Identifying Mutation Subsumption Relations

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

ABSTRACT

One recent promising direction in reducing costs of mutation analysis is to identify redundant mutations. We propose a technique to discover redundant mutations by proving subsumption relations among method-level mutation operators using weak mutation testing. We conceive and encode a theory of subsumption relations in Z3 for 40 mutation targets (mutations of an expression or statement). Then we prove a number of subsumption relations using the Z3 theorem prover, and reduce the number of mutations in a number of mutation targets. MuJava-M includes some subsumption relations in MuJava. We apply MuJava and MuJava-M to 187 classes of 17 projects. Our approach correctly discards mutations in 74.97% of the cases, and reduces the number of mutations by 72.52%.

KEYWORDS

subsumption relations, redundant mutations, theorem proving

ACM Reference Format:
Beatriz Souza. 2020. Identifying Mutation Subsumption Relations. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3324884.3418921

1 RESEARCH PROBLEM AND MOTIVATION

Mutation analysis is a popular technique to assess quality of test suites [4, 17, 20]. 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 [19]. We may 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 [12, 19]. Ammann et al. [1] empirically identified that a number of the generated mutants are redundant. Also, Papadakis et al. [21] 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.

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.
ASE '20, September 21–25, 2020, Virtual Event, Australia
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3418921

2 BACKGROUND AND RELATED WORK

Kaminski et al. [11] 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. [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. Marcozzi et al. [15] propose a technique to prune out infeasible and redundant objectives, by proving the validity of logical assertions in the code under test. However, their approach becomes particularly costly when applied on the largest programs with the most meticulous criteria. Guimarães et al. [7] propose an approach to yield dynamic subsumption relations among method-level mutations by using automatic test suite generators [5, 18] 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.

3 APPROACH

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 [8]. We encode a theory of subsumption relations in Z3 and use its theorem prover [3] to automatically identify redundant mutations. We use the latest version of Z3 after fixing the bugs found [22]. We consider most of the method-level mutation operators available in the MuJava tool [14], which are First Order Mutations (FOMs). We also encode and prove subsumption relations among Higher Order Mutations (HOMs), which are created by combining FOMs, using our approach. It is important to notice that 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 [9].

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. Notice that, for the `lexp == rexp` mutation target, we can apply two mutation operators, ROR and COI (see Table 1), and generate eight FOMs: `ROR !=`, `ROR >`, `ROR >=`, `ROR <`, `ROR <=`, `ROR true`, `ROR false`, and `COI !(==)`. Moreover, combining ROR and COI we can generate seven HOMs: `COI ROR !(!=)`, `COI ROR !(>)`, `COI ROR !(>=)`, `COI ROR !(<)`, `COI ROR !(<=)`, `COI ROR !(true)`, `COI ROR !(false)`.

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

Table 1: It presents the minimal set of mutations for 10 out of 40 targets identified in our approach, and the size of the minimal set of mutations compared to the original one. OP_1: select CDL, ODL, or VDL. The mutations in red are HOMs.

Mutation Target	Mutation Operators	Minimal Set	Size
`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( ^ )`, `COR(true)`	33.3%
`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%
`++exp`	AORS (1), AODS (1), LOI (1), ODL (1), `LOI AORS (1)`, `LOI AODS (1)`, `LOI ODL (1)`	LOI AODS(~exp)	14.3%
`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%
`lexp == rexp`	ROR (7), COI (1), `ROR COI (7)`	`ROR(false)`, `ROR(>=)`, `ROR(<=)`	20.0%
`lexp != rexp`	ROR (7), COI (1), `ROR COI (7)`	`ROR(<)`, `ROR(true)`, `ROR(>)`	20.0%

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, an unary expression. Some mutants (exp++, and exp−−) are equivalent to the program exp in our encoding using weak mutation testing.

Listing 1: Subsumption Relations for `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, ..., False, True]
# Step 4
muts = removeEquivalentMutants(p, muts)
# Step 5
muts = compareAllMuts(p, muts, conds)
# Step 6
identifyRedundantMutants(muts)
```

To identify all subsumption relations in Step 5, we have to call the `compareAllMuts` function passing `p` and `muts` as parameters. For the `lexp == rexp` mutation target, our results indicate that we only need to use the following mutations: `ROR <=`, `ROR >=`, `ROR false`, `COI ROR !(>)`, `COI ROR !(<)`, `COI ROR !(true)`. These mutations dominate the others. 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.

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 >=` and `COI ROR !(<)`, `ROR <=` and `COI ROR !(>)`. 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 mutations set contains only three mutations: `ROR false`, `ROR >=`, and `ROR <=`, which represents a reduction of 80% of the mutations for this target. Notice that, for some targets, the minimal mutations set is composed only by HOMs, as for the `++exp` mutation target (see Table 1).

4 EVALUATION

Weak mutation testing [8] is a modification to mutation testing that is computationally more efficient, and can be applied in a manner that is almost as effective as mutation testing [16]. However, Lindström and Márki [13] find that the subsumption relations for ROR identified using weak mutation testing do not hold for strong mutation testing. We evaluate our subsumption relations in the context of strong mutation testing. MuJava-M includes some subsumption relations in MuJava [7]. We applied MuJava and MuJava-M to 17 large real open source projects. We analyzed 1,571 occurrences of mutation targets in 187 classes. MuJava generated 7,344 mutants and MuJava-M generated 2,018 mutants. We use EvosuiteR, a specialization of Evosuite [5], to generate the minimal test sets. Our approach achieves an effectiveness of 74.97% and a reduction rate of 72.52%.

Gopinath et al. [6] find that none of the mutation reduction strategies evaluated produced an effectiveness advantage larger than 5% in comparison with random sampling. We compared our approach with random sampling. The random sampling approach yields an average effectiveness of 75% in correctly identifying the baseline minimal set of mutants when it uses a reduction rate of 40%. Our approach presented yields an effectiveness of 74.97% when it uses a reduction rate of 72.52%.

5 CONCLUSION

We automatically identify and prove, in few seconds, a number of subsumption relations using Z3, and reduce the number of mutations in a number of mutation targets. We use MuJava and MuJava-M [7] in 187 classes of 17 real projects. Our approach achieves an effectiveness of 74.97% and a reduction rate of 72.52%. 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. All artifacts are available [2].

ACKNOWLEDGMENTS

Thanks to my supervisor, Rohit Gheyi, for his support and guidance.

REFERENCES

[1] Paul Ammann, Marcio Eduardo Delamaro, and Jeff Offutt. 2014. Establishing theoretical minimal sets of mutants. In *ICST* (Cleveland, OH, USA). IEEE, Piscataway, NJ, USA, 21–30.

[2] Artifacts. 2020. https://colab.research.google.com/drive/1kldmXBGgJIOEpCCr6o9cm3P0Jnz7mMix?usp=sharing. (2020).

[3] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS* (Budapest, Hungary). Springer, Berlin, Germany, 337–340.

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

[5] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *FSE* (Szeged, Hungary). ACM, New York, NY, USA, 416–419.

[6] Rahul Gopinath, Iftekhar Ahmed, Mohammad Alipour, Carlos Jensen, and Alex Groce. 2017. Mutation Reduction Strategies Considered Harmful. *IEEE TR* 66 (2017), 854–874. Issue 3.

[7] Marcio Guimarães, Leo Fernandes, Márcio Ribeiro, Marcelo d'Amorim, and Rohit Gheyi. 2020. Optimizing Mutation Testing by Discovering Dynamic Mutant Subsumption Relations. In *ICST* (Porto, Portugal). IEEE, Piscataway, NJ, USA, To appear.

[8] William Howden. 1982. Weak Mutation Testing and Completeness of Test Sets. *TSE* 8, 4 (1982), 371–379.

[9] Y. Jia and M. Harman. 2009. Higher Order Mutation Testing. *IST* 51, 10 (2009), 1379 – 1393.

[10] René Just, Gregory M Kapfhammer, and Franz Schweiggert. 2012. Do redundant mutants affect the effectiveness and efficiency of mutation analysis?. In *ICST* (Montreal, QC, Canada). IEEE, Piscataway, NJ, USA, 720–725.

[11] Gary Kaminski, Paul Ammann, and Jeff Offutt. 2011. Better predicate testing. In *AST* (Waikiki, Honolulu, HI, USA). ACM, New York, NY, USA, 57–63.

[12] Marinos Kintis, Mike Papadakis, and Nicos Malevris. 2010. Evaluating mutation testing alternatives: A collateral experiment. In *APSEC* (Sydney, NSW, Australia). IEEE, Piscataway, NJ, USA, 300–309.

[13] Birgitta Lindström and András Márki. 2019. On strong mutation and the theory of subsuming logic-based mutants. *STVR* 29, 1-2 (2019), e1667.

[14] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. 2005. MuJava: an automated class mutation system. *STVR* 15, 2 (2005), 97–133.

[15] M. Marcozzi, S. Bardin, N. Kosmatov, M. Papadakis, V. Prevosto, and L. Correnson. 2018. Time to Clean Your Test Objectives. In *ICSE*. ACM, Gothenburg, Sweden, 456–467.

[16] A. Jefferson Offutt and Stephen D. Lee. 1994. An empirical evaluation of weak mutation. *TSE* 20, 5 (1994), 337–344.

[17] Jeff Offutt. 2011. A mutation carol: Past, present and future. *IST* 53, 10 (2011), 1098 – 1107.

[18] Carlos Pacheco, Shuvendu Lahiri, Michael Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *ICSE* (Minneapolis, MN, USA). IEEE, Piscataway, NJ, USA, 75–84.

[19] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Threats to the Validity of Mutation-based Test Assessment. In *ISSTA* (Saarbrücken, Germany). ACM, New York, NY, USA, 354–365.

[20] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Chap. six - Mutation Testing Advances: An Analysis and Survey. *ADV COMPUT* 112 (2019), 275–378.

[21] Mike Papadakis and Nicos Malevris. 2010. An empirical evaluation of the first and second order mutation testing strategies. In *ICST* (Paris, France). IEEE, Piscataway, NJ, USA, 90–99.

[22] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT Solvers via Semantic Fusion. In *PLDI* (London, UK). ACM, New York, NY, USA, 718–730.