securityandtrust.lu

Fault-based Automated Quality Assurance of Test Suites: Practical Mutation Testing for Satellite and Embedded Systems

Interdisciplinary Centre for Security, Reliability and Trust

University of Luxembourg

November 25, 2019

UNIVERSITÉ DU LUXEMBOURG

Chapter 1

Code-driven Mutation Testing

1.1 Mutation Testing Process

Figure 1.1 shows the reference code-driven mutation testing process that will be considered in this book. The process depicted in Figure 1.1 has been inspired by the mutation testing process described in related work [1, 2]. The process is based on two main sub-processes, *Test Suite Evaluation* and *Test Suite Augmentation*. Sections 1.1.1 and provide an overview of each of them, Sections 1.1.1 to 1.1.2 describe in details the building blocks and the issues to overcome in order to efficiently apply code oriented data mutation.

1.1.1 Test Suite Evaluation

The Test Suite Evaluation process concern the automatic generation of modified versions (i.e., the mutants) of the software under test (SUT) and the evaluation of the quality of the SUT test suite. It consists of three activities: *create mutants*, *execute mutants*, and *analyze results*. These activities are typically automated by toolsets that often include strategies to address scalability issues. Mutation testing activities and related optimizations are depicted in Figure 1.1 and described in the following paragraphs.

The Test Suite Evaluation process starts with engineers providing the SUT and a selection of the mutation operators to consider. The set of mutation of available mutation operators typically depends on the toolset implementing the mutation testing process. A typical set of mutation operators implemented by most of the existing toolsets consists of the relational (ROR), logical (LCR), arithmetic (AOR), absolute (ABS) and unary insertion (UOI) operators [3]. Section 1.2 provides an overview of the mutation operators defined in the literature that can be applied in the context of space and embedded systems.

In the mutation testing process, the activity *create mutants* concerns the application of the mutation operators to the source code of the SUT; it leads to the generation of modified versions of the SUT (i.e., the mutants) that should be compiled and then executed against the test suite to evaluate the test suite quality.

Unfortunately, the mutation process leads to a high number of mutants to be generated, which leads to scalability issues due to the time required to compile and

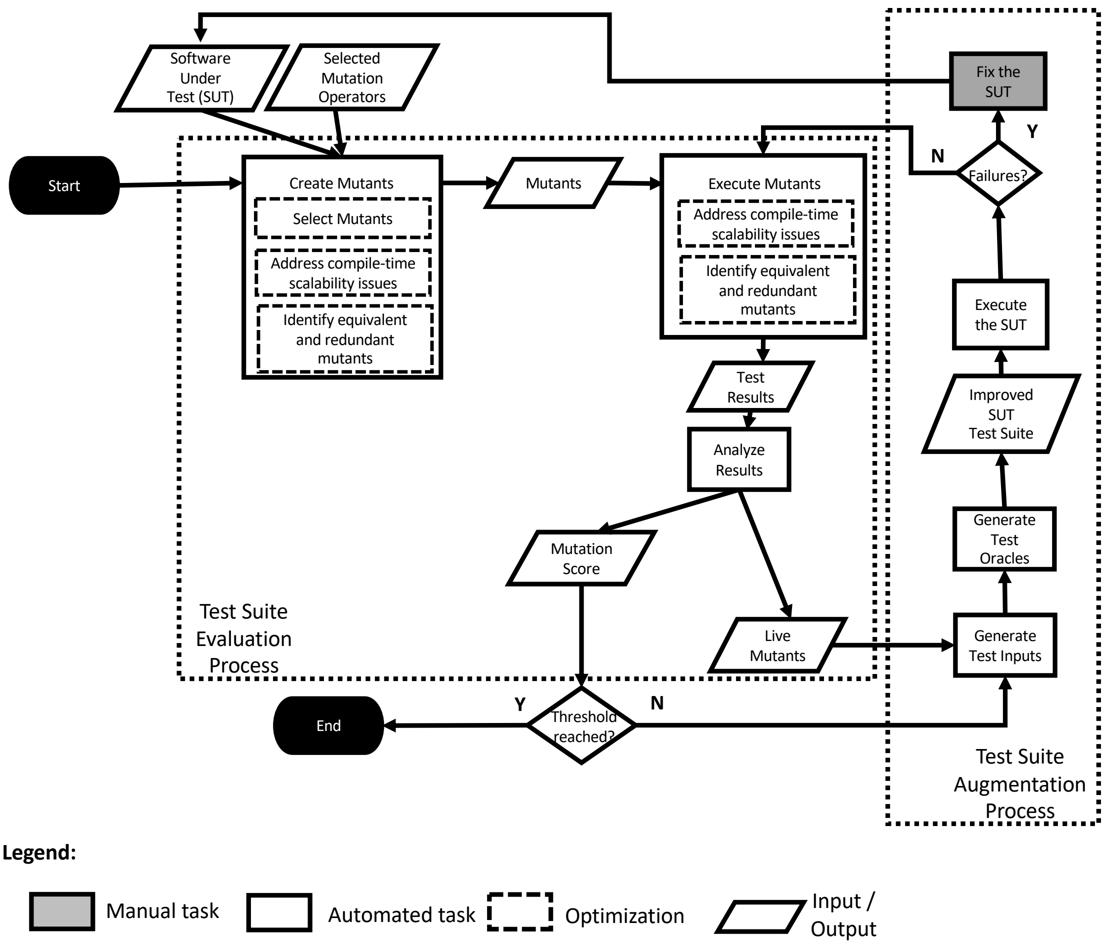

Figure 1.1: Mutation Testing Process.

execute the different versions generated. Recent surveys provide an overview of existing optimization techniques [4]; the most relevant optimization approaches target the compilation processes, the execution of mutants, and mutants selection.

Two main mutant selection approaches have been defined: the selection of mutation operators and the random selection of mutants [5]. The first approach consists of the empirical identification of a subset of mutation operators that is sufficient to predict the mutation score [6, 7]. The second approach consists of randomly selecting a certain percentage of mutants from the generated ones [8], possibly with a uniform distribution of the different mutation operators [5]. Empirical results with academic case studies [5] show that the first approach is not superior to random selection when selecting the same number of mutants. Other work [9] show that the combination of operator-based selection and random sampling leads to better results since it leads to high mutation score (above 98%) while reducing the average mutation testing time to 6.54%. The use of higher-order mutants is another solution to reduce the overall number of mutants. Other optimizations are framework specific, for example Mull limits mutations to reachable code [10]. Section **??** provides an overview of mutant selection approaches.

To reduce the time spent in the compilation of the generated mutants, mutant schemata [11] are adopted to encode all the mutants in a single file and parametrize the mutant execution so that mutants are compiled in a single pass and selected at runtime. Section 1.3.1 provides an overview of compile-time optimization approaches.

Another optimization concerns the identification of equivalent and redundant mutants. Equivalent mutants are mutants that behave as the original program, while redundant mutants are mutants that lead to the same test failures. To detect if a program and one of its mutants are equivalent is undecidable [12]; however, heuristics to partially address the problems have been defined in the literature. Trivial compiler optimization might be adopted to detect both equivalent and redundant mutants; it relies on the idea that source code that leads to the same program behaviour often belongs to the same optimized compiled code [13]. Other approaches concern the adoption of symbolic execution [14, 15] and the runtime monitoring of the SUT (e.g., mutants that lead to the same execution paths are likely equivalent [16]). Finally, Shin et al. proposed the *distinguishing mutation adequacy criterion*, which aims to ensure that the test suite includes enough different test inputs so that every mutant is distinguished by each other, if feasible [17]. Solutions to address the problem of identifying equivalent and redundant mutants are detailed in Sections 1.3.3 and 1.3.4, respectively.

The execution of mutants implies the execution of the test suite of SUT against all the generated mutants. Optimizations for the execution of mutants concern the scalability of the mutants execution process and the identification at run-time of equivalent and redundant mutants. A well-known optimization concerning the scalability of the mutants execution process is the split-stream optimization, which consists of generating a modified version of the SUT that creates multiple processes (one for each mutant) only when the mutated code is reached [18]. With split-stream, the code shared among multiple mutants is executed only once thus saving time and resources. Other execution optimizations consist of minimizing the number of processes being created by sharing one single process among mutants that bring the system into the same state [19]. Section 1.3.2 provides an overview of techniques to address run-time scalability issues.

The analysis of test results concerns the identification of mutants that lead to the failure of at least one test case of the SUT test suite; these mutants are said to be *killed*. Mutants that do not lead to the failure of any test case are said to be *live mutants*. The identification of killed and live mutants enable the definition of a mutation adequacy criterion as follow, *a test suite is mutation-adequate if all mutants are killed by at least one test of the test suite*. Also, the percentage of killed mutants is used to quantitatively measure the quality of a test suite. This measure is referred to as *mutation score*. Because of equivalent and redundant mutants, mutation-adequacy is difficult to achieve while the mutation score might not be representative of test suites quality [20]. Section 1.3.5 provides an overview of solutions addressing the problems related to the computation of the mutation score.

The capability of a test case to kill a mutant often depends on the observability of the system state. To overcome the limitations due to observability, different strategies for distinguishing program executions (i.e, to determine if the execution of two test cases led to different results) have been defined. These strategies are known as strong, weak, firm, and flexible mutation testing. Strong mutation coverage indicate that the computation of the mutation score is based on the percentage of mutants identified by test failures, i.e., based on difference between the expected and the observed output of the system. Weak mutation coverage consists of verifying if the state of the system has been altered, with respect to the original code, after the execution of the mutated statement. Firm mutation coverage consists of verifying if the change in the state of the system propagates after the mutated code, e.g., at function boundaries. Flexible weak mutation consists of checking if the mutated code leads to object corruption [21]. The main difference among these four coverage strategies is that only strong mutation coverage enables engineers to assess the quality of test cases in their entirety, i.e., by evaluating both the capability of triggering an erroneous behavior and the capability of reporting the erroneous behaviour thanks to complete test oracles. The other strategies only evaluate the capability of the test suites of triggering the erroneous behavior.

1.1.2 Test Suite Augmentation

The Test Suite Augmentation process concerns the definition of test cases that kill live mutants. Although the test suite can be augmented manually by engineers after inspecting the source code of the live mutants, we focus on the possibility of automating such process. The Test Suite Augmentation process consists of four activities Identify Test Inputs, Generate Test Oracles, Execute the SUT, Fix the SUT. The first two activities lead to the definition of new complete test cases, the execution of the SUT enable engineers to determine if the newly defined test cases spot faults not identified by the original test suite, which is one of the benefit of mutation testing. Finally, fixing the SUT is performed manually by the engineer in case of test failures.

Concerning the automated generation of test cases for mutated C programs, existing work investigated the adoption of the KLEE symbolic execution engine [23] and the use of bounded model checking [24]. Other work combines dynamic symbolic execution (DSE) with search-based software testing (SBST) to generate test inputs that lead to strong mutations [25].

Concerning the generation of test oracles, a state-of-the-art approach consists of the generation of assertions that verify the value of variables that enable the killing of mutants [26]. Such approach has been adopted in the context of Java programs but not for C or embedded systems.

If test failures are not observed, engineers evaluate the quality of the newly generated test suite by observing the mutation score and inspecting live mutants.

Section 1.4 provides an overview of approaches for the automated generation of test cases.

1.2 Mutation Operators

Mutation testing introduces small syntactical changes into the code of a program (source code, intermediate representation, or executable code) through a set of mutation operators. Mutation operators have two goals, (1) induce simple syntax changes based on errors that programmers typically make, and (2) force common testing goals (e.g., code coverage testing).

Table XX provides an overview of existing mutation operators that can be applied to space software. We selected operator for programming languages likely adopted in the context of space and embedded systems. These include operators dedicated to the C and C++ programming language, operators targeting Low-Level Intermediate Representation (LLVM-IR), operators dedicated to the ADA programming language, operators for the SQL and Simulink languages, operators that can be applied to any Object Oriented programming language but for which an implementation dedicated to C++ is not available. The last six columns of Table XX indicate the programming language targeted by the operator; when an operator support one of the specified programming languages we report a reference to the tool implementing the operator or, in case an implementation is not available, the paper in which the operator has been described.

The operators appearing in Table XX are organized in XX categories: ADA tasks, arithmetic, bitwise, casts, constants, control-flow, coverage, deletion, ADA expressions, floating points, function calls, logical, memory operations, object-oriented, ADA operands, relational, variable references, shift, SQL, statements, strings, structures and system-level operators. In the following, we provide an overview of each category:

- Category *ADA expressions* includes operators that work by introducing mutations at an expression-level granularity in the ADA programming language.
- Category *ADA operands* includes operators that perform mutations by replacing operands in ADA expressions (e.g., variables, constants, records, arrays, pointers).

- Category *ADA tasks* includes operators that mutate ADA code at the granularity of tasks and aim to trigger problems related to concurrency in the program.
- Category *arithmetic* includes operators that performs mutations on arithmetic expressions and assignments, usually these mutations replaces an element by an operator $op \in \{+, -, *, /, \%\}$.
- Category *bitwise* includes operators that performs bitwise mutations on expressions and assignments, these mutations aims for replacing an element of an expression by an operator $op \in \{|, \&, \hat{}, \sim\}$.
- Category *casts* includes operators that work by mutating casting expressions.
- Category *constants* includes operators that mutate constants in the code.
- Category *control-flow* includes operators that modify and alter the execution flow of the running application.
- Category *coverage* includes operators that mutate the code in such a way that the static structure of the code (e.g., branches) is likely covered during execution.
- Category *deletion* includes operators that mutate the code by deleting statements, operators, constants and variables.
- Category *floating points* includes operators working on floating point comparison (FPC).
- Category *function calls* includes operators that mutate function calls and their components (e.g., signature and parameters).
- Category *logical* includes operators that performs logical mutations on expressions and assignments; these mutations replace an element of an expression by an operator $op \in \{\&\&, ||, !\}$.
- Category *memory operations* includes operators seeking memory issues such as buffer overflow vulnerabilities, uninitialized memory access, NULL point dereferencing, and memory leaks caused by faulty heap management.
- Category *object-oriented* includes operators related to object oriented programming languages. These mutation operators aim to simulate problems concerning encapsulation, inheritance, method overloading, object and member replacement, polymorphism and language-specific features (e.g., Java and C++).
- Category *relational* includes a set of operators that perform relational mutations on expressions and assignments; usually these mutations replace an element by an operator $op \in \{<, <=, ==, !=, >, >=\}$.
- Category *variable references* includes operators mutating different types of variable references in the code (e.g., scalars, arrays and pointers).

- Category *shift* includes operators that performs shift mutations on expressions and assignments; usually, these mutations replace an element by an operator $op \in \{<<, <<<, >>, >>>\}$.
- Category SQL includes operators that include mutations on SQL clauses, SQL `NULL` operators, and SQL `WHERE` operators.
- Category *statements* includes operators performing syntactical changes at a statement-level granularity.
- Category *strings* includes operators that performs mutations on variables of string type.
- Category *structures* includes operators that mutate C/C++ structures.
- Category *system-level* includes operators that performs mutations at a system-level; usually, these mutation operators aim to cause faults concerning the integration of system components or the configuration of the system itself.

1.3 Limitations of Mutation Testing

1.3.1 Compile-time Scalability

1.3.2 Run-time Scalability

1.3.3 Equivalent Mutants

1.3.4 Redundant Mutants

1.3.5 Mutation Score Calculation

1.4 Automated Augmentation of Test Suites

Chapter 2

Data Mutation Testing

Bibliography

[1] A. J. Offutt and R. H. Untch, "Mutation 2000: Uniting the orthogonal," in *Mutation testing for the new century*. Springer, 2001, pp. 34–44.

[2] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: an analysis and survey," in *Advances in Computers*. Elsevier, 2019, vol. 112, pp. 275–378.

[3] G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators a. jefferson offutt ammei lee george mason university," *ACM Transactions on software engineering methodology*, vol. 5, no. 2, pp. 99–118, 1996.

[4] F. C. Ferrari, A. V. Pizzoleto, and J. Offutt, "A systematic review of cost reduction techniques for mutation testing: Preliminary results," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2018, pp. 1–10.

[5] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei, "Is operator-based mutant selection superior to random mutant selection?" in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 435–444.

[6] A. Siami Namin, J. H. Andrews, and D. J. Murdoch, "Sufficient mutation operators for measuring test effectiveness," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 351–360.

[7] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi, "Toward the determination of sufficient mutant operators for c," *Software Testing, Verification and Reliability*, vol. 11, no. 2, pp. 113–136, 2001.

[8] W. E. Wong and A. P. Mathur, "Reducing the cost of mutation testing: An empirical study," *Journal of Systems and Software*, vol. 31, no. 3, pp. 185–196, 1995.

[9] L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid, "Operator-based and random mutant selection: Better together," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2013, pp. 92–102.

[10] F. Hariri and A. Shi, "Srciror: a toolset for mutation testing of c source code and llvm intermediate representation." in *ASE*, 2018, pp. 860–863.

[11] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," in *ACM SIGSOFT Software Engineering Notes*, vol. 18, no. 3. ACM, 1993, pp. 139–148.

[12] T. A. Budd and D. Angluin, "Two notions of correctness and their relation to testing," *Acta Inf.*, vol. 18, no. 1, pp. 31–45, Mar. 1982. [Online]. Available: http://dx.doi.org/10.1007/BF00625279

[13] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 936–946.

[14] M. Papadakis and N. Malevris, "Mutation based test case generation via a path selection strategy," *Information and Software Technology*, vol. 54, no. 9, pp. 915–932, 2012.

[15] B. Kurtz, P. Ammann, and J. Offutt, "Static analysis of mutant subsumption," in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2015, pp. 1–10.

[16] D. Schuler and A. Zeller, "Covering and uncovering equivalent mutants," *Software Testing, Verification and Reliability*, vol. 23, no. 5, pp. 353–374, 2013.

[17] D. Shin, S. Yoo, and D. Bae, "A theoretical and empirical study of diversity-aware mutation adequacy criterion," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 914–931, Oct 2018.

[18] S. Tokumoto, H. Yoshida, K. Sakamoto, and S. Honiden, "Muvm: Higher order mutation analysis virtual machine for c," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2016, pp. 320–329.

[19] B. Wang, Y. Xiong, Y. Shi, L. Zhang, and D. Hao, "Faster mutation analysis via equivalence modulo states," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 295–306.

[20] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. Le Traon, "Threats to the validity of mutation-based test assessment," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 354–365. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931040

[21] P. R. Mateo, M. P. Usaola, and J. L. F. Aleman, "Validating second-order mutation at system level," *IEEE Transactions on Software Engineering*, vol. 39, no. 4, pp. 570–587, 2012.

[22] A. S. Namin, X. Xue, O. Rosas, and P. Sharma, "Muranker: a mutant ranking tool," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 572–604, 2015.

[23] D. Holling, S. Banescu, M. Probst, A. Petrovska, and A. Pretschner, "Nequivack: Assessing mutation score confidence," in *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2016, pp. 152–161.

[24] H. Riener, R. Bloem, and G. Fey, "Test case generation from mutants using model checking techniques," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2011, pp. 388–397.

[25] M. Harman, Y. Jia, and W. B. Langdon, "Strong higher order mutation-based test data generation," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 212–222.

[26] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Transactions on Software Engineering*, vol. 38, no. 2, pp. 278–292, 2011.

[27] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Jozala, "Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation," *IEEE Transactions on Software Engineering*, vol. 40, no. 1, pp. 23–42, 2013.

[28] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon, and M. Harman, "Detecting trivial mutant equivalences via compiler optimisations," *IEEE Transactions on Software Engineering*, vol. 44, no. 4, pp. 308–333, 2017.

[29] D. Baldwin and F. Sayward, "Heuristics for determining equivalence of program mutations." GEORGIA INST OF TECH ATLANTA SCHOOL OF INFORMATION AND COMPUTER SCIENCE, Tech. Rep., 1979.

[30] A. J. Offutt and W. M. Craft, "Using compiler optimization techniques to detect equivalent mutants," *Software Testing, Verification and Reliability*, vol. 4, no. 3, pp. 131–154, 1994.

[31] A. J. Offutt and J. Pan, "Detecting equivalent mutants and the feasible path problem," in *Proceedings of 11th Annual Conference on Computer Assurance. COMPASS'96*. IEEE, 1996, pp. 224–236.

[32] ——, "Automatically detecting equivalent mutants and infeasible paths," *Software testing, verification and reliability*, vol. 7, no. 3, pp. 165–192, 1997.

[33] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[34] J. M. Voas and G. McGraw, *Software fault injection: inoculating programs against errors*. John Wiley & Sons, Inc., 1997.

[35] R. Hierons, M. Harman, and S. Danicic, "Using program slicing to assist in the detection of equivalent mutants," *Software Testing, Verification and Reliability*, vol. 9, no. 4, pp. 233–262, 1999.

[36] M. Harman, R. Hierons, and S. Danicic, "The relationship between program dependence and mutation analysis," in *Mutation testing for the new century*. Springer, 2001, pp. 5–13.

[37] M. Ellims, D. Ince, and M. Petre, "The csaw c mutation tool: Initial results," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 2007, pp. 185–192.

[38] E. Martin and T. Xie, "A fault model and mutation testing of access control policies," in *Proceedings of the 16th international conference on World Wide Web*. ACM, 2007, pp. 667–676.

[39] L. du Bousquet and M. Delaunay, "Towards mutation analysis for lustre programs," *Electronic Notes in Theoretical Computer Science*, vol. 203, no. 4, pp. 35–48, 2008.

[40] E. S. Mresa and L. Bottaci, "Efficiency of mutation operators and selective mutation strategies: An empirical study," *Software Testing, Verification and Reliability*, vol. 9, no. 4, pp. 205–232, 1999.

[41] M. Papadakis and N. Malevris, "An empirical evaluation of the first and second order mutation testing strategies," in *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, 2010, pp. 90–99.

[42] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 402–411.

[43] M. E. Delamaro, J. Offutt, and P. Ammann, "Designing deletion mutation operators," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 2014, pp. 11–20.

[44] K. Adamopoulos, M. Harman, and R. M. Hierons, "How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution," in *Genetic and evolutionary computation conference*. Springer, 2004, pp. 1338–1349.

[45] J. Offutt, Y.-S. Ma, and Y.-R. Kwon, "The class-level mutants of mujava," in *Proceedings of the 2006 international workshop on Automation of software test*. ACM, 2006, pp. 78–84.

[46] G. K. Kaminski and P. Ammann, "Using a fault hierarchy to improve the efficiency of dnf logic mutation testing," in *2009 International Conference on Software Testing Verification and Validation*. IEEE, 2009, pp. 386–395.

[47] B. J. Grün, D. Schuler, and A. Zeller, "The impact of equivalent mutants," in *2009 International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, 2009, pp. 192–199.

[48] Y. Jia and M. Harman, "Higher order mutation testing," *Information and Software Technology*, vol. 51, no. 10, pp. 1379–1393, 2009.

[49] M. Kintis, M. Papadakis, and N. Malevris, "Evaluating mutation testing alternatives: A collateral experiment," in *2010 Asia Pacific Software Engineering Conference*. IEEE, 2010, pp. 300–309.

[50] A. J. Offutt, "Investigations of the software testing coupling effect," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 1, no. 1, pp. 5–20, 1992.

[51] M. Polo, M. Piattini, and I. García-Rodríguez, "Decreasing the cost of mutation testing with second-order mutants," *Software Testing, Verification and Reliability*, vol. 19, no. 2, pp. 111–131, 2009.

[52] J. C. Maldonado, M. E. Delamaro, and R. A. F. Romero, "Bayesian-learning based guidelines to determine equivalent mutants," *Machine Learning Applications in Software Engineering*, vol. 16, no. 6, p. 150, 2005.

[53] D. Schuler, V. Dallmeier, and A. Zeller, "Efficient mutation testing by checking invariant violations," in *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 2009, pp. 69–80.

[54] D. Schuler and A. Zeller, "(un-) covering equivalent mutants," in *2010 Third International Conference on Software Testing, Verification and Validation*. IEEE, 2010, pp. 45–54.

[55] M. Kintis and N. Malevris, "Identifying more equivalent mutants via code similarity," in *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, vol. 1. IEEE, 2013, pp. 180–188.

[56] ——, "Using data flow patterns for equivalent mutant detection," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2014, pp. 196–205.

[57] ——, "Medic: A static analysis framework for equivalent mutant identification," *Information and Software Technology*, vol. 68, pp. 1–17, 2015.

[58] A. T. Acree Jr, "On mutation." GEORGIA INST OF TECH ATLANTA SCHOOL OF INFORMATION AND COMPUTER SCIENCE, Tech. Rep., 1980.

[59] S. Nica and F. Wotawa, "Using constraints for equivalent mutant detection," *arXiv preprint arXiv:1207.2234*, 2012.

[60] M. Kintis, M. Papadakis, and N. Malevris, "Isolating first order equivalent mutants via second order mutation," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 701–710.

[61] ——, "Employing second-order mutation for isolating first-order equivalent mutants," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 508–535, 2015.

[62] M. Papadakis, M. Delamaro, and Y. Le Traon, "Mitigating the effects of equivalent mutants with mutant classification strategies," *Science of Computer Programming*, vol. 95, pp. 298–319, 2014.

[63] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. Le Traon, "Threats to the validity of mutation-based test assessment," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 354–365.

[64] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in linux device drivers," in *Acm sigops operating systems review*, vol. 42, no. 4. ACM, 2008, pp. 247–260.

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

[66] G. Kaminski, P. Ammann, and J. Offutt, "Improving logic-based testing," *Journal of Systems and Software*, vol. 86, no. 8, pp. 2002–2012, 2013.

[67] P. Delgado-Pérez, S. Segura, and I. Medina-Bulo, "Assessment of c++ object-oriented mutation operators: A selective mutation approach," *Software Testing, Verification and Reliability*, vol. 27, no. 4-5, p. e1630, 2017.

[68] S. Anand, C. S. Păsăreanu, and W. Visser, "Jpf–se: A symbolic execution extension to java pathfinder," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2007, pp. 134–138.

[69] M. Papadakis and N. Malevris, "Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing," *Software Quality Journal*, vol. 19, no. 4, p. 691, 2011.

[70] M. Papadakis, N. Malevris, and M. Kallia, "Towards automating the generation of mutation tests," in *Proceedings of the 5th Workshop on Automation of Software Test*. ACM, 2010, pp. 111–118.

[71] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. De Halleux, and H. Mei, "Test generation via dynamic symbolic execution for mutation testing," in *2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–10.

[72] K. Jamrozik, G. Fraser, N. Tillman, and J. De Halleux, "Generating test suites with augmented dynamic symbolic execution," in *International Conference on Tests and Proofs*. Springer, 2013, pp. 152–167.

[73] M. Papadakis and N. Malevris, "Automatic mutation test case generation via dynamic symbolic execution," in *2010 IEEE 21st International Symposium on Software Reliability Engineering*. IEEE, 2010, pp. 121–130.