of the Annual Conference on Computer Assurance, 1996, pp. 224–236.

[50] J. Offutt, J. Pan, Automatically detecting equivalent mutants and infeasible paths, Softw. Test. Verif. Reliab. 7 (3) (1997) 165–192.

[51] R. Hierons, M. Harman, S. Danicic, Using program slicing to assist in the detection of equivalent mutants, Softw. Test. Verif. Reliab. 9 (4) (1999) 233–262.

[52] B.J. Grün, D. Schuler, A. Zeller, The impact of equivalent mutants, in: Proceeings of the International Conference on Software Testing, Verification and Validation Workshops, 2009, pp. 192–199.

[53] D. Schuler, A. Zeller, (Un-)covering equivalent mutants, in: Proceedings of the International Conference on Software Testing, Verification and Validation, 2010, pp. 45–54.

[54] D. Schuler, A. Zeller, Covering and uncovering equivalent mutants, Softw. Test. Verif. Reliab. 23 (5) (2013) 353–374.

[55] D. Schuler, V. Dallmeier, A. Zeller, Efficient mutation testing by checking invariant violations, in: Proceedings of the International Symposium on Software Testing and Analysis, 2009, pp. 69–80.

[56] A. Igarashi, B.C. Pierce, P. Wadler, Featherweight java: A minimal core calculus for java and GJ, ACM Trans. Programm. Lang. Syst. 23 (3) (2001) 396–450.

[57] S. Owre, J. Rushby, N. Shankar, D. Stringer-Calvert, PVS: an experience report, in: Applied Formal Methods—FM-Trends 98, in: Lecture Notes in Computer Science, vol. 1641, Springer-Verlag, Germany, 1998, pp. 338–345.