UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
UNIDADE ACADÊMICA DE SISTEMAS E COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

BEATRIZ BEZERRA DE SOUZA

MOST HIGHER ORDER MUTANTS ARE USELESS FOR METHOD-LEVEL MUTATION OPERATORS USING WEAK MUTATION

CAMPINA GRANDE - PB

2020

BEATRIZ BEZERRA DE SOUZA

MOST HIGHER ORDER MUTANTS ARE USELESS FOR METHOD-LEVEL MUTATION OPERATORS USING WEAK MUTATION

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharela em Ciência da Computação.

Orientador: Professor Dr. Rohit Gheyi.

CAMPINA GRANDE - PB

2020

S729m Souza, Beatriz Bezerra de.
Most higher mutants are useless for method-level mutation operators using weak mutation. / Beatriz Bezerra de Souza. – 2020.

9 f.

Orientador: Prof. Dr. Rohit Gheyi.
Trabalho de Conclusão de Curso - Artigo (Curso de Bacharelado em Ciência da Computação) - Universidade Federal de Campina Grande; Centro de Engenharia Elétrica e Informática.

1. Mutation analysis. 2. Method-level mutation operaors. 3. Weak mutation. 4. Redundant mutants. 5. Mutant subsumption relations. 6. Subsumption relations I. Gheyi, Rohit. II. Título.

CDU:004(045)

Elaboração da Ficha Catalográfica:

Johnny Rodrigues Barbosa
Bibliotecário-Documentalista
CRB-15/626

BEATRIZ BEZERRA DE SOUZA

MOST HIGHER ORDER MUTANTS ARE USELESS FOR METHOD-LEVEL MUTATION OPERATORS USING WEAK MUTATION

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Professor Dr. Rohit Gheyi
Orientador – UASC/CEEI/UFCG

Professor Dr. Wilkerson de Lucena Andrade
Examinador – UASC/CEEI/UFCG

Professor Dr. Tiago Lima Massoni
Professor da Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 2020.

CAMPINA GRANDE - PB

Most Higher Order Mutants are Useless for Method-Level Mutation Operators Using Weak Mutation

Beatriz Souza
beatriz.souza@ccc.ufcg.edu.br
Federal University of Campina Grande
Campina Grande, Brazil

Rohit Gheyi
rohit@dsc.ufcg.edu.br
Federal University of Campina Grande
Campina Grande, Brazil

ABSTRACT

Mutation analysis is a popular but costly approach to assess the quality of test suites. One of the attempts to reduce the costs associated to mutation analysis is to identify subsuming higher order mutants (HOMs), i.e., mutants that are harder to kill than the first order mutants (FOMs) from which they are constructed. However, it is not known how many HOMs subsume FOMs. In this paper, we use our previous approach, which discovers redundancy in mutations by proving subsumption relations among method-level mutation operators using weak mutation testing, to encode and prove subsumption relations among FOMs and HOMs. We encode a theory of subsumption relations in the Z3 theorem prover for 27 mutation targets (mutations of an expression or statement). We encode 233 FOMS and 438 HOMs and automatically prove a number of subsumption relations using Z3. Our results indicate that 91% of all mutants could be discarded on average. Moreover, 97.5% of all HOMs could be discarded and HOMs compose only 16.67% of the subsuming mutants sets on average.

ACM Reference Format:
Beatriz Souza and Rohit Gheyi. 2020. Most Higher Order Mutants are Useless for Method-Level Mutation Operators Using Weak Mutation. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Mutation analysis [4, 12] is a popular technique to assess quality of test suites. The technique introduces syntactic changes to the code, creating faulty programs called mutants, and checks if the mutants are detected by the test suite. If at least a test case fails on a given mutant, it is said that the test suite killed the mutant; otherwise, the mutant survived. Test suites that kill more mutants are more adequate to detect real errors [13].

The expensiveness associated to mutation testing is 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 [28]. A redundant mutant does not contribute to the test assessment process because they are killed when other mutants are also killed [18, 28]. 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. [1] empirically identified that almost 99% of the generated mutants are redundant. Also, Papadakis et al. [29] 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. [17] 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% of redundant mutants. Just et al. [14] expanded this idea with two more mutation operators. Both works use truth table 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. [8] proposes an approach to yield dynamic subsumption relations among method-level mutants by using automatic test suite generators, such as Randoop [27] and EvoSuite [5] 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. Our previous work proposes an approach consisting of six steps to discover subsumption relations among method-level mutants using theorem proving in the context of weak mutation testing [6]. We encode a theory of subsumption relations in Z3 and use its theorem prover [2] to automatically identify redundant mutants.

Higher order mutation testing (HOMT) [11], an approach that generates mutants by applying mutation operators more than once, is one of the attempts to reduce the expensiveness associated to mutation testing [23]. However, the costs of creating HOMs are also high, since the large number of possible fault combinations creates a set of candidate combinations that is exponentially large [11]. Moreover, combination of faults that are harder to detect than any of the individual constituent faults are relatively rare [11].

In this paper, we use our previous approach [6] to find subsumption relations among FOMs and HOMs. Our goal is to answer the following research questions:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, July 2017, Washington, DC, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- [RQ1] How many mutants could be discarded when considering both FOMs and HOMs?
- [RQ2] How many HOMs could be discarded?
- [RQ3] Are HOMs harder to kill than FOMs?

We organize this paper as follows. We explain mutant subsumption relations in Section 2. Section 3 describes our approach to identify subsumption relations using Z3 [6]. Section 4 presents our findings. Finally, we relate our approach to others (Section 5), and present concluding remarks (Section 6).

2 MUTANT SUBSUMPTION RELATIONS

Mutation analysis uses mutation operators to introduce faults in the program to create mutants deliberately [4]. 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 [16]: a mutation refers to a syntactic change (e.g., `a && b → 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 → a == b, a > b → a != b, a > b → a >= b, a > b → a < b, a > b → a <= b, a > b → true, and a > b → false`. However, some mutations may not be necessary for the effectiveness of mutation analysis and are actually useless.

Subsumption relationships identify redundancy in sets of mutants 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.

There are three types of subsumption relations: True subsumption, Dynamic subsumption, and Static subsumption [19]. True subsumption assumes full knowledge of the relationships among mutants and though valuable as a concept, is undecidable to compute, either through enumeration or analysis. Dynamic subsumption 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), dynamic subsumption approximates "true" subsumption. Static subsumption is computed based on an analysis of the mutants, either manual or automated. Still, as the analysis tends toward capturing the complete semantics of the artifact under test, (again not possible), static subsumption also approximates "true" subsumption.

In our approach, supposing that `m1` and `m2` belong to a set of mutants `M` of a program `p`, we say that `m2` subsumes `m1` if and only if the following conditions are satisfied:

1. There exists some test case `t` such that `m2` and `p` compute different outcomes on `t` (`t` kills `m2`);
2. For every possible test case `t` for `p`, if `m2` computes a different outcome than `p` on `t` (`t` kills `m2`), then so does `m1` (`t` kills `m1`);
3. There exists some test case `t` such that `m1` and `p` compute different outcomes on `t` (`t` kills `m1`), but `t` does not kill `m2`.

The first condition guarantees that `m2` is not an equivalent mutant. In the second condition, `m2` is killed by at least the same set of test cases that kill `m1`. Notice that we can have more test cases that kill `m1` but cannot kill `m2`. The last condition guarantees that `m2` is harder to kill than `m1`. We consider a stronger notion of subsumption relations than Kurtz et al. [19] by including the last condition.

Studying subsumption relations can help us build more efficient mutation testing tools, significantly improving the practical applicability of mutation in industry.

3 ENCODING AND PROVING SUBSUMPTION RELATIONS

We use the Z3 [2] API for Python, which has a theorem prover, to prove subsumption relations using weak mutation testing. We consider most MuJava method-level mutation operators [26], such as operators that mutate arithmetic, relational, and logical expressions. We do not focus on the object-oriented ones, i.e., the class-level mutation operators.

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

- `exp`: unary expression, such as identifiers, variables, literals;
- `lexp` and `rexp`: unary expressions, or binary expression;
- `rhs`: unary expressions, or binary expression used in statements.

Table 3 illustrates a number of method-level mutation targets in which MuJava is able to apply a set of mutations from one or more mutation operators. Accordingly, for each target, we present the number of possible mutations that can be generated for the target using FOMT and HOMT. For example, 15 FOMs and 42 HOMs can be created from the target `lexp && rexp`. From the mutation target `lexp | rexp`, we can create 10 FOMs and 24 HOMs, and so on.

3.1 Overview of Our Approach

The approach proposed in our previous work consists of six steps [6], which we followed as follow:

1. Declare variables and conditions
   In this step, we reduced expressions to x and y, where the types of x and y depend on the mutation target. E.g. For the mutation target `lexp && rexp`, we declared x and y as boolean variables in the Z3 Python API. We can declare other types of variables in Z3 [2]: Int (integer numbers), BitVec (bit-vector variables) and so on.
2. Specify a program
   To specify our programs, we use the declared variables and available operations. In Z3, we have the following boolean operators: And, Or, Not, Implies (implication), If, and so on.
3. Specify a list of mutants
   Here, we specify the FOMs for each program, as we did in our previous work. Moreover, we also specify the HOMs that we were able to create by combining the FOMs.
4. Identify and remove equivalent mutants
   We identify and remove equivalent mutants by calling the `removeEquivalentMutants` function defined in our previous work.
5. Identify subsumption relations

To identify all subsumption relations, we call the `compareAllMuts` function, defined in our previous work, passing the program and the mutants. Based on the output, the script automatically derives the mutant subsumption graph.

(6) Identify redundant mutants in the minimal set
In this step, we call the `identifyRedundantMutants` function passing all dominant mutants identified in Step 5. We discarded the HOMS that were duplicate to FOMs, since FOMs are easier to create.

3.2 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 declaring variables and conditions. For the integer expression `lexp == rexp`, we simplify it to `x == y` and declare `x` and `y` as integer variables in the Z3 Python API (Step 1) as shown in Listing 1. Then, in Step 2, we specify the program and in Step 3 we create all possible mutations of all possible mutation operators for our target (See Listing 1). Table 1 outlines all possible mutations applied to the `lexp == rexp` mutation target. The first eight mutants are FOMs, whereas the last seven mutants are HOMs derived from COI and ROR mutation operators.

Before identifying subsumption relations, we have to call the `removeEquivalentMutants` function in Step 4. If we find an equivalent mutant, we have to remove it from our analysis. Otherwise an equivalent mutant will dominate all other mutants since it is impossible to kill it. We find equivalent mutants in some targets. 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.

To identify all subsumption relations in Step 5, we have to call the `compareAllMuts` function passing `p` and `muts` as parameters. Based on the output, our script automatically derives the following mutant subsumption graph presented in Figure 1 for the `lexp == rexp` mutation target. We create a node for each mutation, and an arrow between two nodes, when a mutation subsumes another one. For example, since ROR false subsumes ROR >, 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 mutations: ROR <=, ROR >=, ROR false, COI ROR !(x>y), COI ROR !(x<y), COI ROR true. 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.

Finally, we can reduce even more the number of mutations by checking whether there are some dominant mutants that are redundant to other ones in the subsuming mutants set in Step 6. We can check it by calling the `identifyRedundantMutants` function passing all dominant mutants identified in Step 5. For the `lexp == rexp` mutation target, the following mutations are redundants: ROR (false) and COI ROR (true), ROR (lexp >= rexp) and COI ROR !(lexp < rexp), ROR (lexp <= rexp) and COI ROR !(lexp > rexp). Since they are redundant, we can select one of each redundant mutation, instead of selecting all of them. As FOMs are simpler than HOMs, we selected the FOMs instead of the HOMs. Hence, for the `lexp == rexp` mutation target, the subsuming mutants set contains only three mutations: ROR (false), ROR (lexp >= rexp), and ROR (lexp <= rexp), which represents a reduction of 98% of the mutants for this target, as presented in Table 3.

Table 1: Mutations applied to the `lexp == rexp` mutation target.

Operator(s)	Mutation(s)
ROR	`lexp == rexp` ⟹ `lexp != rexp`
ROR	`lexp == rexp` ⟹ `lexp > rexp`
ROR	`lexp == rexp` ⟹ `lexp >= rexp`
ROR	`lexp == rexp` ⟹ `lexp < rexp`
ROR	`lexp == rexp` ⟹ `lexp <= rexp`
ROR	`lexp == rexp` ⟹ `true`
ROR	`lexp == rexp` ⟹ `false`
COI	`lexp == rexp` ⟹ `!(lexp == rexp)`
COI ROR	`lexp == rexp` ⟹ `!(lexp != rexp)`
COI ROR	`lexp == rexp` ⟹ `!(lexp > rexp)`
COI ROR	`lexp == rexp` ⟹ `!(lexp >= rexp)`
COI ROR	`lexp == rexp` ⟹ `!(lexp < rexp)`
COI ROR	`lexp == rexp` ⟹ `!(lexp <= rexp)`
COI ROR	`lexp == rexp` ⟹ `!(true)`
COI ROR	`lexp == rexp` ⟹ `!(false)`

4 RESULTS

In this section we present the answer to each research question in turn, indicating how the results answer each.

All mutant subsumption relation graphs and proof scripts are publicly available [32].

4.1 Answer to RQ1

RQ1 is designed to investigate the quantity of the subsumed mutants in general. To begin the analysis, the fifth column of Table 3 presents the size of the subsuming mutants set found in each mutation target by our approach. On average, the size of the minimal set of mutations for each target is 9%. Which implies that, when applying both FOMT and HOMT, 91% of all mutants could be discarded on average.

4.2 Answer to RQ2

RQ2 is designed to investigate the quantity of the subsumed HOMs. In total, we encoded 438 HOMs. However, only 11 HOMs, which are highlighted in Table 2, were considered subsuming according to our approach. Therefore, 97.5% of all encoded HOMs could be discarded.

Table 2: Subsuming HOMs found by our approach for the mutation targets presented in Table 3.

Mutation Target	Subsuming HOMs
`lexp ^ rexp (bool)`	COI COR(lexp&&!rexp), COI COR(!lexp&&rexp)
`lexp == rexp (bool)`	COI COR(!x||y), COI COR(x||!y), COI COR !(x||y)
`lexp != rexp (bool)`	COI COR(!x&&y), COI COR(x&&!y), COI COR !(x&&y)
`exp`	LOI AOIS ~(++exp)
`++exp`	LOI AODS(~exp)
`--exp`	LOI AORS(~exp)

4.3 Answer to RQ3

RQ3 is designed to investigate whether HOMs are harder to kill than FOMs. We found that HOMs compose just 16.67% of all the mutants present in the subsuming mutants set on average (See the HOMs ⊂ Minimal Set column in Table 3), whereas FOMs compose 83.33% (See the FOMs ⊂ Minimal Set column in Table 3). HOMs are present in the subsuming mutants set of 5 out of the 27 mutation targets, as can be seen in Table 3 and highligheted in Table 2. Only two of the mutation targets have the subsuming mutants set composed uniquely by HOMs: `++exp` and `--exp`. Therefore, to our study, FOMs seen to be harder to kill than FOMs and only 11 HOMs are as hard to kill as FOMs.

Listing 1: Identify Subsumption Relations for `lexp == rexp` target.

```
# Step 1
x = Int('x')
y = Int('y')
conds = True
# Step 2
p = x==y
# Step 3
muts = [True, False, x>=y, x<=y,
        x>y, x<y, Not(programa),
        Not(x>=y), Not(x<=y),
        Not(x>y), Not(x<y),
        Not(x!=y), True, False]
# Step 4
muts = removeEquivalentMutants(p, muts)
# Step 5
compareAllMuts(p, muts, conds)
```

5 RELATED WORK

There are some strategies to reduce costs for mutation analysis in the literature [30]. Kaminski et al. [17] defined a sufficient replacements for ROR mutations. Using a similar strategy, Just et al. [14] 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. [8] proposed an approach to identify subsumption relations using automatic test suite generators in the context of strong mutation testing. In contrast, we use 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 [16] 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 [29] 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. [25] presented an empirical approach to define an appropriate set of selective mutation operators. The idea was to randomly select a subset of mutation operators [22], [33]. Perez et al. [3] explored Evolutionary Mutation Testing to reduce the number of mutants to be executed. Namin et al. [31] 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 [10],[9].

However, in another study, Gopinath et al. [7] 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. [15] stated that existing approaches to selective mutation take no account of program context and this is fundamental to avoid losing useful information.

Jia and Harman [11] introduced the concept of subsuming HOMs. They define a subsuming HOM as mutant that is harder to kill than their constituent FOMs. They suggested some approaches to find the subsuming HOMs by using some meta-heuristic algorithms: greedy algorithm, genetic algorithm and hill climbing algorithm. They experimented with 10 benchmark C programs under test (14850 LoC and 35473 test cases in total) and the results indicate that the genetic algorithm is the most efficient algorithm for finding those subsuming HOMs, while the greedy algorithm and the hill climbing algorithm can also be used to improve the quality of the results. In order to find higher order mutants that are hard to kill and more realistic complex faults, Langdon et al. [20] introduced a new form of mutation testing: Multi Objective Higher Order Mutation Testing. They find examples that pose challenges to testing in the higher order space that cannot be represented in the first order space.

Nguyen and Pham [24] performed an empirical study to investigate whether HOMs are harder to kill than FOMs. They selected 8 real-world open-source projects and used an extended Judy tool [21] as the supporting tool to generate HOMs, execute mutation testing and evaluate HOMs with the full set of built-in mutation operators of Judy. To generate the HOMs, they applied 6 different algorithms, 5 multi objective optimization algorithms and a random algorithm. To answer their research question, they focus on the ratio of number of test cases which can kill each generated HOM to the number of test cases which can kill each constituent FOMs. Their experimental results indicate that about 50% HOMs are harder to kill than their constituent FOMs.

Most recently and closer to our current work, our previous work proposes an approach consisting of six steps to discover subsumption relations among method-level mutants using theorem proving in the context of weak mutation testing [6]. We encode a theory of

Table 3: It presents the mutation targets, the amount of method-level first order and second order mutations that the operators are able to create in the corresponding target, the subsuming mutants set for each target identified in our approach, the size of the subsuming mutants set compared to the original set of mutants, and the amount of FOMs and HOMs in relation to the total size of the subsuming mutants set. OP_1: select CDL, ODL, or VDL.

Mutation Target	# FOMs	# HOMs	Subsuming Mutants Set	Size	FOMs ⊂ Minimal Set	HOMs ⊂ Minimal Set
lexp + rexp (for Z^+)	8	12	AORB(*)	5%	100%	0%
lexp - rexp (for Z^+)	8	12	OP_1(lexp)	5%	100%	0%
lexp * rexp (for Z^+)	8	12	AORB(+), OP_1(lexp), OP_1(rexp)	15%	100%	0%
lexp ˆ rexp (bool)	15	42	COI(lexp ˆ !rexp), COR(||), COI COR(lexp&&!rexp), COI COR(!lexp&&rexp)	7%	50%	50%
lexp && rexp	15	42	OP_1(lexp), OP_1(rexp), COR(False), ROR(==)	7%	100%	0%
lexp || rexp	15	42	OP_1(lexp), OP_1(rexp), COR(True), COR(ˆ)	7%	100%	0%
lexp == rexp (bool)	15	42	COR(&&), COI COR(!x||y), COI COR(x||!y), COI COR !(x||y)	7%	25%	75%
lexp != rexp (bool)	15	42	COR(||), COI COR(!x&&y), COI COR(x&&!y), COI COR !(x&&y),	7%	25%	75%
lexp == rexp	8	7	ROR(false), ROR(>=), ROR(<=)	2%	100%	0%
lexp != rexp	8	7	ROR(<), ROR(True), ROR(>)	2%	100%	0%
lexp > rexp	8	7	ROR(False), ROR(!=), ROR(>=)	2%	100%	0%
lexp >= rexp	8	7	ROR(True), ROR(==), ROR(>)	2%	100%	0%
lexp < rexp	8	7	ROR(False), ROR(!=), ROR(<=)	2%	100%	0%
lexp <= rexp	8	7	ROR(True), ROR(==), ROR(<)	2%	100%	0%
lexp != rexp (obj)	8	5	ROR(True), ROR(>), ROR(<)	2%	100%	0%
lexp & rexp	10	24	OP_1(lexp), OP_1(rexp), SOR(<<), SOR(>>)	11.8%	100%	0%
lexp | rexp	10	24	OP_1(lexp), OP_1(rexp), LOR(ˆ), SOR(>>)	11.8%	100%	0%
lexp ˆ rexp	10	24	LOR(|), SOR(<<), SOR(>>)	8.8%	100%	0%
lexp >> rexp	10	24	OP_1(lexp), OP_1(rexp), LOR(ˆ), LOR(|), LOR(&), SOR(<<)	17.6%	100%	0%
lexp << rexp	10	24	LOR(ˆ), SOR(>>), LOR(&)	8.8%	100%	0%
exp	6	9	AOIU(-exp), LOI AOIS ~(++exp)	13.3%	50%	50%
+exp	3	2	LOI(~exp)	20%	100%	0%
-exp	3	2	AODU(exp)	20%	100%	0%
++exp	4	3	LOI AODS(~exp)	14.3%	0%	100%
exp++	4	3	LOI(~exp)	14.3%	100%	0%
--exp	4	3	LOI AORS(~exp)	14.3%	0%	100%
exp--	4	3	LOI(~exp)	14.3%	100%	0%

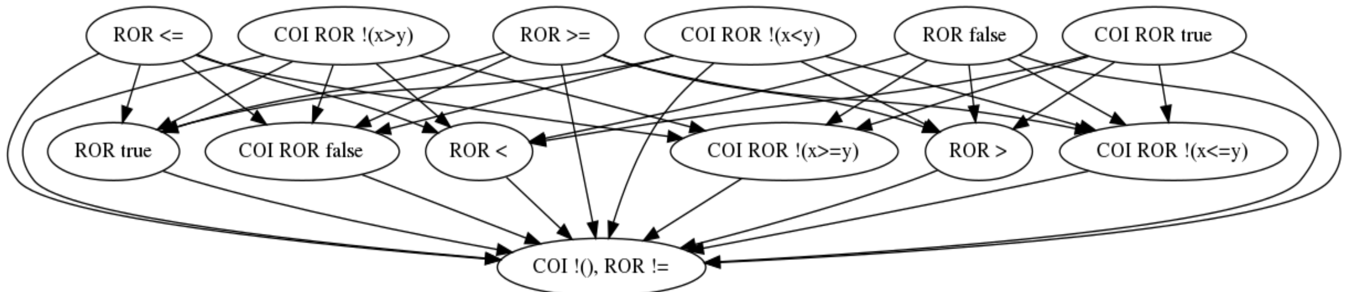

Figure 1: Mutation subsumption graph for the `lexp == rexp` mutation target. Mutations ROR(<=), ROR(>=), ROR(false), COI ROR!(x>y), COI ROR!(x<y), COI ROR(true) dominate the other mutations.

subsumption relations in Z3 and use its theorem prover [2] to automatically identify redundant mutants. We found that developers can avoid using on average 66.3% of mutations during mutation testing. Our previous work differ from our current work in the amount of encoded HOMs. Previously, we only encoded two HOMs, whereas currently we encode 438 HOMs.

6 CONCLUSIONS

In this work, we prove a number of subsumption relations for method-level FOMs and HOMs using our previous approach [6]. We encode 233 FOMS and 438 HOMs in 27 mutation targets and automatically prove a number of subsumption relations using Z3. Developers can avoid using on average 91% of mutations when considering both FOMs and HOMs. Moreover, most HOMs may be useless, since 97.5% of them could be discarded and HOMs only compose 16.67% of the subsuming mutants set on average. The results may help to challenge the importance of HOMs [11, 24] and also help to build better mutation testing tools that will allow to reduce the mutation testing costs.

As future work, we intend to prove more subsumption relations by considering more mutation targets, other language constructs and mutations. We may also compare the subsuming HOMs found by other approaches, such as the meta-heuristic algorithms used by Jia et al. [11], with the subsuming HOMs found by our approach to identify whether they differ or not. 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 of weak mutation testing.

REFERENCES

[1] Paul Ammann, Marcio Eduardo Delamaro, and Jeff Offutt. 2014. Establishing Theoretical Minimal Sets of Mutants. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation (ICST '14)*. IEEE Computer Society, USA, 21–30. https://doi.org/10.1109/ICST.2014.13

[2] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[3] Pedro Delgado-Pérez, Sergio Segura, and Inmaculada Medina-Bulo. 2017. Assessment of C++ object-oriented mutation operators: A selective mutation approach. *Softw. Test., Verif. Reliab.* 27 (2017).

[4] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (April 1978), 34–41. https://doi.org/10.1109/C-M.1978.218136

[5] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 416–419. https://doi.org/10.1145/2025113.2025179

[6] Rohit Gheyi, Márcio Ribeiro, Beatriz Souza, Marcio Guimarães, Leo Fernandes, Marcelo d'Amorim, Vander Alves, Leopoldo Teixeira, and Baldoino Fonseca. 2020. Identifying Method-Level Mutation Subsumption Relations using Z3 (To appear). *IST* (2020).

[7] R. Gopinath, I. Ahmed, M. A. Alipour, C. Jensen, and A. Groce. 2017. Mutation Reduction Strategies Considered Harmful. *IEEE Transactions on Reliability* 66, 3 (Sep. 2017), 854–874. https://doi.org/10.1109/TR.2017.2705662

[8] Marcio Guimarães, Leo Fernandes, Marcio Ribeiro, Marcelo d'Amorim, and Rohit Gheyi. 2020. Optimizing mutation testing by discovering dynamic mutant subsumption relations. In *Proceedings of the 13th International Conference on Software Testing, Verification and Validation(ICST)*. IEEE, To appear.

[9] Shamaila Hussain. 2008. Mutation Clustering. (01 2008).

[10] Changbin Ji, Zhenyu Chen, Baowen Xu, and Zhihong Zhao. 2009. A Novel Method of Mutation Clustering Based on Domain Analysis. In *Proceedings of the 21st International Conference on Software Engineering & Knowledge Engineering (SEKE'2009), Boston, Massachusetts, USA, July 1-3, 2009*. Knowledge Systems Institute Graduate School, 422–425.

[11] Y. Jia and M. Harman. 2009. Higher Order Mutation Testing. *Information and Software Technology* 51, 10 (2009), 1379 – 1393. https://doi.org/10.1016/j.infsof.2009.04.016 Source Code Analysis and Manipulation, SCAM 2008.

[12] Y. Jia and M. Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (Sep. 2011), 649–678. https://doi.org/10.1109/TSE.2010.62

[13] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 654–665. https://doi.org/10.1145/2635868.2635929

[14] Rene Just, Gregory M. Kapfhammer, and Franz Schweiggert. 2012. Do Redundant Mutants Affect the Effectiveness and Efficiency of Mutation Analysis?. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST '12)*. IEEE Computer Society, USA, 720–725. https://doi.org/10.1109/ICST.2012.162

[15] René Just, Bob Kurtz, and Paul Ammann. 2017. Inferring Mutant Utility from Program Context. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 284–294. https://doi.org/10.1145/3092703.3092732

[16] René Just and Franz Schweiggert. 2015. Higher Accuracy and Lower Run Time: Efficient Mutation Analysis Using Non-Redundant Mutation Operators. *Softw. Test. Verif. Reliab.* 25, 5–7 (Aug. 2015), 490–507. https://doi.org/10.1002/stvr.1561

[17] Gary Kaminski, Paul Ammann, and Jeff Offutt. 2011. Better Predicate Testing. In *Proceedings of the 6th International Workshop on Automation of Software Test (AST '11)*. Association for Computing Machinery, New York, NY, USA, 57–63. https://doi.org/10.1145/1982595.1982608

[18] Marinos Kintis, Mike Papadakis, and Nicos Malevris. 2010. Evaluating Mutation Testing Alternatives: A Collateral Experiment. In *Proceedings of the 2010 Asia Pacific Software Engineering Conference (APSEC '10)*. IEEE Computer Society, USA, 300–309. https://doi.org/10.1109/APSEC.2010.42

[19] Bob Kurtz, Paul Ammann, Marcio E. Delamaro, Jeff Offutt, and Lin Deng. 2014. Mutant Subsumption Graphs. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation Workshops (ICSTW '14)*. IEEE Computer Society, USA, 176–185. https://doi.org/10.1109/ICSTW.2014.20

[20] William B. Langdon, Mark Harman, and Yue Jia. 2009. Multi Objective Higher Order Mutation Testing with Genetic Programming. In *Proceedings of the 2009 Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC-PART '09)*. IEEE Computer Society, USA, 21–29. https://doi.org/10.1109/TAICPART.2009.18

[21] L. Madeyski and N. Radyk. 2010. Judy - a mutation testing tool for java. *IET Software* 4, 1 (Feb 2010), 32–42. https://doi.org/10.1049/iet-sen.2008.0038

[22] A. P. Mathur. 1991. Performance, effectiveness, and reliability issues in software testing. In *[1991] Proceedings The Fifteenth Annual International Computer Software Applications Conference*. 604–605. https://doi.org/10.1109/CMPSAC.1991.170248

[23] Quang-Vu Nguyen and Lech Madeyski. 2014. Problems of Mutation Testing and Higher Order Mutation Testing. *Advances in Intelligent Systems and Computing* 282 (01 2014), 157–172. https://doi.org/10.1007/978-3-319-06569-4_12

[24] Quang-Vu Nguyen and Duong-Thu-Hang Pham. 2018. Is Higher Order Mutant Harder to Kill Than First Order Mutant? An Experimental Study. In *Intelligent Information and Database Systems*, Ngoc Thanh Nguyen, Duong Hung Hoang, Tzung-Pei Hong, Hoang Pham, and Bogdan Trawiński (Eds.). Springer International Publishing, Cham, 664–673.

[25] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. 1996. An Experimental Determination of Sufficient Mutant Operators. *ACM Trans. Softw. Eng. Methodol.* 5, 2 (April 1996), 99–118. https://doi.org/10.1145/227607.227610

[26] J. Offutt, Y. Ma, and Y. Kwon. 2006. MuJava: a mutation system for java. In *Software Engineering, International Conference on*. IEEE Computer Society, Los Alamitos, CA, USA, 827–830. https://doi.org/10.1145/1134285.1134425

[27] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, USA, 75–84. https://doi.org/10.1109/ICSE.2007.37

[28] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Threats to the Validity of Mutation-Based Test Assessment. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 354–365. https://doi.org/10.1145/2931037.2931040

[29] Mike Papadakis and Nicos Malevris. 2010. An Empirical Evaluation of the First and Second Order Mutation Testing Strategies. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW '10)*. IEEE Computer Society, USA, 90–99. https://doi.org/10.1109/ICSTW.2010.50

[30] Alessandro Pizzoleto, Fabiano Ferrari, Jeff Offutt, Leo Fernandes, and Márcio Ribeiro. 2019. A Systematic Literature Review of Techniques and Metrics to Reduce the Cost of Mutation Testing. *Journal of Systems and Software* 157 (07 2019). https://doi.org/10.1016/j.jss.2019.07.100

[31] Akbar Siami Namin, James H. Andrews, and Duncan J. Murdoch. 2008. Sufficient Mutation Operators for Measuring Test Effectiveness. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. Association for Computing Machinery, New York, NY, USA, 351–360. https://doi.org/10.1145/1368088.1368136

[32] Beatriz Souza and Rohit Gheyi. 2020. Most Higher Order Mutants are Useless for Method-Level Mutation Operators Using Weak Mutation (Artifact). https://drive.google.com/drive/folders/1L8qSCSFOUX5-gxBXw1iraKdczOmHoj62?usp=sharing Accessed: 2020-06-10.

[33] W. Eric Wong and Aditya P. Mathur. 1995. Reducing the Cost of Mutation Testing: An Empirical Study. *J. Syst. Softw.* 31, 3 (Dec. 1995), 185–196. https://doi.org/10.1016/0164-1212(94)00098-0