Efficiently Finding Useless Mutants

Beatriz Souza[1] **and Rohit Gheyi**[1]

[1] Departamento de Sistemas e Computação
Universidade Federal de Campina Grande (UFCG) – Campina Grande, PB – Brazil

{beatriz.souza@ccc.ufcg.edu.br,rohit@dsc.ufcg.edu.br}

Abstract. *Mutation analysis is a popular but costly approach to assess the quality of test suites. Equivalent and redundant mutants contribute to increase costs and are harmful to the design of test suites. We propose a lightweight technique to identify equivalent and redundant mutants based on theorem proving with Z3 in the context of weak mutation testing. The experiments reveal that our technique detects all equivalent mutants detected by TCE and we have an average reduction of 72.52% of mutants, when considering entire programs. We also apply our technique on HOMs. When considering both FOMs and HOMs, 91% of the mutations could be discarded on average. The results found by our approach may help to make mutation testing less expensive and more accurate.*

1. Introduction

Mutation analysis is a powerful technique to assess quality of test suites [DeMillo et al. 1978, Offutt 2011, Papadakis et al. 2019]. 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. Just et al. [Just et al. 2014] empirically identify a statistically significant correlation between mutant detection and real fault detection.

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 suites 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 [Petrovic and Ivankovic 2018].

However, some mutants are equivalent or redundant, that is, they may not be necessary for the effectiveness of mutation analysis and thus we may discard them [Papadakis et al. 2016]. Equivalent mutants have the same behavior as the original program [Budd and Angluin 1982, Jia and Harman 2011, Madeyski et al. 2014]. Redundant mutants are killed when other mutants are also killed [Kintis et al. 2010, Papadakis et al. 2016]. Then, the generation of these mutants increases the total cost and does not help to improve the test suite.

Madeyski et al. [Madeyski et al. 2014] report that the rate of equivalent mutants might lie between 4% and 39%. In addition, manually checking mutant equivalence is error-prone (people judged equivalence correctly in about 80% of the cases [Acree 1980]) and time consuming (approximately 15 minutes per equivalent mutant [Schuler and Zeller 2013]). Ammann et al. [Ammann et al. 2014] empirically identified that almost 99% of the generated mutants are redundant. Also, Papadakis et

al. [Papadakis and Malevris 2010] 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.

Weak mutation testing 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 [Offutt and Lee 1994]. Weak mutation testing requires that a test case causes a mutated program component to compute a different value than the program component. Mutation testing, on the other hand, requires that a test case causes a mutated program to compute a different value than the program [Offutt and Lee 1994].

We propose a lightweight technique consisting of six steps to discover equivalent and redundant mutants using theorem proving in the context of weak mutation testing [Howden 1982]. We encode a theory of equivalence and redundancy in Z3 and use its theorem prover [de Moura and Bjørner 2008] to automatically identify equivalent and redundant mutants (Section 2). Our technique is lightweight, we do not need to create mutants, compile them, create test suites, and execute them.

We apply our technique to 40 mutation targets, considering most of the method method-level mutation operators available in MuJava. Our technique identifies 13 equivalences for seven mutation targets and, removing redundant mutations, reduces 59% of the mutations on average in the context of weak mutation testing.

Moreover, we investigate whether our results hold in the context of strong mutation testing. To evaluate our approach, we apply MuJava [Ma et al. 2005], a tool that generates mutants for programs written in Java, to 20 mutation targets in 5 real large projects. Then, we ran Trivial Compiler Equivalence *(TCE)* [Kintis et al. 2017], which is a sound tool to find equivalent mutants, against the mutants generated by MuJava. Our technique detects all equivalent mutants detected by TCE, but with less effort. We also modified MuJava to include our results for 24 mutation targets. We call MuJava-M this new version of MuJava. Then we apply MuJava and MuJava-M to 5 real large projects. MuJava generated 7,386 mutants and MuJava-M generated 2,041 mutants. We have an average reduction of 72.52% of mutants, when considering entire programs.

Based on the types of faults seeded, mutants can be classified as First Order Mutants (FOMs) and Higher Order Mutants (HOMs). First order mutants seed only simple faults, generated by a single syntactic change to the original program. Higher order mutation testing (HOMT) [Jia and Harman 2009], 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 [Nguyen and Madeyski 2014]. 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 [Jia and Harman 2009]. We also investigate equivalence and redundant relations among mutations for HOMs using our approach. We create 233 FOMs and 438 HOMs for 27 mutation targets. When considering both FOMs and HOMs, 91% of the mutations could be discarded on average.

The results found by our approach may help to build better mutation testing tools that will allow to reduce the mutation testing costs and help testers to evaluate more accurately the strength of their test suites.

2. Technique

We propose a technique using the Z3 [de Moura and Bjørner 2008] API for Python, which has a theorem prover, to identify equivalent and redundant mutations using weak mutation testing. We focus on two types of redundant mutations: duplicate and subsumed. Duplicate mutations are semantically equal mutations [Marcozzi et al. 2018]. A mutation `m1` subsumes a mutation `m2` if whenever `m1` is detected, `m2` is also detected [Guimarães et al. 2020]. For each mutation target, the main steps of our approach are the following:

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

A mutation target is a language expression or statement in which it is possible to apply a set of mutations of one or more mutation operators (e.g., `a + b`, `a > b`, `exp++`, etc) [Guimarães et al. 2020].

Steps 1 and 2 are required to instantiate a mutation target in the Z3 [de Moura and Bjørner 2008] API for Python. In Step 3, we specify a list of mutations for the instantiated target. For Steps 4-6, we encode a theory in Z3 to detect equivalent, duplicate, and subsumed mutants. We use the latest version of Z3 after fixing the bugs found by Winterer et al. [Winterer et al. 2020].

Listing 1 specifies how to prove a theorem using the Z3 Python API. It can yield three answers: the theorem is valid or invalid, or it does not know the answer. The command `Solver` creates a general purpose solver in Z3 [de Moura and Bjørner 2008]. Constraints can be added using the `add` function. The `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. Z3 does not yield `unknown` in our study.

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
```

The `identifyEquivalentMutants` function, presented in Listing 2, returns equivalent mutants of a program.

Listing 2. Identifying equivalent mutants in Z3.

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

2.1. Running Example

Next we show how to use our approach to identify some equivalence relations for the `lexp != rexp` mutation target. 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 3. Then, in Step 2, we specify the program. In Step 3, we declare the FOMs based on the method-level mutation operators available in MUJAVA. Moreover, we declare the HOMs by combining the FOMs (See Listing 3).

Listing 3. Identify Equivalent Mutants for `lexp != rexp` target.

```
# Step 1
x = Int('x')
y = Int('y')

# Step 2
p = x!=y

# Step 3
muts = [x==y, x>y, x>=y, x<y, x<=y, True, False, Not(p),
        Not(x==y), Not(x>y), Not(x>=y), Not(x<y),
        Not(x<=y), False, True]

# Step 4
identifyEquivalentMutants(p, muts)
```

Notice that for the `lexp != rexp` mutation target, we can apply two mutation operators, `ROR` and `COI` (see Table 1), and generate eight FOMs using MUJAVA: `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)`.

To identify all equivalent mutants in Step 4, we have to call the `identifyEquivalentMutants` function passing `p` and `muts` as parameters. For the `lexp != rexp` mutation target, our script indicates that the `COI ROR !(==)` mutant is equivalent.

3. Evaluation

To evaluate our approach, we consider the following research questions:

RQ$_1$ How many equivalent mutants does our approach detect using weak mutation testing? How does our approach compare with Trivial Compiler Equivalence, which is one of the best techniques available to detect equivalent mutants?

RQ$_2$ How many subsumed mutants does our approach detect using weak mutation testing? To what extent our results hold for complete programs?

RQ$_3$ How many mutants could be discarded when considering both FOMs and HOMs?

3.1. RQ$_1$: Number of Equivalent Mutants

We evaluate our technique, considering Steps 1-4, in 40 mutation targets applying most MUJAVA method-level mutation operators [Ma et al. 2005], 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. It presents mutation targets, method-level mutations that each operator is able to create in the corresponding target, the set of equivalent mutants for each target identified in our approach, and the percentage of equivalent mutants.

Mutation Target	Mutation Operators	Equivalent Mutants	Percentage
`lexp + rexp`	AORB (2), VDL (2), CDL (2), ODL (2)	-	0.0%
`lexp - rexp`	AORB (2), VDL (2), CDL (2), ODL (2)	-	0.0%
`lexp * rexp`	AORB (2), VDL (2), CDL (2), ODL (2)	-	0.0%
`lexp ^ rexp (bool)`	COR (4), ROR(2), COI (3), VDL (2), CDL (2), ODL (2)	ROR(!=)	6.67%
`lexp && rexp`	COR (4), ROR(2), COI (3), VDL (2), CDL (2), ODL (2)	-	0.0%
`lexp || rexp`	COR (4), ROR(2), COI (3), VDL (2), CDL (2), ODL (2)	-	0.0%
`lexp == rexp (bool)`	ROR (1), COI (3), VDL (2), CDL (2), ODL (2)	-	0.0%
`lexp != rexp (bool)`	ROR (1), COI (3), VDL (2), CDL (2), ODL (2)	-	0.0%
`lexp == rexp`	ROR (7), COI (1)	-	0.0%
`lexp != rexp`	ROR (7), COI (1)	-	0.0%
`lexp > rexp`	ROR (7), COI (1)	-	0.0%
`lexp >= rexp`	ROR (7), COI (1)	-	0.0%
`lexp < rexp`	ROR (7), COI (1)	-	0.0%
`lexp <= rexp`	ROR (7), COI (1)	-	0.0%
`lexp != rexp (obj)`	ROR (7), COI (1)	-	0.0%
`lexp & rexp`	LOR (2), VDL (2), CDL (2), ODL (2)	-	0.0%
`lexp | rexp`	LOR (2), VDL (2), CDL (2), ODL (2)	-	0.0%
`lexp ^ rexp`	LOR (2), SOR (2), CDL (2), ODL (2)	-	0.0%
`lexp >> rexp`	LOR (3), SOR (1), VDL (2), CDL (2), ODL (2)	-	0.0%
`lexp << rexp`	LOR (3), SOR (1), VDL (2), CDL (2), ODL (2)	-	0.0%
`exp`	AOIS (4), AOIU (1), LOI (1)	AOIS(exp–), AOIS(exp++)	33.33%
`+exp`	AODU (1), LOI (1), ODL (1)	AODU(exp), ODL(exp)	66.67%
`-exp`	AODU (1), LOI (1), ODL (1)	-	0.0%
`++exp`	AORS (1), AODS (1), LOI (1), ODL (1)	-	0.0%
`exp++`	AORS (1), AODS (1), LOI (1), ODL (1)	AORS(exp), AODS(exp), ODL(exp)	75%
`--exp`	AORS (1), AODS (1), LOI (1), ODL (1)	-	0.0%
`exp--`	AORS (1), AODS (1), LOI (1), ODL (1)	AORS(exp), AODS(exp), ODL(exp)	75%
`!exp`	COD (1), ODL (1)	-	0.0%
`~exp`	AODU (1), LOD (1), ODL (1)	-	0.0%
`lhs += rhs`	ASRS (2), ODL (1), SDL (1)	-	0.0%
`lhs -= rhs`	ASRS (2), ODL (1), SDL (1)	-	0.0%
`lhs *= rhs`	ASRS (2), ODL (1), SDL (1)	-	0.0%
`lhs <<= rhs`	ASRS (1), ODL (1), SDL (1)	-	0.0%
`lhs >>= rhs`	ASRS (1), ODL (1), SDL (1)	-	0.0%
`lhs &= rhs`	ASRS (2), ODL (1), SDL (1)	-	0.0%
`lhs |= rhs`	ASRS (2), ODL (1), SDL (1)	-	0.0%
`lhs ^= rhs`	ASRS (2), ODL (1), SDL (1)	-	0.0%
`lexp == rexp`	ROR (7), COI (1), ROR COI (7)	COI ROR !(!=)	6.67%
`lexp != rexp`	ROR (7), COI (1), COI ROR (7)	COI ROR !(==)	6.67%
`++exp`	ROR (7), COI (1), COI ROR (7)	-	0.0%

Table 1 presents 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 specify the set of corresponding mutation operators able to apply mutations into the target [Guimarães et al. 2020]. For each operator, we provide the number of possible mutations (in parentheses) that such operator can apply into the target. For

example, the Logical Operator Replacement (LOR) operator can apply two mutations to the `lexp | rexp` target.

For the 40 mutation targets presented in Table 1, our technique found equivalent mutants for seven of them. We find 13 mutations that yield equivalent mutants in total. For example, for the mutation targets `exp++` and `exp--`, our approach classified the following mutations as equivalent: `AORS(exp)`, `AODS(exp)`, and `ODL(exp)`. We manually analyze whether the equivalent mutants detected by our technique are indeed equivalent. We do not find false positives.

TCE [Papadakis et al. 2015] is one of the best static analysis tools to detect some types of equivalent mutants. A direct comparison of our approach with *TCE* is not possible, as TCE aims at identifying strong mutant equivalences. However, we can always assume that weakly equivalent mutants are also strongly equivalent ones [Marcozzi et al. 2018].

To compare our approach with *TCE*, we apply MUJAVA to 5 real projects, which are described in Table 2, and generate mutants for 20 mutation targets. MUJAVA generates 5,297 mutants. We executed *TCE* against the 5,297 mutants generated by MUJAVA. *TCE* found 406 equivalent mutants in total. All of them created by the `AOIS exp--` and `AOIS exp++` mutations of the `exp` mutation target. In our approach using weak mutation testing, we also find the same equivalent mutants for the `exp` mutation target.

Table 2. Five programs used in our evaluation.

Project	Version	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

3.2. RQ_2: Number of Subsumed Mutants

We evaluate our technique, considering Steps 1-6, in the 40 mutation targets presented in Table 1. We have an average reduction of 59% of mutations in the context of weak mutation testing.

To evaluate our approach, considering entire programs, we modified MUJAVA to include our results for 24 mutation targets. We call MUJAVA-M this new version of MUJAVA. Then we apply MUJAVA and MUJAVA-M to the five projects presented in Table 2. MUJAVA generated 7,386 mutants and MUJAVA-M generated 2,041 mutants. We have an average reduction of 72.52% of mutants, when considering entire programs.

More details regarding this question and the results that we obtained are available in our previous article [Gheyi et al. 2021].

3.3. RQ_3: Higher Order Mutants

We apply our technique, considering Steps 1-6, in the 27 mutation targets presented in Table 3. For each target, we encode HOMs of second order, combining two FOMs.

For the 27 mutation targets presented in Table 3, we create 233 FOMs and 438 HOMs. 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.

We found that HOMs compose just 16.67% of all the mutants present in the subsuming mutants set on average (See the HOMs $\subset$ Minimal Set column in Table 3), whereas FOMs compose 83.33% (See the FOMs $\subset$ Minimal Set column in Table 3). HOMs are present in the subsuming mutants set of six out of the 27 mutation targets, as can be seen in Table 3. 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 HOMs and only 11 HOMs are as hard to kill as FOMs.

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 $\subset$ Minimal Set	HOMs $\subset$ 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 ^ rexp (bool)`	15	42	COI(lexp ^ !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(^)	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(^), SOR(>>)	11.8%	100%	0%
`lexp ^ rexp`	10	24	LOR(|), SOR(<<), SOR(>>)	8.8%	100%	0%
`lexp >> rexp`	10	24	OP_1(lexp), OP_1(rexp), LOR(^), LOR(|), LOR(&), SOR(<<)	17.6%	100%	0%
`lexp << rexp`	10	24	LOR(^), 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%

4. Related Work

There are several veins of research that are related to this work. Fernandez et al. developed various rules for Java programs to detect equivalent and redundant mutants [Fernandes et al. 2017]. Marcozzi et al. proposed a sound and scalable technique to prune out a significant part of the infeasible and redundant objectives produced by a panel of white-box criteria [Marcozzi et al. 2018].

There has been a lot of focus on computational costs and the equivalent mutant problem [Jia and Harman 2011]. There is much focus on avoiding redundant mutants, which leads to increase of computational costs and inflation of the mutation score [Just and Schweiggert 2015]. In our work, we use the Z3 theorem prover to detect equivalent and redundant mutants.

5. Conclusion

In this work, we propose a lightweight technique to detect equivalent and redundant mutants using Z3. Developers only need to specify the types and mutations in our encoding to identify equivalent and redundant mutants. We do not need to create mutants, compile them, create test suites, and execute them, as previous works [Fernandes et al. 2017].

The experiments reveal that our technique detects all equivalent mutants detected by TCE and we have an average reduction of 72.52% of mutants, when considering entire programs. We also apply our technique on HOMs. When considering both FOMs and HOMs, 91% of the mutations could be discarded on average.

The results found by our approach may help to build better mutation testing tools that will allow to reduce the mutation testing costs and help testers to evaluate more accurately the strength of their test suites.

Note: The first author encoded and proved equivalences and redundancies for some FOMs and all HOMs. Moreover, she also conducted the evaluation by applying MUJAVA to 5 projects to evaluate our approach in the context of strong mutation testing.

References

Acree, J. A. T. (1980). *On mutation*. PhD thesis, Georgia Institute of Technology.

Ammann, P., Delamaro, M. E., and Offutt, J. (2014). Establishing theoretical minimal sets of mutants. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, page 21–30, USA. IEEE Computer Society.

Budd, T. and Angluin, D. (1982). Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45.

de Moura, L. M. and Bjørner, N. (2008). Z3: an efficient SMT solver. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340.

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

Fernandes, L., Ribeiro, M., Carvalho, L., Gheyi, R., Mongiovi, M., Santos, A., Cavalcanti, A., Ferrari, F., and Maldonado, J. C. (2017). Avoiding useless mutants. In *Proceedings of the Generative Programming: Concepts & Experiences*, pages 187–198.

Gheyi, R., Ribeiro, M., Souza, B., Guimarães, M., Fernandes, L., d'Amorim, M., Alves, V., Teixeira, L., and Fonseca, B. (2021). Identifying method-level mutation subsumption relations using Z3. *Information and Software Technology*, 132:106496.

Guimarães, M., Fernandes, L., Ribeiro, M., d'Amorim, M., and Gheyi, R. (2020). Optimizing mutation testing by discovering dynamic mutant subsumption relations. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 198–208. IEEE.

Howden, W. (1982). Weak mutation testing and completeness of test sets. *Transactions on Software Engineering*, 8(4):371–379.

Jia, Y. and Harman, M. (2009). Higher order mutation testing. *Information and Software Technology*, 51(10):1379 – 1393.

Jia, Y. and Harman, M. (2011). An analysis and survey of the development of mutation testing. *Transactions on Software Engineering*, 37(5):649–678.

Just, R., Jalali, D., Inozemtseva, L., Ernst, M. D., Holmes, R., and Fraser, G. (2014). Are mutants a valid substitute for real faults in software testing? In *Proceedings of the Foundations of Software Engineering*, pages 654–665.

Just, R. and Schweiggert, F. (2015). Higher accuracy and lower run time: efficient mutation analysis using non-redundant mutation operators. *Software Testing, Verification and Reliability*, 25(5-7):490–507.

Kintis, M., Papadakis, M., Jia, Y., Malevris, N., Traon, Y. L., and Harman, M. (2017). Detecting trivial mutant equivalences via compiler optimisations. *Transactions on Software Engineering*, 44(4):308–333.

Kintis, M., Papadakis, M., and Malevris, N. (2010). Evaluating mutation testing alternatives: A collateral experiment. In *Proceedings of the 2010 Asia Pacific Software Engineering Conference*, APSEC '10, page 300–309, USA. IEEE Computer Society.

Ma, Y.-S., Offutt, J., and Kwon, Y.-R. (2005). MuJava: an automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97–133.

Madeyski, L., Orzeszyna, W., Torkar, R., and Jozala, M. (2014). Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *Transactions on Software Engineering*, 40(1):23–42.

Marcozzi, M., Bardin, S., Kosmatov, N., Papadakis, M., Prevosto, V., and Correnson, L. (2018). Time to clean your test objectives. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 456–467, New York, NY, USA. Association for Computing Machinery.

Nguyen, Q.-V. and Madeyski, L. (2014). Problems of mutation testing and higher order mutation testing. *Advances in Intelligent Systems and Computing*, 282:157–172.

Offutt, A. J. and Lee, S. D. (1994). An empirical evaluation of weak mutation. *Transactions on Software Engineering*, 20(5):337–344.

Offutt, J. (2011). A mutation carol: Past, present and future. *Information and Software Technology*, 53(10):1098 – 1107.

Papadakis, M., Henard, C., Harman, M., Jia, Y., and Le Traon, Y. (2016). Threats to the validity of mutation-based test assessment. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 354–365, New York, NY, USA. Association for Computing Machinery.

Papadakis, M., Jia, Y., Harman, M., and Le Traon, Y. (2015). Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *ICSE*, ICSE '15, pages 936–946, Piscataway, NJ, USA. IEEE Press.

Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Traon, Y. L., and Harman, M. (2019). Chapter six - mutation testing advances: An analysis and survey. *Advances in Computers*, 112:275–378.

Papadakis, M. and Malevris, N. (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, page 90–99, USA. IEEE Computer Society.

Petrovic, G. and Ivankovic, M. (2018). State of mutation testing at google. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 163–171.

Schuler, D. and Zeller, A. (2013). Covering and uncovering equivalent mutants. *Software Testing, Verification and Reliability*, 23(5):353–374.

Winterer, D., Zhang, C., and Su, Z. (2020). Validating SMT solvers via semantic fusion. In *Proceedings of the Programming Language Design and Implementation*, pages 718–730.