Rehabilitating equivalent mutants as static anomaly detectors in software artifacts

Paolo Arcaini*, Angelo Gargantini*, Elvinia Riccobene†, Paolo Vavassori*
*Dipartimento di Ingegneria, Università degli Studi di Bergamo, Italy
Email: {paolo.arcaini, angelo.gargantini, paolo.vavassori}@unibg.it
†Dipartimento di Informatica, Università degli Studi di Milano, Italy
Email: elvinia.riccobene@unimi.it

***Abstract*—In mutation analysis a mutant is said *equivalent* if it leaves the semantics of the program or the model unchanged. Equivalent mutants are usually seen as an inconvenience; for example, in software testing they cannot be detected by a test and, therefore, they fictitiously reduce the mutation score of a test suite. In this paper, instead, equivalent mutants are seen as an *opportunity*, since they can be used to find some *static anomalies* of software artifacts, i.e., anomalies that can be removed without affecting the artifact semantics. The proposal is applicable to different kinds of software artifacts as source code, Boolean expressions, and feature models.**

1. Introduction

Mutation analysis has a long history and has been applied to several areas of software engineering [19], mainly to mutation testing [28]. In mutation testing, faults are artificially introduced in the code under test and test cases are used to detect (or *kill*) those faults (*mutants*). Good tests can kill all the injected faults in the program or at least most of them: a test suite has a *mutation score* equal to the portion of mutants it can kill. Other approaches can be used to kill mutants, and mutation analysis can be applied also to other artifacts, not only to program code. For instance, in [2] we used mutation analysis in order to assess the quality of a static quality process for specifications of the NuSMV model checker. In all these cases, the main goal is to kill as many mutants as possible. However, some mutants are impossible to kill: a mutant is said *equivalent* if it leaves the semantics of the program (or model) unchanged. Equivalent mutants are seen as an inconvenience and the equivalent mutant problem is considered one of the main causes why mutation testing is seldom used in practice [32], [28]. For instance, in software testing they cannot be detected by a test and thus they reduce the quality index (mutation score) of a test suite without a real justification. In test generation, equivalent mutants pose a challenge: they consume resources without producing any useful test. For these reasons, several attempts try to eliminate them (e.g., by filtering), or to automatically find and avoid them [24], [15].

. 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)
10th International Workshop on Mutation Analysis (Mutation 2015)
978-1-4799-1885-0/15/$31.00 ©2015 IEEE

In this paper, we argue that mutation analysis can be extended in order to find other defects (anomalies) in software artifacts. The approach we propose leads to a *rehabilitation* of equivalent mutants, which are given a new ability to detect several types of anomalies. Equivalent mutants are seen, therefore, as an *opportunity*, and the long time experience in finding them can be reused in order to discover anomalies.

We define a certain type of *anomalies*, that we call *static*, in terms of equivalent mutants. We show that if, given an artifact A and a quality of A (like readability, efficiency and, so on), we are able to produce an equivalent mutant with better quality than A, then A contains a static anomaly that should be removed. We show that this concept applies to several types of artifacts, for several families of anomalies, and for several types of mutation operators.

Due to space limitation, we assume the reader to be familiar with the concepts of mutation operator, mutation analysis, and equivalent mutants [19].

Sect. 2 introduces the concept of static anomalies and a technique, exploiting equivalent mutants, for discovering them; we show that mutation operators can work as anomaly detectors. Sect. 3 shows the application of the technique to different software artifacts. Sect. 4 presents some threats to the validity of our proposal, and outlines future research directions. Sect. 5 concludes the paper.

2. Detecting Static Anomalies

Software anomalies are defined in the IEEE standard[1] as follows:

Any condition that deviates from the expected based on requirements specifications, design documents, user documents, standards, etc. or from someone's perceptions or experiences. Anomalies may be found during, but not limited to, the review, test, analysis, compilation, or use of software products or applicable documentation.

According to the standard, each artifact should have some quality attributes (like *readability*, *compactness*, *efficiency*, *correctness*, etc.) and an anomaly is any deviation in terms of the expected (quality) attributes. For example, faults represent deviations w.r.t. the expected behavior; dead code is a deviation w.r.t. compactness.

1. IEEE Standard Classification for Software Anomalies (1044-2009)

In this paper, we focus on *static anomalies*, i.e., anomalies that can be removed without changing the "meaning" of the artifact. Static anomalies regard the artifacts structure and they relate to qualities that may be statically measured.

2.1. Using mutation to detect static anomalies

We first define the concept of static anomaly in terms of equivalence and quality of artifacts.

We assume that one can define a quality q over artifacts and that q induces a partial order (of better quality) $>_q$ among all the artifacts, i.e., an artifact may be *better* than another in terms of a certain quality q. Whenever possible, we will define q as a real-valued function over the considered artifacts, such that q induces a total order.

Moreover, we assume that it is possible to check equivalence among artifacts. Intuitively, two artifacts are *equivalent* if they have the same *meaning*. The exact definition of equivalence depends on the type of artifacts.

Given a certain quality q, an artifact may contain a static anomaly in terms of q if the following condition holds:

Definition 1 (**Static anomaly**). *Given an artifact A and its mutation A', if A' is equivalent to A but $A' >_q A$, then A contains a* static anomaly. *The static anomaly is the difference between A' and A.*

Def. 1 gives the foundation of a methodology for defining, finding, and possibly removing static anomalies. Having in mind a software quality, we argue that it is possible to find a mutation operator that detects the static anomaly regarding that quality. Normally the designer starts with a precise static anomaly and the related quality q in mind. Then, he/she tries to find or define a suitable mutation operator. Moreover, a way for checking equivalence between artifacts and for comparing their quality must be devised. At this point, given an artifact A, the designer can build a mutation A', check the equivalence, and compare the quality. If A is equivalent to A' and A' is better than A in terms of q, then a static anomaly has been found. Mutation operators are also able to remove the found anomaly: the mutant A' has no longer the detected anomaly in it.

We propose to use mutation operators as *anomaly detector*s. However, not all the mutation operators are anomaly detectors: those that never produce an equivalent mutant with better quality are not.

Some mutation operators always increase a given quality, but the produced mutant may be non-equivalent, and, therefore, only the equivalence must be checked.

Example 1. Let us consider the Statement Deletion mutation operator (SDL) [13], removing a statement in a program, and the quality *compactness*, defined as $1/\#statements$. Applying SDL to a program always increases the program compactnesses. If the removal of a statement from a program originates an equivalent program, then the original program contains a static anomaly (a useless statement). However, if the removed instruction is live code that affects the program state, the mutant is non-equivalent.

Other mutation operators, instead, always produce equivalent mutants, but they may decrease the quality under consideration, and, therefore, only the quality must be checked.

Example 2. The *rename* refactoring renames variables/methods/classes/... names. Such refactoring always produces equivalent mutants and, with respect to the *readability* quality, may either increase or decrease the readability of an artifact. If the readability increases, the original artifact contains a static anomaly (*low readability*).

In the worst case, the mutation operator may both decrease the quality and produce a non-equivalent mutant, and, therefore, both the equivalence and the quality must be checked.

Example 3. *Instruction reordering*, i.e., swapping the order of two instructions A and B, may produce an equivalent mutant (if the two instructions are *independent*) and improve the *readability* (if B refers to the code preceding A and A does not) or the *efficiency*. If a swapping produces an equivalent mutant with better readability, then the original artifact contains a static anomaly (*low readability*).

3. Static anomalies in software artifacts

To argument our thesis that equivalent mutants may indicate the presence of static anomalies, we provide different classes of artifacts having specific quality measures and mutation operators.

3.1. Source code

The first type of artifacts we consider are programs. We report in this section several static anomalies for source code using Java, although we believe that most of them can occur in the majority of the programming languages.

Mutation operators. For our purposes, we consider **ROR** (one of the five classical operators) and we introduce three more operators:

ROR *Relational Operator Replacement*: It replaces a relational operator with a different one.

SDL *Statement Deletion operator* [13]: It removes an entire statement from the program.

RNM *Rename operator*: It renames a variable/method.

ICM *Inline Constant Mutator*: It mutates inline constants. An inline constant is a literal value assigned to a non-final variable.

Equivalence. Equivalent mutants for source code leave the program's overall semantics unchanged [14]. Finding them is particularly difficult, also because they cannot be caught by any test suite and their number can be high. In [30], the authors found that about 45% of all undetected mutants turned out to be equivalent and being able to find them by hand was very time consuming. There are several attempts to automatize the solution of this problem. For instance, in [25], the authors introduce a tool that uses constraint

```
final boolean bDebug = false;
public void method() {
    if (bDebug) {
        // do something
    }
}
```

Code 1: Unreachable code

```
public int m(int b) {
    int a;
    a = 2;
    a = b;
    return a;
}
```

Code 2: Useless code

```
public int max(int[] val) {
    int r = 0;
    for(int i = 0 ; i < val.length; i++) // use 1 instead
        if (val[i] >= val[r]) // use > instead
            r = i;
    return val[r];
}
```

Code 3: Inefficient code

solving for proving the equivalence. A similar approach is presented in [27]. Other authors use program slicing for solving the equivalent mutant problem [18] or data flow analysis and compiler optimizations [26]. Another technique to detect (non-)equivalent mutants, based on *state infection conditions*, is presented in [20]. In [31], the authors show that Bayesian Learning used in Artificial Intelligence can help the tester to determine the equivalent mutants. In [22], second order mutation is exploited for the same goal. For a complete review of exiting techniques for detecting equivalent mutants we remind to [23].

3.1.1. Source code static anomalies.

Dead code. One static anomaly that can be easily detected is the *dead code*. Dead code is code that either is never executed (unreachable code) or it is executed but its effects are never used in any other computation (useless code). Removing dead code improves the quality *compactness* of the code (without changing the code behavior): it is more compact, more easily maintainable and more suitable for mobile applications. Indeed, in recent years there has been an increasing trend toward the incorporation of computers into a variety of devices where the amount of available memory is limited. This makes desirable trying to reduce the size of applications where possible [11]. The mutation operator SDL can remove dead code: If the code without a statement is equivalent, then it contains fewer lines of code and, therefore, is more compact. Note that SDL may remove side-effect free statements, like logging; in this case, the compactness quality would improve, but other qualities like maintainability would decrease.

Codes 1 and 2 contain dead code: the code inside the conditional statement of Code 1 is never executed, while the first assignment in Code 2 has no visible effects. Applying the SDL mutation operator to the first assignment of Code 2 removes the static anomaly, since it produces an equivalent mutant which is more compact. We checked the previous two codes with two static analysis tools, namely FindBugs and PMD. FindBugs does not detect either anomalies, whereas PMD is able to detect the anomaly in Code 2, classifying it as *DD-Anomaly* (i.e., *a recently defined variable is redefined*). Note that code review would easily find both anomalies, but it would require human effort.

Poor readability. Another quality aspect is code *readability*: it measures how much a text is readable (by a human). The readability of a program is important for its maintainability, and is thus a key factor in overall software quality. Aggarwal et al. claim that source code readability and documentation readability are both critical to the maintainability of a project [1]. Other researchers have noted that the act of reading code is the most time-consuming component of all maintenance activities [12].

Refactoring mutation operators rename variables names, introduce constants, identify a part of code and extract a method or a sub-program for it, etc. They all produce equivalent mutants (by definition) and, if properly used, they can augment the code readability.

Inefficient code. Sometimes a code does not contain dead code, but nevertheless it can be modified in order to improve its *efficiency* (in terms of average number of executed statements). For instance, Code 3 could be modified as shown in the comments and the code would be equally functional but more efficient. In this case, as mutation operators we could use ROR and ICM. Note that both ROR and ICM may produce non-equivalent mutants and weaken the efficiency.

3.2. Boolean expressions

Boolean expressions are those involving Boolean operators like AND, OR, and NOT (denoted by $\wedge$, $\vee$, $\neg$). Boolean expressions frequently occur in complex conditions under which some program code is executed or a specification action is performed. They are frequently used to provide semantics to other formalisms (like feature models [7]). Boolean inputs are explicitly found in models of digital logic circuits: in these cases, the extraction of Boolean expressions is rather straightforward. More often, Boolean inputs derive from abstraction techniques that consist in replacing complex formulae with Boolean predicates. These techniques can be applied to high level specifications in which complex conditions about the state are replaced with atomic predicates. Similar abstraction techniques can be applied to source code, for instance, in order to obtain Boolean programs [6]. We follow the definitions and notations used in [21]: the symbols x_1, x_2, etc. are referred to as *variables*, and an occurrence of a variable in a formula is referred to as a *condition*. For example, the formula $x_1 \wedge x_2 \vee x_1$ contains two variables (x_1 and x_2) and three conditions (two x_1's and one x_2). Often there are some constraints among the variables of a Boolean expression.

Mutation operators. There are 10 classical mutation operators (also known as *fault classes*) for Boolean expressions. The hierarchy of these operators has been studied by [21] and by [29], and was later corrected by [10]. For our purposes, we only introduce two of them (and we use the same Boolean expression $\varphi : x_1 \vee \neg x_2$ to explain them):

MVF *Missing Variable Fault*: An occurrence of a condition is omitted in the expression. For example, x_1 is an MVF mutation of φ.

SA0/SA1 *Stuck-At-0/1 Fault*: An occurrence of a condition is replaced by `false`/`true` in the expression. For example, $x_1 \vee true$ is a SA1 mutation of φ.

Equivalence. Two Boolean expressions are *equivalent* if they assume the same truth values for the same input values. Given a boolean expression φ with the constraints Γ, a mutation φ' is equivalent to φ if $\Gamma \models \varphi \leftrightarrow \varphi'$. Checking equivalence of two Boolean expressions is straightforward by using a SAT (or an SMT) solver.

3.2.1. Boolean expressions static anomalies.

Redundant condition. Some conditions (i.e., occurrences of a variable) in a Boolean expression may be completely *redundant*, with the effect of making the expression more difficult to read and to solve. Redundant conditions are those that can be removed without changing the semantics of the expression. As quality, we consider the *simplicity* of a Boolean expression, defined as $1/\#conditions$. Redundant conditions can be discovered and removed by the mutation operator MVF. Applying the MVF operator always increases the simplicity, but it may produce non-equivalent mutants.

Let us consider the expression $\varphi : x \leq 10 \wedge x \leq 5$ where x is an integer variable. φ can be abstracted in the Boolean specification $a \wedge b$ where a stands for $x \leq 10$ and b stands for $x \leq 5$. Between a and b there is the constraint $b \rightarrow a$. If we apply MVF to the condition $x \leq 10$ of φ, we obtain the equivalent expression $\varphi' : x \leq 5$ having a greater simplicity.

Fixed-value expression. A non-constant expression supposedly changes its value by changing the value of the inputs. A *fixed-value* (but not constant) expression is, on the contrary, an expression that always takes a fixed value (either true or false). As quality, we consider the *coverability* that measures how it is difficult to cover the input space of the expression under test. Coverability can roughly be defined as $1/2^n$ where n is the number of conditions. Fixed-value expressions can be detected by the SA0/SA1 mutation operators. The application of SA0 and SA1 requires to check the equivalence but not the quality, since the coverability is always increased by SA0/SA1.

Let us consider the following Java Boolean expression `x <= Integer.MAX_VALUE` where `x` is an Integer variable. Applying SA1 to the expression produces an equivalent mutant with a better coverability (from $1/2^1$ to $1/2^0$).

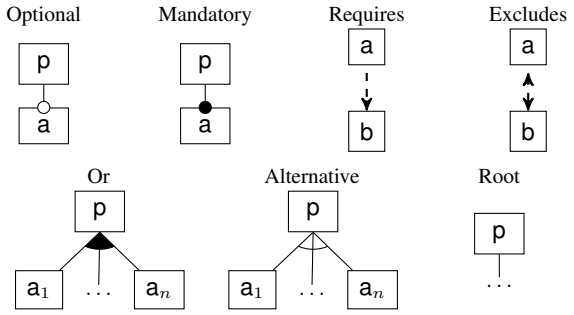


Table 1: Feature models standard notation

3.3. Feature models

In software product line (SPL) engineering, feature models (FMs) are a special type of information model representing all possible products of a SPL in terms of features and relations among them. Specifically, a basic feature model is a hierarchically arranged set of features, where each parent-child relation between them is one of the following types (each having a graphical notation as shown in Table 1):

- *Or*: At least one of the sub-features must be selected if the parent is selected.
- *Alternative* (xor): Exactly one of the sub-features must be selected whenever the parent feature is selected.
- *And*: If the relation between a feature and its sub-features is neither an *Or* nor an *Alternative*, it is called *And*. Each child of an *And* must be either:
  - *Mandatory*: Child feature is required, i.e., it is selected when its respective parent feature is selected.
  - *Optional*: Child feature is optional, i.e., it may or may not be selected if its parent feature is selected.

In addition to the parental relations, it is possible to add *constraints*, i.e., cross-tree relations that specify incompatibility between features:

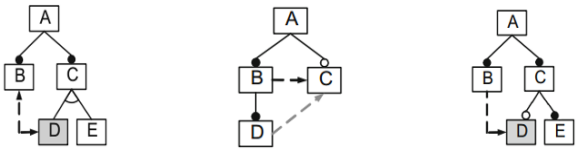
- A *requires* B: The selection of feature A in a product implies the selection of feature B.
- A *excludes* B: A and B cannot be part of the same product.

A *configuration* of a feature model $\mathcal{M}$ is a subset of the features in $\mathcal{M}$ that must include the root. A configuration is *valid* if it respects all the parental relations and the constraints. A valid configuration is called a *product*, since it represents a possible instance of the feature model.

Feature models semantics can be rather simply expressed by using propositional logic [7], [8]. Every feature becomes a propositional letter, and every relation among features becomes a propositional formula modeling the constraints about them. Given a feature model $\mathcal{M}$, we identify with $\text{BOF}(\mathcal{M})$ its representation as propositional formula.

Mutation operators. Mutation analysis has been applied to feature models in [16], [17]. In [4], we devised several fault classes and corresponding mutation operators for feature models. In this paper, those used to discover anomalies are:

OTM *Optional To Mandatory*: An optional relation is changed to mandatory.



(a) Dead feature (b) Redundant constraint (c) False optional

Figure 1: Feature models anomalies (in gray)

MF *Missing Feature*: A feature f (except the root) is removed and it is replaced by its sub-features which inherit the same relation f had with its parent. f is replaced by `false` in any constraint containing it.

MC *Missing Constraint*: A constraint is removed.

Equivalence. Two feature models are *equivalent* if they describe the same set of products. A technique for equivalent mutants detection that uses an SMT-solver is described in [4]; it consists in representing a feature model $\mathcal{M}$ and one of its mutants $\mathcal{M}'$ as propositional formulae $\text{BOF}(\mathcal{M})$ and $\text{BOF}(\mathcal{M}')$ and checking their equivalence.

3.3.1. FMs static anomalies.

Dead feature. A feature is *dead* if it is not present in any product of the FM. As quality measure we adopt *liveness*, defined as $(\#Fs-\#DFs)/\#Fs$, where $\#DFs$ is the number of dead features and $\#Fs$ is the number of features. The higher the liveness, the fewer dead features are contained in the FM. We can detect dead features by using the mutant operator MF. Applying MF can diminish the liveness and can produce a non-equivalent mutant when the removed feature is not dead. However, if the removed feature is dead, then the mutant is equivalent and it has a better liveness[2]. In Fig. 1a, feature D is dead, because it can never be selected in any program: removing it from the FM does not modify the set of products but increases the liveness from 4/5 to 1.

Redundant constraint. A constraint is *redundant* when it does not introduce any further restriction or information to the FM. Redundant constraints make the FM more difficult to understand and introduce useless relations between features. We adopt *cyclomatic simplicity* as quality measure, which can be defined as the inverse of the cyclomatic complexity (CC) [5]. CC is the number of distinct cycles and hence it is equivalent to the number of constraints. We can detect redundant constraints using the mutation operator MC; applying MC always increases the quality, but it may produce non-equivalent mutants: removing one constraint increases the cyclomatic simplicity and, if the mutant is equivalent, the removed constraint is redundant. In Fig. 1b, the *requires* constraint between D and C is redundant because it is implied by the *requires* constraint between B and C, and, therefore, it can be safely removed.

2. Even if we do not know the initial number of dead features $\#DFs$, we are sure that removing a dead feature increases the value of liveness from $(\#Fs-\#DFs)/\#Fs$ to $(\#Fs-\#DFs)/(\#Fs-1)$.

False optional. A feature is a *false optional* if it is marked as optional but it is present in all the products of the FM (i.e., it behaves like mandatory). As quality measure, we adopt *solvability*, roughly estimated as $\#MANs/\#Fs$, where $\#MANs$ is the number of mandatory features. An FM with high solvability can be easily solved, i.e., a product can be easily found, since the condition to add a mandatory feature F in a product is simply the presence of F's parent in the product. Applying OTM always increases the solvability, but it may produce non-equivalent mutants. In Fig. 1c, feature D is a false optional because it is selected in all products; turning D to mandatory produces an equivalent mutant with better solvability. Note that removing the anomaly makes the *requires* constraint between B and D redundant. Indeed, removing a static anomaly may expose another static anomaly. In this case, the introduced redundant constraint can be removed as well, without introducing another anomaly.

4. Threats to validity

Our approach is subject to some threats to validity.

First, removing a static anomaly is not always the right choice, since it may be due to a different anomaly. For instance, a feature in an FM is dead because the model is over-constrained: removing it may be not the right solution. In this case, the mutation operator should be used only as detector but not as remover.

Removing a static anomaly can introduce another static anomaly, as in Sect. 3.3.1 for false optionals; in that case, the introduced anomaly can be removed without reintroducing the original anomaly. However, sometimes two qualities may be in conflict, like readability and compactness. In this case, the user should choose which quality is more important.

Equivalence checking may be hard to execute and it may be not automatable. For instance, checking equivalence of source code is in general an undecidable problem (but also checking for an anomaly is undecidable) and, because it is difficult to automate, a time-consuming activity [14]. However, in particular cases, incomplete techniques can be devised (see Sect. 3.1): they do not guarantee to prove equivalence, but they can be used in practice. They can be exploited to find anomalies without false positives. Moreover, for some formal notations (like feature models), equivalence checking is feasible by using tools like SAT/SMT solvers.

Finally, quality may be not formally defined and its measurement may require some human intervention. For instance, readability in general can be judged only by human experts, although some proposals exist to automatize its measurement [9]. Our approach is suitable when the quality between two artifacts can be easily compared.

We cannot exclude that some static anomalies for some artifacts may be not detectable or very difficult to detect by the proposed technique. However, we have shown that our proposal applies to several types of anomalies/qualities and a wide variety of artifacts. We plan to investigate other types of anomalies and artifacts and to see if our approach is still viable. We have already performed some initial experiments with other classes of artifacts. We have started to investigate

anomalies in the context of formal verification (e.g., anomalies for specifications of the NuSMV model checker [2]), and defects of combinatorial testing models [3].

5. Conclusions

Equivalent mutants are usually seen as a drawback in mutation analysis. In this paper, we have shown that they can also be seen as an opportunity, since they may be used to discover static anomalies of software artifacts, i.e., anomalies that can be removed without affecting the artifact semantics. We have shown that our proposal is applicable to different kinds of software artifacts as, for example, source code, Boolean expressions, and feature models.

References

[1] K. Aggarwal, Y. Singh, and J. Chhabra. An integrated measure of software maintainability. In *Reliability and Maintainability Symposium, 2002. Proceedings. Annual*, pages 235–241, 2002.

[2] P. Arcaini, A. Gargantini, and E. Riccobene. Using mutation to assess fault detection capability of model review. *Software Testing, Verification and Reliability*, pages n/a–n/a, 2014.

[3] P. Arcaini, A. Gargantini, and P. Vavassori. Validation of models and tests for constrained combinatorial interaction testing. In *The 3rd International Workshop on Combinatorial Testing (IWCT 2014) - ICST*, pages 98–107, 2014.

[4] P. Arcaini, A. Gargantini, and P. Vavassori. Generating tests for detecting faults in feature models. In *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2015.

[5] E. Bagheri and D. Gasevic. Assessing the maintainability of software product line feature models using structural metrics. *Software Quality Journal*, 19(3):579–612, Sept. 2011.

[6] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 203–213, New York, NY, USA, May 2001. ACM.

[7] D. Batory. Feature models, grammars, and propositional formulas. In H. Obbink and K. Pohl, editors, *Proceedings of the 9th International Conference on Software Product Lines*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer-Verlag, 2005.

[8] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.

[9] R. Buse and W. Weimer. Learning a metric for code readability. *Software Engineering, IEEE Transactions on*, 36(4):546–558, July 2010.

[10] Z. Chen, T. Y. Chen, and B. Xu. A revisit of fault class hierarchies in general boolean specifications. *ACM Trans. Softw. Eng. Methodol.*, 20(3):13:1–13:11, Aug. 2011.

[11] S. K. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.*, 22(2):378–415, Mar. 2000.

[12] L. E. Deimel, Jr. The uses of program reading. *SIGCSE Bull.*, 17(2):5–14, June 1985.

[13] L. Deng, J. Offutt, and N. Li. Empirical evaluation of the statement deletion mutation operator. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 84–93. IEEE, 2013.

[14] B. J. Grun, D. Schuler, and A. Zeller. The impact of equivalent mutants. In *The 4th International Workshop on Mutation Analysis (Mutation 2009) - ICST*, pages 192–199. IEEE, 2009.

[15] M. Harman, R. Hierons, and S. Danicic. Mutation testing for the new century. chapter The Relationship Between Program Dependence and Mutation Analysis, pages 5–13. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[16] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. le Traon. Assessing software product line testing via model-based mutation: An application to similarity testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pages 188–197, March 2013.

[17] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. Le Traon. Towards automated testing and fixing of re-engineered feature models. In *Proc. of the 2013 Int. Conf. on Software Engineering*, ICSE '13, pages 1245–1248, Piscataway, NJ, USA, 2013. IEEE Press.

[18] R. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, 9(4):233–262, Dec. 1999.

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

[20] R. Just, M. D. Ernst, and G. Fraser. Efficient mutation analysis by propagating and partitioning infected execution states. In *Proc. of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 315–326, New York, NY, USA, 2014. ACM.

[21] K. Kapoor and J. P. Bowen. Test conditions for fault classes in Boolean specifications. *ACM Transactions on Software Engineering and Methodology*, 16(3):10, 2007.

[22] M. Kintis, M. Papadakis, and N. Malevris. Isolating first order equivalent mutants via second order mutation. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 701–710, April 2012.

[23] 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. *Software Engineering, IEEE Transactions on*, 40(1):23–42, Jan 2014.

[24] E. S. Mresa and L. Bottaci. Efficiency of mutation operators and selective mutation strategies: an empirical study. *Software Testing, Verification and Reliability*, 9(4):205–232, 1999.

[25] S. Nica and F. Wotawa. Using constraints for equivalent mutant detection. *Electronic Proceedings in Theoretical Computer Science*, 86:1–8, July 2012.

[26] A. J. Offutt and W. M. Craft. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification and Reliability*, 4(3):131–154, 1994.

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

[28] A. J. Offutt and R. H. Untch. Mutation testing for the new century. chapter Mutation 2000: Uniting the Orthogonal, pages 34–44. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[29] V. Okun, P. E. Black, and Y. Yesha. Comparison of fault classes in specification-based testing. *Information and Software Technology*, 46(8):525–533, 2004.

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

[31] A. M. R. Vincenzi, E. Y. Nakagawa, J. C. Maldonado, M. E. Delamaro, and R. A. F. Romero. Bayesian-learning based guidelines to determine equivalent mutants. *International Journal of Software Engineering and Knowledge Engineering*, 12(6):675–689, 2002.

[32] X. Yao, M. Harman, and Y. Jia. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 919–930, New York, NY, USA, 2014. ACM.