A Lightweight Technique to Identify Equivalent 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 mutants are useless and contribute to increase costs. We propose a lightweight technique to identify equivalent mutants by proving equivalences with Z3 in the context of weak mutation testing. To evaluate our approach, we apply our technique for 40 mutation targets (mutations of an expression or statement) and automatically identify 13 equivalent mutations for seven mutation targets. We manually confirm that the equivalent mutants detected by our technique are indeed equivalent. Moreover, we evaluate our approach in the context of strong mutation testing against mutants generated by* MuJava *for 5 projects. Our technique detects all equivalent mutants detected by TCE. The results of our technique can be useful to improve mutation testing tools by avoiding the application of 13 mutations for 7 mutation targets.*

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 and do not contribute to the test assessment process because they have the same behavior as the original program [Budd and Angluin 1982, Jia and Harman 2011, Madeyski et al. 2014]. So, these mutants are useless [Fernandes et al. 2017]. Listing 1 illustrates one equivalent mutant. The mutation operator applied introduces a post decrement to a local variable (`value--`). Notice that this introduction does not change the behavior when compared to the original program, since the decrement would happen after the function returns and `value` is a local variable. In this sense, M_1 is useless. 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]).

Listing 1. Original program and equivalent mutant.

```
// Original program - O
public long function(long value) {
    return value;
}

// Equivalent Mutant - M1
public long function(long value) {
    return value--;
}
```

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 four steps to discover equivalent mutants using theorem proving in the context of weak mutation testing [Howden 1982]. We encode a theory of equivalence in Z3 and use its theorem prover [de Moura and Bjørner 2008] to automatically identify equivalent mutants (Section 2). Our technique is lightweight, we do not need to create mutants, compile them, create test suites, and execute them, as previous works [Fernandes et al. 2017]. Our technique automatically identifies and prove 13 equivalences for seven mutation targets. Then, we manually check and confirm that the equivalent mutants detected by our approach are indeed equivalent.

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 can use our results to improve mutation testing tools. For example, our technique may be useful to reduce costs of mutation testing by avoiding the application of 13 mutations for 7 mutation targets.

We organize this paper as follows. Section 2 describes our lightweight technique to identify equivalent mutants using Z3. Section 3 presents the evaluation of our approach. Finally, we relate our technique to others (Section 4), and present concluding remarks (Section 5).

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 mutants using weak mutation testing. 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 equivalent mutants.

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.

Listing 2. 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
```

We encode a theory in Z3 to detect equivalent mutants, in Step 4. We use the latest version of Z3 after fixing the bugs found by Winterer et al. [Winterer et al. 2020]. Listing 2 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.

The `identifyEquivalentMutants` function presented in Listing 3 returns equivalent mutants of a program.

Listing 3. Identifying equivalent mutants in Z3.

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

The mutations created by MuJava are First Order Mutations (FOMs). We also encode and prove equivalence 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 [Jia and Harman 2009].

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 4. 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 4).

Listing 4. 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?
- **RQ$_2$** To what extent does our approach work in the context of strong mutation testing?

The artifacts used to answer these questions and the results that we obtained are available in our Google Colaboratory notebook [Artifacts 2020].

3.1. RQ$_1$: Number of Equivalent Mutants

We evaluate our technique 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. The mutations in red are HOMs.

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. Table 2 describes the mutation operators considered in our work [Ma and Offutt].

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 (see Table 1). Eleven of the equivalent mutants found are FOMs. For the mutation targets `exp++` and `exp--`, our approach classified the following mutations as equivalent:

`AORS(exp)`, `AODS(exp)`, and `ODL(exp)`. We encoded HOMs for three mutation targets. We found two equivalent mutants which are HOMs. For the `lexp == rexp` mutation target the `COI ROR !(!=)` mutation always yield equivalent mutants, and for the `lexp != rexp` mutation target the `COI ROR !(==)` mutation is equivalent. We manually analyze whether the equivalent mutants detected by our technique are indeed equivalent. We do not find false positives.

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

3.2. RQ_2: Strong Mutation Testing

In this section, our goal is to check whether our results hold in the context of strong mutation testing. We use *TCE* [Kintis et al. 2017], which is a sound tool that applies compiler optimizations and checks the bytecode of the original program and the mutant to find equivalence. A human-based equivalence verification reveals that *TCE* has the ability to detect approximately 30% of all the existing equivalent mutants [Kintis et al. 2017]. *TCE* is one of the best static analysis tools to detect some types of equivalent mutants.

We apply MUJAVA to 5 real projects, and generate mutants for 20 mutation targets. MUJAVA generates 5,297 mutants. 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. Table 3 also presents the amount of mutants created by MUJAVA for each program.

Table 3. Five programs used in our evaluation.

Project	Version	LOC	MuJava
joda-time	2.10.1	28,790	2,652
commons-math	3.6.1	100,364	651
commons-lang	3.6	27,267	1,126
h2	1.4.199	134,234	210
javassist	3.20	35,249	658

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 (see Table 1).

For the following 19 mutation targets, neither TCE nor our approach find any equivalent mutants: `lexp + rexp`, `lexp - rexp`, `lexp * rexp`, `lexp && rexp`, `lexp || rexp`, `lexp == rexp (bool)`, `lexp != rexp (bool)`, `lexp == rexp`, `lexp != rexp`, `lexp > rexp`, `lexp >= rexp`, `lexp < rexp`, `lexp <= rexp`, `lexp != rexp (obj)`, `lexp & rexp`, `lexp | rexp`, `lexp ^ rexp`, `lexp << rexp`, `--exp`.

MuJava does not generate HOMs. So, it does not generate the equivalent mutants using the HOMs `COI ROR !(==)` and `COI ROR !(!=)` for the mutation targets `lexp == rexp` and `lexp != rexp`. We also tried to evaluate our approach to more mutation targets, such as `exp++`. However, MuJava did not generate mutants for them. This way, we did not detect equivalent mutants for the following mutation targets with *TCE*, as we did with our approach (see Table 1): `+exp`, `exp++`, `exp--`. In summary, all mutants detected by *TCE* are also detected by our technique.

4. Related Work

Addressing the equivalent mutant is not a recent problem [Jia and Harman 2011]. Previous studies have been addressing this problem and surveys on this topic have been published [Jia and Harman 2011, Madeyski et al. 2014, Pizzoleto et al. 2019]. Budd and Angluin [Budd and Angluin 1982] show that detecting an equivalent mutant is an undecidable problem. In our work, we show that for some mutation operators, Z3 automatically proves a theorem and yields an answer.

To tackle the mutation equivalent problem, some studies used compiler optimizations [Baldwin and Sayward 1979]. The intuition is that code optimization can transform the original program and the mutant in a way which their compiled object codes will be identical. Kintis et al. [Kintis et al. 2017] propose the Trivial Compiler Equivalence (*TCE*) and used this idea in popular languages (C and Java) and mutation tools (Milu and MuJava). Using this strategy, the original program and the mutant need to be compiled, and some equivalent mutants cannot be detected.

Kintis and Malevris [Kintis and Malevris 2015] use static analysis to avoid some equivalent mutants by introducing data-flow patterns. They indicate that a number of equivalent mutants can be avoided by just analyzing the original program under test.

Offutt and Pan [Offutt and Pan 1997] use constraint solvers to detect equivalent mutants and infeasible paths automatically. Harman et al. [Harman et al. 2001] use the program slicing technique to assist in the detection of equivalent mutants. Fernandes et al. [Fernandes et al. 2017] propose a strategy to help with identifying rules to avoid equivalent mutants. Shuller and Zeller [Schuler and Zeller 2013] and Grun et al. [Grün et al. 2009] indicate that changes in coverage can be used to detect non-equivalent mutants. In our work, we use the Z3 theorem prover to detect equivalent mutants.

5. Conclusion

In this work, we propose a lightweight technique to detect equivalent mutants using Z3. Developers only need to specify the types and mutations in our encoding to identify equivalent mutants (see Listing 4). We do not need to create mutants, compile them, create test suites, and execute them, as previous works [Fernandes et al. 2017]. We apply our approach to 40 method-level mutation targets where MUJAVA [Ma et al. 2005], can apply mutations. We also prove equivalence among Higher Order Mutations (HOMs), by combining the FOMs created by MUJAVA. Our technique automatically identifies and prove 13 equivalences for seven mutation targets. Then, we manually check and confirm that the equivalent mutants detected by our approach are indeed equivalent. Moreover, to evaluate our approach in the context of strong mutation testing, we compare our approach to *(TCE)* [Kintis et al. 2017]. To perform the comparison, we apply MUJAVA to 20 mutation targets in 5 real large projects. Then, we execute *TCE* against the mutants generated by MUJAVA. Our technique detects all equivalent mutants detected by TCE, but with less effort.

Our approach may be useful to reduce costs of mutation testing. We recommend developers to avoid generating mutants for some mutation operators applied to mutation targets presented in Table 1. In this way, developers will avoid generating mutants, compiling them, and then running *TCE* to detect equivalent mutants. Moreover, *TCE* can only identify equivalent mutants that have the same bytecode. However, there are cases in which we may have equivalent mutants that have different bytecodes, and *TCE* will not be able to detect [Fernandes et al. 2017].

As future work, we intend to identify more equivalent mutants by encoding more FOMs and HOMs. We also aim at improving our technique to detect equivalent mutants created by other mutation operators (i.e. class-level mutation operators), by proving more complex assertions. Finally, we also intend to extend MUJAVA to avoid generating the equivalent mutants detected by our technique.

Note: The first author helped to encode and prove equivalent mutants 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.

Acknowledgments

We thank the anonymous reviewers for their valuable comments. This work was partially supported by CNPq and CAPES grants.

References

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

Artifacts (2020). A technique to identify equivalent mutants. At https://colab.research.google.com/drive/1ts_2m0tW0GLWsVJ_qhC59V9l8rVPdtsa?usp=sharing.

Baldwin, D. and Sayward, F. (1979). Heuristics for determining equivalence of program mutations. Technical report, DTIC Document.

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.

Grün, B. J., Schuler, D., and Zeller, A. (2009). The impact of equivalent mutants. In *Proceedings of the International Conference on Software Testing, Verification, and Validation Workshops*, pages 192–199.

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*, page To appear. IEEE.

Harman, M., Hierons, R., and Danicic, S. (2001). The relationship between program dependence and mutation analysis. In *Proceedings of the Mutation Testing for the New Century*, pages 5–13.

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.

Kintis, M. and Malevris, N. (2015). MEDIC: A static analysis framework for equivalent mutant identification. *Information and Software Technology*, 68:1–17.

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.

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

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.

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.

Offutt, J. and Pan, J. (1997). Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability*, 7(3):165–192.

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.

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.

Pizzoleto, A., Ferrari, F., Offutt, J., Fernandes, L., and Ribeiro, M. (2019). A systematic literature review of techniques and metrics to reduce the cost of mutation testing. *Journal of Systems and Software*, 157.

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.