A novel use of equivalent mutants for static anomaly detection in software artifacts

Paolo Arcaini[a], Angelo Gargantini[b], Elvinia Riccobene[c], Paolo Vavassori[b]

[a] *Charles University in Prague, Faculty of Mathematics and Physics, Czech Republic*
[b] *Department of Economics and Technology Management, Information Technology and Production, Università degli Studi di Bergamo, Italy*
[c] *Dipartimento di Informatica, Università degli Studi di Milano, Italy*

Abstract

Context: In mutation analysis, a mutant of a software artifact, either a program or a model, is said *equivalent* if it leaves the artifact *meaning* unchanged. Equivalent mutants are usually seen as an inconvenience and they reduce the applicability of mutation analysis.
Objective: Instead, we here claim that equivalent mutants can be useful to define, detect, and remove *static anomalies*, i.e., deficiencies of given *qualities*: If an equivalent mutant has a better quality value than the original artifact, then an anomaly has been found and removed.
Method: We present a process for detecting static anomalies based on mutation, equivalence checking, and quality measurement.
Results: Our proposal and the originating technique are applicable to different kinds of software artifacts. We present anomalies and conduct several experiments in different contexts, at specification, design, and implementation level.
Conclusion: We claim that in mutation analysis a new research direction should be followed, in which equivalent mutants and operators generating them are welcome.

Keywords: Equivalent mutant, static anomaly, quality measure

---

1. Introduction

Mutation has a long history and has been applied to several areas of software engineering [1] and to different kinds of software artifacts, as code and formal specifications [2, 3, 4]. The main application of mutation analysis is

mutation testing [5], in which faults are artificially introduced in the code under test and test cases are used to detect (or *kill*) these 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. However, some mutants are impossible to kill since only *behavioral* faults can be detected by a test: a mutant is said *equivalent* if it leaves the behavior of the program unchanged. Equivalent mutants cannot be detected by a test and thus they reduce the mutation score of a test suite without a real justification. Also in test generation, equivalent mutants pose a challenge: they consume resources without producing any useful test. Equivalent mutants are therefore usually seen as an inconvenience and the equivalent mutant problem is considered as one of the main causes why mutation testing is seldom used in practice [5, 6]. Several attempts have been proposed to eliminate them (e.g., by filtering), or to automatically find and avoid them [7, 8].

Due to the aforementioned problems, equivalent mutants earned a bad reputation. Following our early position paper [9], we aim in this paper at *rehabilitating* equivalent mutants reputation, since we claim that they can be useful to discover *non-behavioral faults*, a category of faults that has not been targeted in mutation analysis so far. When looking for non-behavioral faults, equivalent mutants should be seen as an *opportunity*, and the long time experience in finding them should be reused.

We define a certain type of non-behavioral faults, that we call *static anomalies*, 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. Differently from classical approaches targeting behavioral faults, we aspire to have a lot of equivalent mutants, since they can be used to detect anomalies.

We present a process for detecting static anomalies based on mutation, equivalence checking, and quality checking. We show that this process is applicable to several types of artifacts produced at different phases of the software life cycle (at specification, design, and implementation levels), for several anomalies, and using several mutation operators.

The paper is organized as follows. Sect. 2 introduces some background on the classical definition of software anomaly, mutation, and the problem of equivalent mutants. Sect. 3 presents our definition of static anomalies in terms of equivalent mutants and a technique for discovering them. Sects. 4 and 5 show the application of the technique at the specification level (on fea-

ture models and NuSMV models), Sects. 6 and 7 at the implementation level (on Boolean expressions and source code), and Sect. 8 on package dependency graphs (either at the design or implementation level). Sect. 9 discusses some threats to the validity of our proposal, while Sect. 10 presents some related work. Finally, Sect. 11 concludes the paper.

2. Background

We here briefly review some basic concepts on software anomalies, mutation, and equivalent mutants.

2.1. Software anomalies

Software anomalies are defined in the IEEE standard [10] as:

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.

In this paper, we refer to *software artifact* as any product that is developed along the software life cycle at several levels: for example, source code at implementation level or models at specification level.

According to the IEEE standard, each software 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.

We here 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.

2.2. Mutation

Mutation is a well known technique in the context of software artifacts as program code and formal specifications. It consists in introducing small modifications into the artifact such that these simple syntactic changes, called *mutations*, represent typical mistakes that programmers or designers often make. These faults are deliberately seeded into the original artifact in order

to obtain a set of faulty variations called *mutants*. A transformation rule generating a mutant from the original artifact is known as *mutation operator*.

Mutation is very often used in combination with program testing, and its use is twofold. Mutants are classically used to assess the quality of test suites. High quality test suites should be able to distinguish the original program from its mutants, i.e., to detect the seeded faults. Given a test suite T, if the result of running a mutant is different from the result of running the original program for at least a test case in T, then the mutant is said to be *killed*; otherwise, it is said to have *survived*. A test suite has a *mutation score* equal to the portion of mutants it can kill. After all test cases have been executed, there may still be a few surviving mutants. To improve the test suite T, the program tester can provide additional test inputs to kill these surviving mutants. In this case, mutation is used for *test generation* purposes.

The history of mutation can be traced back to the 70s [1]. Mutation has been mainly applied to programming languages, but also at the design level to formal specifications [2, 3, 4, 11, 12, 13].

2.3. Equivalent mutants

When a mutant has the same meaning (e.g., the same behavior for programs or the same logical models for Boolean expressions) as the original artifact, it is said to be *equivalent*. These mutants are syntactically different but semantically equivalent to the original artifact.

Equivalent mutants are considered as one of the main causes why mutation testing is seldom used in practice [5, 6]. In software testing, equivalent mutants do not represent actual faults and can not be detected (killed) by a test. They thus reduce the quality index (mutation score) of a test suite without a real justification. In test generation, equivalent mutants consume resources without producing any useful test.

In code mutation, automatically detecting all equivalent mutants is impossible [14] because program equivalence is undecidable [15]. Several attempts try to eliminate (e.g., by filtering) or to avoid them [7, 8].

In the context of other software artifacts with a higher abstraction level than code, and with a concept of equivalence and a technique for checking it, some approaches for detecting equivalent mutants have been developed [2, 3]. However, also in these contexts, equivalent mutants are seen as an inconvenience [16, 17] and efforts to detect them are only finalized to skip them.

In the approach presented here, we rehabilitate equivalent mutants, in the sense that the goal of detecting them is finalized to use them to improve

artifacts' qualities, and not to eliminate or avoid them.

3. Using mutation to detect static anomalies

In this section, we introduce our definition of static anomalies in terms of equivalent mutants, and we propose a technique for anomaly detection.

3.1. Static anomalies

We 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 one 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.

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 detection**)**.** Given an artifact A and its mutation A', if A' is equivalent to A (i.e., $A \equiv A'$) and $A' >_q A$, then A contains a static anomaly. The static anomaly is the *difference* between A' and A.

Thesis 1. Each classic static anomaly introduced in the literature can be redefined in terms of Def. 1.

3.2. Detecting static anomalies

Def. 1 gives the foundation of a methodology for defining, finding, and possibly removing static anomalies. Normally, the designer has a precise static anomaly and the related quality q in mind. Def. 1 requires her/him to devise a suitable mutation operator, and ways for checking equivalence between artifacts and for comparing their qualities.

Given a quality q and a mutation operator m, the general process followed for detecting possible anomalies (in terms of quality q) in an artifact A is as follows:

1. build a mutation A' for A using m;
2. check the equivalence between A' and A;
3. compare qualities $q(A)$ and $q(A')$ of A and A';
4. if $A' \equiv A$ and $q(A') > q(A)$, then an anomaly has been found in A.

Table 1: Anomaly detectors classification

		Quality check?	
		Yes	No
Equivalence check?	Yes	Weak detector	Quality enhancer
	No	Refactorer	Anomaly remover

In this way, mutation operators are used as *anomaly detector*s. A mutation operator m can be defined as an anomaly detector w.r.t. a quality q only if it can produce equivalent mutants having better quality.

3.3. Anomaly detector classification

Given a quality q and a mutation operator m, the process described in the previous section does not always require to check the quality and/or the equivalence since some mutation operators always increase a given quality and/or produce equivalent mutants (or they guarantee to increase the quality when the mutant is equivalent). As shown in Table 1, we can identify four kinds of anomaly detectors, according to the fact that equivalence checking (i.e., step 2 of the process) and/or the quality checking (i.e., step 3) must be executed or not.

We overview the four kinds of detectors using examples from code mutation, but the classification applies to any mutation of any software artifact.

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 (i.e., both steps 2 and 3 of the process must be executed). In this case, we call the mutation operator *weak detector*.

Example 1 (Weak detector)**.** *Statement reordering*, i.e., swapping the order of two statements of a program, may produce an equivalent mutant (if the two statements are *independent*), and improve qualities *readability* and *efficiency* of compiling [18]. Both qualities can be defined in terms of length of du-paths [19] (i.e., the paths connecting the definitions of variables with their use): the shorter the du-paths are, the higher the quality is. If a reordering produces an equivalent mutant with better readability (or efficiency), then the original artifact contains a static anomaly (*low readability*). Given the code fragment `x = 3; y = 6; z = x*2;`, applying statement reordering to the last two statements produces an equivalent mutant (i.e., `x = 3; z = x*2; y = 6;`) with a better readability, because the length

of the du-path from the definition of `x` (i.e., `x = 3;`) to its use in `z = x*2;` has been reduced from 2 to 1.

Some mutation operators always increase a given quality, but the produced mutant may be non-equivalent. Some other mutation operators, although they can sometime weaken the quality, they guarantee to increase it when the mutant is equivalent. Therefore, when using these two kinds of mutation operators, only the equivalence must be checked (i.e., step 3 is not executed); in this case, we call the mutation operator *quality enhancer*.

Example 2 (Quality enhancer)**.** Let us consider the Statement Deletion mutation operator (SDL) [20], which removes a statement in a program, and the quality *compactness*, defined as $1/(\#statements+1)$. 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 behavior, 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 (i.e., step 2 is not executed). In this case, we call the mutation operator *refactorer*.

Example 3 (Refactorer)**.** In a program, the *rename* refactoring renames variables, methods, classes, etc., avoiding name clashes. Such refactoring always produces equivalent mutants and, with respect to the *readability* quality, may both increase and decrease the readability of a program. If the readability increases, the original program contains a static anomaly (*low readability*).

In the best case, the mutation operator always produces an equivalent mutant with better quality and, therefore, neither the equivalence nor the quality must be checked (i.e., both steps 2 and 3 are not executed). In this case, we call the mutation operator *anomaly remover*.

Example 4 (Anomaly remover)**.** The *clean up* feature, provided by several IDEs (e.g., Eclipse), refactors the code to make it compliant with language standards (e.g., no unused imports, well-formatted code, presence of annotations, ...). It always produces an equivalent mutant with better *compliance to standards*.

Among the four kinds of anomaly detectors previously described, *anomaly remover*s are the easiest ones to use, since they neither require quality measurement nor equivalence checking. However, the usage simplicity is usually paid by a greater complexity of the mutation operators that, somehow, incorporate the two checks (they can be seen as refactorers with an implicit quality checking). Having an operator with these characteristics is difficult and, therefore, we will not consider them.

Also *refactorer*s are easy to apply, since they do not require equivalence checking that it is usually very expensive. However, in the following we will consider them only for one artifact because (a) the class of anomalies they can detect is rather limited, and (b) *refactoring* (or *restructuring* [21]) techniques have already been widely studied in literature [22, 23].

The aim of the this paper is investigating the use of equivalence checking for detecting static anomalies. Therefore, we mainly consider weak detectors and quality enhancers. However, weak detectors are usually poorly defined and expensive to apply (both quality and equivalence must be checked) and, therefore, we will seldom use them. We will mainly focus on quality enhancers that permit us to experiment how equivalent checking techniques can be reused for detecting static anomalies and, in case of anomaly detection, automatically improve the quality.

In the following sections, we present different kinds of static anomalies for different artifacts that are used throughout the software life cycle: feature models and NuSMV models at specification level, Boolean expressions and source code at implementation level, and package dependency graphs at the design or implementation level. We show how static anomalies can be detected by suitable mutation operators. For each software artifact kind, we briefly describe its notation, the mutation operators that have been proposed in literature for it, and the technique(s) used for checking the equivalence; then, we present some static anomalies with the anomaly detectors able to discover them, and finally we report some experiments we have done. All the detectors reported in the following sections are quality enhancers, unless stated otherwise. All the experiments have been executed on a Linux PC with two Intel(R) Xeon(R) CPU E5-2630 (2.30GHz) and 64 GB of RAM.

4. Feature models

In software product line (SPL) engineering, feature models (FMs) are a special type of information model representing all possible products of an

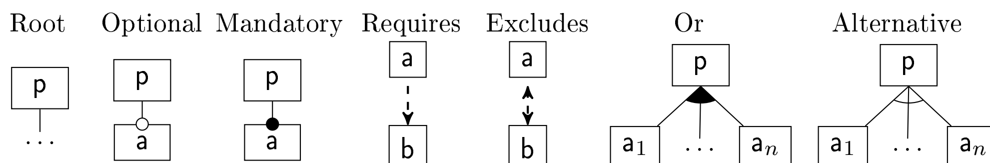


Figure 1: Feature models standard notation

SPL in terms of features and relations among them [24]. 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 Fig. 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 model semantics can be rather simply expressed by using propositional logic [24]. 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 denote as $\textsc{bof}(\mathcal{M})$ its representation as propositional formula.

4.1. Mutation operators

Mutation analysis has been applied to feature models in [25, 26]. In [13], we devised several fault classes and corresponding mutation operators for feature models. In this paper, those used to discover anomalies are:

- **MF** *Missing Feature*: A feature f (except the root) is removed and it is replaced by its sub-features which inherit the same relation the removed feature f had with its parent. f is replaced by `false` in any constraint containing it.
- **MC** *Missing Constraint*: A constraint is removed.
- **OTM** *Optional To Mandatory*: An optional relation is changed to mandatory.

Similar operators have also been defined in [27]. In [28], instead, mutation is applied to the representation as propositional formula of the feature model.

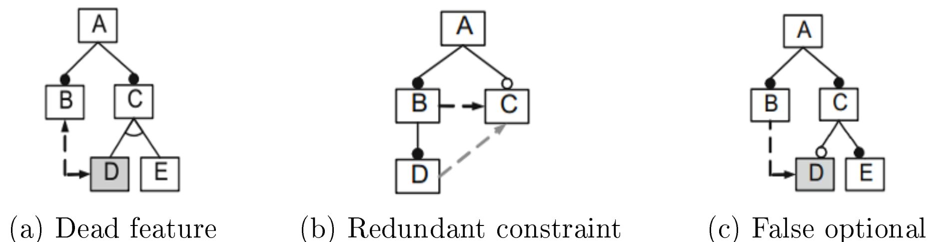
4.2. Equivalence

Two feature models are *equivalent* if they describe the same set of products. Some approaches try to avoid to generate equivalent mutants [16, 17], while others provide some techniques for detecting them. A technique for equivalent mutants detection that uses an SMT-solver is described in [13]; it consists in representing a feature model $\mathcal{M}$ and one of its mutants $\mathcal{M}'$ as propositional formulas $\textsc{bof}(\mathcal{M})$ and $\textsc{bof}(\mathcal{M}')$, and checking their equivalence.

4.3. FMs static anomalies

4.3.1. Dead feature

A feature is *dead* if it is not present in any product of the FM. As quality, we adopt *liveness*, defined as $^{(\#Fs-\#DFs)}/_{\#Fs}$, where $\#Fs$ is the number of features and $\#DFs$ the number of dead features. The higher the liveness, the fewer dead features are contained in the FM. Note that identifying dead features is important. In case of dead feature, two cases are possible: either the feature is really useless and therefore it should be removed (so that the final user does not think that it can belong to some products), or it is useful (i.e., it should be possible to select it) and so the designer finds a fault in the model. In the latter case, dead features normally arise when the designer introduces wrong constraints. We can detect dead features by using 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



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

Figure 2: Examples of feature models anomalies (in gray)

the removed feature is dead, then the mutant is equivalent and it has a better liveness[1]. In Fig. 2a, feature D is dead because it can never be selected in any product: removing it from the FM does not modify the set of products, but increases the liveness from $^4/_5$ to 1.

4.3.2. Redundant constraint

A constraint is *redundant* when it does not add any further restriction or information to the FM. Redundant constraints make the FM more difficult to understand and introduce useless relations between features which may complicate the products generation for the model.

As quality, we adopt *compactness*, defined as $^1/_{(\#CNFlit+1)}$, being $\#CNFlit$ the number of literals of the CNF conversion of the conjunction of all the 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 compactness and if the mutant is equivalent, the removed constraint is redundant. In Fig. 2b, 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: the compactness increases from $^1/_3$ to 1.

4.3.3. 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). Identifying false optional features and turning them to mandatory make the model more clear and easier to handle by tools.

[1] 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 $^{(\#Fs'-\#DFs')}/_{\#Fs'}$ where $\#Fs' = \#Fs - 1$ and $\#DFs' = \#DFs - 1$.

As quality, 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 the mutation operator OTM always increases the solvability, but it may produce non-equivalent mutants. In Fig. 2c, feature D is a false optional because it is selected in all products; turning D to mandatory produces an equivalent mutant with better solvability (from ${}^{4}/_{5}$ to 1). Note that removing the anomaly makes the *requires* constraint between B and D redundant (see Sect. 4.3.2). 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.

4.4. Experiments

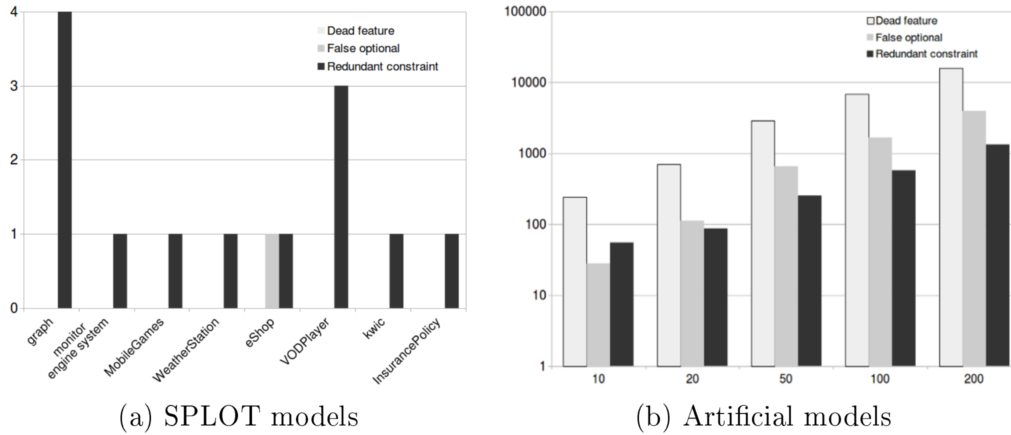
As case studies, we have taken 53 SPLOT models[2] which are often used as benchmarks. We have also included 1,000 artificially generated models that are available on the FeatureIDE website[3]; we took 200 models for five different groups characterized by the number of features: 10, 20, 50, 100, 200 features. The entire experiment, over the two sets, took globally around 17 mins, where 99.23% of the time was spent on equivalence checking. Discovering a single anomaly took on average 24 ms. The percentage of time spent for checking the equivalence varies from 63% (for the set with 10 features) to 99% (for the set with 200 features). Tables 2 shows the experimental results for the two sets of specifications, in terms of number of applied mutations, number of discovered anomalies, and number of anomalies discovered also by the validator of FeatureIDE. 14.5% of SPLOT models and 91.5% of artificial ones have at least one anomaly; Fig. 3a shows the distribution of the anomalies for the eight affected SPLOT models, and Fig. 3b shows the distribution of the anomalies for the 5 groups of the artificial set. Note that SPLOT models mainly contain redundant constraints because modelers usually tend to overspecify their models by adding *similar* constraints. Artificial models, instead, contain several anomalies of any kind because their automatic generation does not take into consideration quality measures.

[2] http://www.splot-research.org/

[3] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/

Table 2: Feature models experiments

Anomaly	SPLOT models # muts	SPLOT models # equiv (anoms)	SPLOT models # found by F.IDE	Artificial models # muts	Artificial models # equiv (anoms)	Artificial models # found by F.IDE
Dead feature	1,632 (MF)	0	0	774,400 (MF)	33,340	33,340
Redundant constraint	257 (MC)	13	13	77,599 (MC)	3,067	3,067
False optional	359 (OTM)	1	1	157,865 (OTM)	6,420	6,420



(a) SPLOT models (b) Artificial models

Figure 3: Feature models anomalies distribution

5. NuSMV

NuSMV [29] is a symbolic model checker that allows the representation of synchronous and asynchronous finite state systems, and the verification of temporal logic formulas. A NuSMV model describes the behavior of a Finite State Machine (FSM) in terms of a "possible next state" relation between states that are determined by the values of variables. A variable type can be Boolean, integer defined over intervals or sets, or an enumeration of symbolic constants. A *state* of the model is an assignment of values to variables. The value of state variables can be determined in the **ASSIGN** portion in the following way:

ASSIGN var := simple_expression −− *simple assignment*
ASSIGN **init**(var) := simple_expression −− *init value*
ASSIGN **next**(var) := next_expression −− *next value*

A simple assignment determines the value of variable *var* in the current state, the instruction **init** permits to determine the initial value(s) of the variable, and the instruction **next** is used to determine the variable value(s) in the next state(s). In both *simple-* and *next-* expressions, a variable value can be determined either unconditionally or conditionally, depending on the form of the expression. One of the most used expressions in NuSMV is the **case** expression, defined as:

case
    left_exp$_1$: right_exp$_1$;
    ...
    left_exp$_n$: right_exp$_n$;
esac

It returns the value of the first *right_exp*$_i$ such that the corresponding *left_exp*$_i$ guard evaluates to TRUE, and the previous $i-1$ guards evaluate to FALSE.

Computation Tree Logic (CTL) and *Linear Temporal Logic* (LTL) formulas are specified in the **LTLSPEC** and **CTLSPEC** sections.

5.1. Mutation operators

Some mutation operators for SMV (the ancestor of NuSMV) models have been proposed in [30] for test case generation by model checking: some modify the description of the FSM and others the temporal logic formulas. A wider set of mutation operators has been defined in [3] for NuSMV models: some operators are the usual ones described in literature (e.g., **LOR**, **SA0**), others are more specific operators tailored on NuSMV models (e.g., **MB**, **SB**). For our purposes, we consider MB and add another one (PS):

MB *Missing Branch*: In a case expression, one of the branches is removed.

PS *Polarity substitution*: In a temporal property, a subformula is substituted with its *polarity* (see Sect. 5.3.2).

5.2. Equivalence

NuSMV models describe Kripke structures and, therefore, equivalence of NuSMV models can be reduced to equivalence of Kripke structures. Such approach is followed in [3], where a technique is presented for checking equivalence of two NuSMV models having the same signature (i.e., same variables defined over the same domains). It consists in the verification of a set of properties over a model obtained from the *merging* of the two models.

```
MODULE main
VAR
    x: boolean;
    y: boolean;
ASSIGN
    x := case
            a: ...;
            a & b: ...;
            ...
        esac;
    y := case
            a: ...;
            !a: ...;
            TRUE: ...;
        esac;
```

Code 1: Examples of unnecessary branches

```
MODULE main
VAR
    hour: 0..23;
    amPm: {AM, PM};
ASSIGN
    init(hour) := 0;
    next(hour) := (hour + 1) mod 12;
    amPm := case
                hour < 12: AM;
                hour > 11: PM;
            esac;

-- The property is vacuous in amPm = PM
CTLSPEC AG(hour > 11 -> amPm = PM )
```

Code 2: Example of vacuous satisfaction

5.3. Static anomalies

5.3.1. Unnecessary branch

In a case expression, a branch is *unnecessary* if its guard can never be evaluated to true or never executed. Let us consider the case expression defining variable `x` in Code 1. The second guard `a & b`, when evaluated, is never true, because `a` is false (the first guard `a` does not hold). Let us now consider the case expression defining variable `y` in Code 1. In this case, the last branch is never executed because either the first or the second branch is taken. Identifying unnecessary branches is important: either they are really useless and so they should be removed to make the model more readable, or they should be executed sometimes and so there is a behavioral fault in the model.

As quality, we consider *minimality of branches*, defined as $1/(\#branches+1)$. The mutation operator MB always increases the *minimality of branches*, but it may produce non-equivalent mutants.

5.3.2. Vacuous satisfaction

A well known problem in formal verification is *vacuous satisfaction*: A property is vacuously satisfied if that property is satisfied and proved true regardless of whether the model really fulfills what the modeler originally had in mind or not. For example, the LTL property $\mathbf{G}(x \rightarrow \mathbf{X}(y))$ is vacuously

satisfied by any model where x is never true. Vacuous properties can convey a false confidence in the model correctness and, therefore, finding them is mandatory because they are a sign of a problem either in the model or in the temporal property.

As quality, we adopt *property fullness*, defined as $1/(\#vacProps+1)$, being $\#vacProps$ the number of vacuous properties of the model.

Several techniques to detect vacuity have been proposed [31, 32]. The general strategy to detect vacuity consists in replacing parts of the property and seeing if this has any effect on the result of the verification. In order to detect vacuity, it is sufficient to replace a non-constant subformula ϕ of property φ with `true` or `false` [32], depending on the polarity of ϕ in φ. The polarity of a subformula ϕ is positive if it is nested in an even number of negations in φ, otherwise it is negative; $pol(\phi)$ is a function such that $pol(\phi) =$ `false` if ϕ has positive polarity in φ, and $pol(\phi) =$ `true` otherwise[4].The replacement of subformula ϕ with ψ in formula φ is denoted as $\varphi[\phi \leftarrow \psi]$.

In order to discover anomalies of this kind, we introduce the mutation operator *polarity substitution* (**PS**). Given a property φ of the model and any atomic proposition ϕ in φ, PS mutates φ into $\varphi' = \varphi[\phi \leftarrow pol(\phi)]$. Because PS does not change the part regarding the behavior of the system, the mutated specification is equivalent to the original one iff φ' is still true.

PS can keep the *property fullness* of the model unchanged (when a property contains multiple vacuous subformulas), but it can never diminish it; moreover, it can produce non-equivalent mutants when the modified temporal property is falsified. However, whenever the mutant is equivalent, the quality increases: therefore, the operator is a quality enhancer.

Let us consider the NuSMV model shown in Code 2. The CTL property checks that if *hour* is greater than 11, then the value of variable *amPm* is *PM*. However, the model does not allow *hour* to be greater than 11; therefore, the CTL property is true regardless the value of *amPm*, since the antecedent of the implication is always false: the CTL property is vacuously true for $amPm = PM$. The mutation operator PS applied to the consequent of the implication produces the specification **AG**(hour > 11 −> FALSE) that is still true over the model. The mutant does not contain vacuous properties

[4] As in [32], we assume that all the occurrences of the subformula ϕ in φ are of a *pure polarity*, i.e., they are either all under an even number of negations (positive polarity), or all under an odd number of negations (negative polarity).

Table 3: NuSMV experiments

Anomaly	# mutations	# equivalents (anomalies)	specs with anomaly (%)
Unnecessary branch	12,085 (MB)	2,646	24.56
Vacuity	1,069 (PS)	36	25.81

anymore (constant subformulas are not considered in vacuity evaluation); property fullness has increased from 1/2 to 1. Note that, in this case, the problem is in the model, not in the specification.

5.4. Experiments

We have taken 68 NuSMV models from the NuSMV repository[5] and computed their anomalies. The experiment for *unnecessary branch* anomaly took around 243 mins, where 5% of the time was spent on equivalence checking. The experiment for *vacuity*, instead, took around 4 mins, where 87% of the time was spent on equivalence checking. Note that equivalence checking for unnecessary branches is quite fast, since it is interrupted as soon as equality is proved to not hold. In the experiment for vacuity, instead, most of the time is spent on equivalence checking since the mutation is very fast and the whole experiment basically consists in verifying the mutated properties. Table 3 reports the results of the experiments, in terms of number of applied mutations, number of found anomalies, and percentage of models having at least an anomaly. Note that 21.9% of the branches is unnecessary. This is mainly due to the fact that the modeler usually adds, as last branch of a case expression, a *default branch* that must be executed if none of the previous branches has been taken: results show that often this branch can be omitted since it is never executed. Experiments also show that 3.4% of the specified formulas are vacuous. This a sign of a serious problem: in case of vacuous satisfaction, the modeler thinks of having correctly verified the intended requirements but, instead, there is a problem either in the model or in the temporal specification. The percentage of models with at least an anomaly of a given type is quite high (24.56% and 25.81% for the two anomaly types); if we only consider the specifications actually containing the construct that

[5] http://nusmv.fbk.eu/

can be affected by the anomaly (i.e., branches and temporal properties), the percentages raise to 29.79% and 27.59%.

6. Boolean expressions

Boolean expressions are those involving Boolean operators like AND, OR, and NOT (denoted by $\wedge$, $\vee$, $\neg$). Improving qualities of Boolean expressions is important since they frequently occur in software artifacts. They express 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, as seen in Sect. 4). Boolean inputs are explicitly found in models of digital logic circuits: in these cases, the extraction of Boolean expressions is rather straightforward [33]. More often, Boolean inputs derive from abstraction techniques that consist in replacing complex formulas with Boolean predicates. These techniques can be applied to high level specifications or to source code, for instance, in order to obtain Boolean programs [34].

We here follow the definitions and notations used in [35]: 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 is a constraint among the variables of a Boolean expression, i.e., another Boolean expression specifying the *allowed* logical values.

6.1. Mutation operators

There are 10 classical mutation operators (also known as *fault classes*) for Boolean expressions [35]. For our purposes, we only introduce the two following mutation operators (and we use the same Boolean expression $\varphi : x_1 \vee \neg x_2$ to explain them) that are sufficient to detect the kind of anomalies we consider:

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

SA0/SA1 *Stuck-At-0/1 Fault*: a subexpression is replaced by `false`/`true` in the expression. For example, $x_1 \vee$ *true* is an SA1 mutation of φ. Note that we have slightly modified the classical definition of SA0/SA1 that requires that only conditions (and not general subexpressions) are replaced.

6.2. 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 constraint δ, a mutation φ' is equivalent to φ if $\delta \models \varphi \leftrightarrow \varphi'$. Checking equivalence of two Boolean expressions is straightforward by using a SAT (or an SMT) solver.

6.3. Static anomalies

6.3.1. 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 $1/(\#conditions+1)$. Redundant conditions can be discovered and removed by the mutation operator MVF. Applying MVF 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 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 better simplicity (from $1/3$ to $1/2$).

6.3.2. Fixed-value expression

A non-constant (sub)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). Identifying a fixed-value expression e is important: if e should really always keep that value, substituting e with its constant value increases the readability; instead, if e should be able to change its value, this anomaly is a sign of a fault in the expression.

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 $(\#constants+1)/(\#conditions+1)$. Fixed-value expressions can be detected by the mutation operators SA0/SA1. The application of SA0 and SA1 requires to check the equivalence but not the quality, since the coverability is always increased by SA0/SA1.

Table 4: Boolean expressions experiments

Anomaly	# mutations	# equivalents (anomalies)	specs with anomaly (%)
Redundant condition	52,579 (MVF)	10,432	27.9
Fixed-value expression	217,010 (SA0/SA1)	37,907	30.2

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

6.4. Experiments

For experimentation, we use the benchmarks that we build in [36] for test generation for Boolean expressions. In that work, we selected several specifications from different sources: 89,332 specifications were extracted from source code of projects hosted on SIR[6], 99,507 specifications were extracted from circuit models in ISCAS format [33], and 131 specifications were derived from the guards of models of the NuSMV model checker. Each specification is composed of a Boolean expression and a constraint. We then identified isomorphic specifications (i.e., specifications that can be transformed one in the other by a simple renaming of the inputs), obtaining 755 single specifications.

For each kind of anomaly, we have computed the number of anomalies of all the considered specifications. The experiment for *redundant condition* took around 44 mins, where 32% of the time was spent in equivalence checking; the experiment for *fixed-value expression* took around 63 mins, where 99.9% of the time was spent in equivalence checking. Note that the mutation operator MVF is computationally expensive and, therefore, it has a significant impact in the experiment for *redundant condition*; mutation operators SA0 and SA1, instead, are very simple and their execution time is negligible in the experiment for *fixed-value expression*. Moreover, there are much more SA0/SA1 mutations than MVF mutations: this is the reason why the experiment for *fixed-value expressions* takes more time. Results are shown in Table 4. It reports the total number of applied mutations, the total number of found anomalies, and the percentage of specifications with at least an anomaly. For the *redundant condition* anomaly, results are related to 754

[6]Software-artifact Infrastructure Repository http://sir.unl.edu

specifications, because the number of mutations of one specification was too high and so we were not able to complete the computation of the number of anomalies. Experiments show that 27.9% of specifications contain at least a redundant condition: this result is mainly due to expressions derived from HW models that are highly redundant. For the same reason, there are several anomalies of type *fixed-value expression* (30.2% of specifications). Note that anomaly *fixed-value expression* subsumes anomaly *redundant condition*, because if a condition is redundant, it can be substituted with either *true* or *false*; however, some specifications have a fixed-value subexpression but not a redundant condition: these specifications are only equivalent to other specifications in which non-atomic subexpressions are substituted with *true* or *false*.

7. Source code

In this section, we consider programs as software artifacts. We report several static anomalies for source code in C, although we believe that most of them can occur in the majority of the programming languages.

7.1. Mutation operators

There exist many mutation operators for programs, although only few of them are sufficient to obtain most of the mutations [1]. For our purposes, we consider the classical operator **ROR** and we introduce two more operators:

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

SDL *Statement Deletion operator* [20]: It removes an entire statement from the program.

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

7.2. Equivalence

Equivalent mutants for source code leave the program's behavior unchanged. Detecting if a program and one of its mutant is equivalent is undecidable [15]. Moreover, the number of equivalent mutants can be high: empirical results show that from 10% to 45% of program mutants are equivalent and finding them by hand can be very time consuming [37]. Fortunately, there has been much research work on the automatic detection of equivalent mutants, as reviewed in details in [14]. Note that there exist some simple techniques that, although not complete, can be easily automatized. For instance, the use of compiler optimizations is advocated in [38].

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

Code 3: Useless code

```
#define DEBUG
void main(void) {
#if defined DEBUG
    printf("Mode=Debug");
#else
    printf("Mode=Release");
#endif
}
```

Code 4: Unreachable code

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

Code 5: Inefficient code

7.3. Static anomalies

7.3.1. 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. 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 3 and 4 contain dead code: the first assignment in Code 3 has no visible effects, while the code inside the else branch of the conditional statement of Code 4 is never executed. Applying the SDL mutation operator to the first assignment of Code 3 removes the static anomaly, since it produces an equivalent mutant which is more compact. Note that code review would easily find both anomalies, but it would require human effort.

7.3.2. 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 [39], and some automatic techniques for its measurement have been proposed [40]. Poor readability

anomalies can be discovered and removed by the RNM operator. Note that RNM is a refactorer, but if it is used in a suitable way (e.g., by using meaningful substituting names), then it becomes an anomaly remover.

7.3.3. 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 5 could be modified as shown in the comment and the code would be equally functional but more efficient. In this case, the ROR mutation operator would discover this anomaly. However, ROR is a weak detector since it may produce non-equivalent mutants and weaken the efficiency.

7.4. Experiments

We have investigated the use of the proposed technique in order to find *dead code* in real source code. We have gathered 17 simple C programs from the literature and from teaching material about dead code (including Wikipedia and the examples given in Sect. 7.3) and we have added all the complex programs presented in [38]. We have implemented a simple SDL operator which removes one entire statement at the time. To check the equivalence, we have used the simple TCE technique proposed in [38] which consists in compiling (with `gcc -O3`) both the original program and the mutant and checking the equivalence. In this way, we ultimately use the optimization techniques of the compiler to find dead code, as suggested also in [41, 42], although we use them in combination with mutation and in order to find anomalies (and not to have a more compact object code).

Table 5 reports the number of mutants we have generated and the number of anomalies we have found. We have also compared the results with the static code checker Cppcheck[7]. As shown by the table, our technique is capable of finding anomalies that Cppcheck cannot. For instance, the source code of Code 4 contains an anomaly that can be discovered only after compiling the code. This is not the only case: we found that many anomalies can pass undetected by Cppcheck, which adopts a rather conservative approach since it openly aims at having zero false positives. On the other hand, Cppcheck is capable of finding dead code that is not a single statement. For instance,

[7] http://cppcheck.sourceforge.net/

Table 5: Dead code anomalies in the source code. #M: number mutations, #A: number of discovered anomalies, #C: number of anomalies discovered by Cppcheck

Program	#lines	#M	#A	#C	Program	#lines	#M	#A	#C
es1.c (Code 3)	14	10	1	1	gzip_gzip.c	5,393	3,004	24	103
es2.c (Code 4)	14	10	1	0	gzip_trees.c	2,113	1,250	330	53
unreachable.c	16	11	1	1	make_job.c	5,622	3,191	14	111
wikipedia1.c	10	9	2	1	make_main.c	6,136	3,582	16	120
wikipedia2.c	12	9	1	0	msmtp_msmtp.c	7,190	4,461	959	133
git_diff.c	18,194	12,344	86	429	msmtp_smtp.c	4,405	2,581	740	69
git_refs.c	15,944	10,606	81	440	vim_eval.c	23,508	16,718	26	258
gsl_blas.c	8,986	5,847	1,617	60	vim_spell.c	17,658	12,410	18	231
gsl_gen.c	9,102	5,817	1,786	70					
					Total			5,702	2,082

Cppcheck has found several `struct` and function declarations that are never used, while we were able to find only useless statements. We plan to extend our SDL operator in order to find more complex anomalies. Our technique requires much more resources to be performed (the entire benchmark took 56 hours, while around 3 minutes for Cppcheck). Most of time (99.8%) is spent by the equivalence checking: for the biggest file, it took 22 hours to complete the equivalence checking of all the 16,718 mutants. However, our technique is not constrained by the power of the static analysis rules, which may be difficult to write, it produces only true positives, and it can be easily performed. To reduce the required time, we plan to select the compile optimizations, to filter the mutations in order to exclude syntactically wrong mutants, and optimize the comparison of object files.

8. Package dependency graph

Package dependency graphs show the dependencies among packages as induced by their classes and they are commonly used to better understand and simplify the relationship among the components of a system [43]. Since they can be derived from requirements models, as UML package diagrams, or directly from source code using some static analysis tools, they can be used to detect related anomalies both at design and implementation level. In this work, we derive the package dependency graphs from the implementations. A package dependency graph represents all the packages of a program as vertexes, and dependencies among packages as arcs: a package `pkg1` depends on a package `pkg2` if a class of `pkg1` depends on a class of `pkg2`.

8.1. Mutation operators

As mutation operators for package dependency graphs we consider:

MC *Move Class*: A class is moved from its package to another package.

A2C *Abstract To Concrete*: An abstract class is transformed to concrete.

8.2. Equivalence

We assume that both mutation operators are always applicable and they produce equivalent artifacts. This is generally the case because a class can be moved from a package to another one (with the necessary changes in visibility) and an abstract class can be converted to a concrete one (with empty implementation if needed). If desired, the user can introduce constraints on the operator applicability.

8.3. Static anomalies

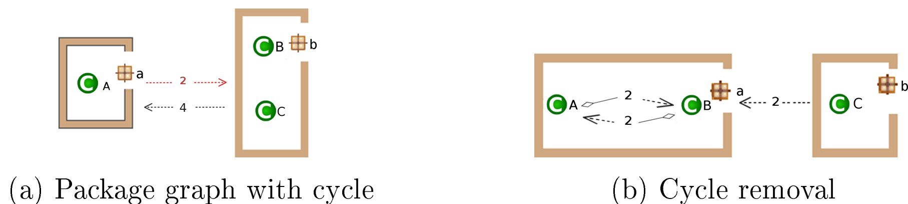
8.3.1. Low modularity

Design guidelines require that cyclic dependencies among packages should be avoided [44]. If two packages depend on each other, they constitute a unique entity that must be designed and maintained all together. This entity constitutes a *module* and it can be defined as the maximum set of packages depending (directly or indirectly) on each other. A module is identified in a dependency graph by a strongly connected component (SCC). The system is ideally modularized when each package is one SCC. As quality attribute we consider *modularity*, defined as $\#SCCs/\#packages$. A modularity equals to one represents the ideal situation, a modularity lower than one means that two or more packages belong to the same SCC. A low modularity implies a greater effort in designing, implementing, testing, and maintaining the system. By increasing the modularity, we can reduce the effort of the designer, the developer, and the tester [45].

Low modularity can be detected by using MC: if by moving a class we can increase the modularity of the system, we have found (and removed) an anomaly. Fig. 4a shows a package dependency graph with a cycle between packages `a` and `b`. Moving class `B` of package `b` to package `a` (as shown in Fig. 4b) removes the cycle and increases the modularity from $^1/_2$ to $^2/_2$.

8.3.2. Deviation from main sequence

Besides modularity, several metrics have been proposed for measuring the quality of the division in packages. The *abstractness* of a package `pkg` is given by the ratio of the number of its abstract classes to the number of its total



(a) Package graph with cycle

(b) Cycle removal

Figure 4: Example of package dependency graph anomalies

classes. The *afferent couplings* (Ca) is the number of other packages that depend upon (classes of) `pkg`. The *efferent couplings* (Ce) is the number of other packages that (the classes of) `pkg` depends upon. Ca and Ce are combined in the *instability* (I) metric, defined as $I = {Ce}/{(Ce+Ca)}$: it measures the resilience of a package to changes, where 0 indicates a totally stable package, whereas 1 indicates a totally unstable one.

Previous measures do not provide any quality assessment, but their combination does. The *distance from the main sequence* has been defined as $D = |A + I - 1|$, where the ideal main sequence is defined as $A + I = 1$ [44]. The less a package *deviates* from the main sequence, the better it is. The idea is that completely abstract packages should also be completely stable, i.e., if all the classes are abstract, they should not depend on classes in other packages. On the opposite side, totally concrete packages could be completely unstable, i.e., if all the classes are concrete, no class in other packages should depend on them. Also the packages staying on the main sequence are considered good: they have the *right* number of concrete and abstract classes in proportion to their efferent and afferent dependencies.

As quality measure, we adopt *average closeness to the main sequence* (AvgClMS), defined as $1 - \sum_{i=1}^{n} {D_i}/{n}$, being D_i the distance from the main sequence of package $\texttt{pkg}_i$. Quality AvgClMS must be taken with a certain level of tolerance: a very small augment of AvgClMS does not really imply the presence of an actual anomaly. Therefore, in Def. 1 we consider that there is an improvement of the quality only if the difference between the qualities of the original graph A and the mutated graph A' is higher than a given threshold T, i.e., $q(A') > q(A) + T$. As value for T, we suggest 0.01.

An anomaly of *deviation from the main sequence* can be detected by using A2C, since it can decrease the abstractness of a package and, therefore, reduce the distance when $(A + I) > 1$. Also MC can detect an anomaly, since it can increase (resp. decrease) I (by affecting the values of Ca and Ce) and,

Table 6: Package dependency graphs experiments

Program	Low modularity			Deviation from the main sequence		
	# muts (MC)	# anoms	# JDepend	# muts (MC/A2C)	# anoms	# JDepend
antlr-4.5.1	31,896	0	29	31,935	582	28
argouml	121,121	520	53	121,218	259	95
choco-3.3.1-j7	85,244	1,108	96	85,294	1,832	86
jedit-5.3.1	45,115	85	29	45,164	3,832	28

therefore, reduce the distance when $(A+I) < 1$ (resp. $(A+I) > 1$). However, when moving a class, although we may reduce the deviation of a package, we may also increase the deviation of another one, and so it is not guaranteed that AvgClMS increases.

8.4. Experiments

For experimentation, we took the four projects considered in [46] (namely, ANTLR, ArgoUML, Chocho solver, and jEdit) and we analyzed their package dependency graphs in order to detect the two considered anomalies. Since all our mutation operators are refactorings, we only had to measure the quality. The experiment for *low modularity* took 382 secs, and the one for *deviation from the main sequence* took 946 secs. Table 6 reports the results. It also reports the number of anomalies detected by the analysis tool JDepend. Regarding *low modularity*, JDepend finds some cycles in all the projects, whereas we do not find cycles in antlr: however, JDepend can report some false positives. For the other three projects, we report more anomalies since we can remove a cycle with multiple mutations. For *deviation from the main sequence*, JDepend can compute the distance for each package: using the tool, we assume that there is an anomaly if the distance is greater than 0.1. For each project, we report the number of packages for which there is an anomaly. Again, we cannot really compare the results of our approach with JDepend results, since we compute the average of the distances over the packages and we state that there is an anomaly if the graph can be improved by mutation (that it is not possible on JDepend that only reasons on the original graph): however, we can observe that we are able to identify anomalies of *deviation from the main sequence* as JDepend.

9. Threats to validity

Our approach is subject to some threats to validity.

First, removing a static anomaly is not always the right choice, since its real cause may be a different anomaly. For instance, a feature in an FM is dead (see Sect. 4.3.1) because the model is over-constrained: removing the feature 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 seen in Sect. 4.3.3 for false optionals; in that case, the introduced anomaly can be removed without reintroducing the original anomaly, but there may exist two *seesawing* anomalies: removing one causes the appearance of the other.

Sometimes two qualities may be in conflict: removing an anomaly can increase a quality and decrease another. For instance, consider readability and compactness: the code `x = y++;` is more compact than `x = y; y ++;` but less readable. In this case, the user should choose which quality is more important and use that to guide the detection and removal of anomalies.

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 can be undecidable) and, because it is difficult to automate, a time-consuming activity [47]. However, in particular cases, incomplete techniques can be devised (see Sect. 7.2): 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 and NuSMV models), equivalence checking is feasible by using tools like SAT/SMT solvers or model checkers.

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 [40]. In these cases, the use of quality enhancers and anomaly removers is preferable since they do not require quality checking. For instance, clean up operators are widely used to increase readability.

We cannot exclude that some static anomalies defined in the literature may be not detectable by the proposed technique. However, the examples presented in this work report many types of anomalies/qualities and a wide variety of artifacts used in several phases of the software life cycle and, therefore, they give us confidence that Thesis 1 holds in practice.

10. Related work

Related work regarding mutation operators and equivalence checking for the considered software artifacts has been presented in the previous sections. We here review works presenting approaches for detecting static anomalies.

Feature model anomalies are considered also in [48], where the three kinds of anomalies we consider (i.e., dead feature, redundant constraint, and false optional) are described as *refactorings*. The approach is similar to ours since it is based on mutation (in [48], a mutant is called *edit*), but their algorithm is more complicated since they are not only interested in checking if the original model and the mutated one are equivalent (i.e., the mutated model is a *refactoring*), but they also want to discover if the mutated one is a *generalization* (i.e., it describes a proper superset of products), a *specialization* (i.e., it describes a proper subset of products), or an *arbitrary edit* (i.e., there is no proper inclusion in the described products) of the original one.

An approach for detecting static anomalies of NuSMV models has been proposed in [49]. The desired qualities are defined as *meta-properties* and they are expressed as temporal properties: the violation of a meta-property is the signal of the presence of a static anomaly.

For different programming languages, several tools automatically look for common errors as, for example, FindBugs, PMD and Checkstyle for Java, or Splint for C[8]. These tools look for erroneous code but also for *stylistic conventions* violations that may indicate a possible problem. For example, the pattern *Unwritten field* of FindBugs signals if a field has never been written and always returns its default value: the violation of this pattern could show that the field is not necessary or that it must be updated somewhere. Note that the use of mutation to improve the quality of programs is also proposed in [50, 51], with similar goal of removing anomalies related to efficiency. In [50], the authors aim at improving the efficiency to solve a given class of problems, and in [51] the authors show how to tune program parameters in order to reduce runtime costs.

Regarding software architectures and program refactoring, the classical technique for detecting anomalies is to use static analysis and some heuristics (like estimation of cycle *undesirability* in [46]). There exist several attempts to automatically improve the source code structure by using mutation. How-

[8] http://findbugs.sourceforge.net/, http://pmd.sourceforge.net/, http://checkstyle.sourceforge.net/, http://www.splint.org/

ever, some recent studies show that using fitness functions for code refactoring can be ineffective [52].

We had previous experience in detecting static anomalies also for software artifacts not presented here. A technique for detecting static anomalies for combinatorial models has been presented in [53]: it proposes different techniques for detecting redundant constraints, useless values, and useless parameters, but they are not expressed in terms of mutations of the model.

11. Conclusions

Equivalent mutants are usually seen as a drawback in mutation analysis and so techniques have been studied in the past for discovering them in different software artifacts. In this paper, we claim that equivalent mutants can be seen as an opportunity and techniques for equivalence checking can be useful for detecting static anomalies, i.e., deficiencies of a given quality (as compactness, readability, ...): if an equivalent mutant with a better quality can be found for a given artifact, this means that the artifact contained a static anomaly. We have proposed a technique that combines mutation, equivalence checking, and quality checking for detecting static anomalies. We have shown that the technique can be simplified (avoiding either the equivalence or the quality checking, or both) for some particular qualities and mutation operators. We have demonstrated that our proposal is applicable to different kinds of software artifacts, as feature models, NuSMV models, Boolean expressions, source code, and package dependency graphs. Although each software artifact has its own specific static anomalies, we may identify some commonalities among the anomalies of different artifacts. Several anomalies are related to a *lack of minimality* in the artifact (i.e., the presence of elements that can be removed without changing the meaning of the artifact), as *redundant constraint* in feature models and *redundant condition* in Boolean expressions; in both cases, the artifact without the anomaly is more compact. Other anomalies, instead, identify *unnecessary elements* (i.e., never *used* elements), as *dead feature* in feature models, *unnecessary branch* in NuSMV, and *dead code* in source code; in all the three cases, the anomalies can really indicate an element that can be removed or they can reveal a problem in the artifact. A third class of anomalies is related to a *lack of clarity* in the artifact, as *false optional* in feature models, *fixed-value expression* in Boolean expressions, and *low modularity* for package dependency graphs; in

all the three cases, the artifact without the anomaly is more clear to read and understand.

Experiments show that, in most cases, the proposed approach is able to capture the same anomalies detected by other techniques. We claim that, although our method cannot be competitive in terms of time with these techniques because equivalence checking is still a computationally expensive task, our approach constitutes a conceptual framework for any type of static anomaly detection methodology. The aim of our work is to rehabilitate the reputation of equivalent mutants and propose to consider them in future static analysis tools, in particular for those software artifacts for which equivalence checking is not so expensive.

As future work, we plan to define anomaly detectors using those mutation operators that are known to produce a lot of equivalent mutants [6]. We also plan to study new operators producing a great deal of equivalent mutants. This could open a new research direction in mutation analysis, complementary to the current approaches focused on mutation testing, in which equivalent mutants and operators generating them are welcome.

Acknowledgements

The work was partially supported by Charles University research funds PRVOUK.

References

[1] Y. Jia, M. Harman, An analysis and survey of the development of mutation testing, IEEE Transactions on Software Engineering 37 (5) (2011) 649–678. doi:10.1109/TSE.2010.62.

[2] S. D. R. S. De Souza, J. C. Maldonado, S. C. P. F. Fabbri, W. L. De Souza, Mutation Testing Applied to Estelle Specifications, Software Quality Control 8 (1999) 285–301. doi:10.1023/A:1008978021407.

[3] P. Arcaini, A. Gargantini, E. Riccobene, Using mutation to assess fault detection capability of model review, Software Testing, Verification and Reliability 25 (5-7) (2015) 629–652. doi:10.1002/stvr.1530.

[4] T.-C. Lee, P.-A. Hsiung, Mutation Coverage Estimation for Model Checking, in: F. Wang (Ed.), Automated Technology for Verification

and Analysis, Vol. 3299 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2004, pp. 354–368.

[5] A. J. Offutt, R. H. Untch, Mutation 2000: Uniting the orthogonal, in: W. Wong (Ed.), Mutation Testing for the New Century, Vol. 24 of The Springer International Series on Advances in Database Systems, Springer US, 2001, pp. 34–44. doi:10.1007/978-1-4757-5939-6_7.

[6] X. Yao, M. Harman, 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, ACM, New York, NY, USA, 2014, pp. 919–930. doi:10.1145/2568225.2568265.

[7] E. S. Mresa, L. Bottaci, Efficiency of mutation operators and selective mutation strategies: an empirical study, Software Testing, Verification and Reliability 9 (4) (1999) 205–232. doi:10.1002/(SICI)1099-1689(199912)9:4<205::AID-STVR186>3.0.CO;2-X.

[8] M. Harman, R. Hierons, S. Danicic, The relationship between program dependence and mutation analysis, in: W. Wong (Ed.), Mutation Testing for the New Century, Vol. 24 of The Springer International Series on Advances in Database Systems, Springer US, 2001, pp. 5–13. doi:10.1007/978-1-4757-5939-6_4.

[9] P. Arcaini, A. Gargantini, E. Riccobene, P. Vavassori, Rehabilitating equivalent mutants as static anomaly detectors in software artifacts, in: Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on, 2015, pp. 1–6. doi:10.1109/ICSTW.2015.7107452.

[10] IEEE standard classification for software anomalies (1044-2009) (1995).

[11] S. C. P. F. Fabbri, M. E. Delamaro, J. C. Maldonado, P. C. Masiero, Mutation analysis testing for finite state machines, in: Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on, 1994, pp. 220 –229. doi:10.1109/ISSRE.1994.341378.

[12] L. Liu, H. Miao, Mutation operators for Object-Z specification, in: Proceedings 10th IEEE Int. Conf. Engineering of Complex Computer Systems ICECCS 2005, 2005, pp. 498–506. doi:10.1109/ICECCS.2005.65.

[13] P. Arcaini, A. Gargantini, P. Vavassori, Generating tests for detecting faults in feature models, in: Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on, IEEE, 2015, pp. 1–10. doi:10.1109/ICST.2015.7102591.

[14] L. Madeyski, W. Orzeszyna, R. Torkar, M. Jozala, Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation, IEEE Trans. Softw. Eng. 40 (1) (2014) 23–42. doi:10.1109/TSE.2013.44.

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

[16] J. M. Ferreira, S. R. Vergilio, M. A. Quináia, A mutation approach to feature testing of software product lines, in: The 25th International Conference on Software Engineering and Knowledge Engineering, Boston, MA, USA, June 27-29, 2013., Knowledge Systems Institute Graduate School, 2013, pp. 232–237.

[17] R. Â. Matnei Filho, S. R. Vergilio, A mutation and multi-objective test data generation approach for feature testing of software product lines, in: XXIX Simpósio Brasileiro de Engenharia de Software (SBES 2015), 2015, pp. 1–10.

[18] Y. N. Srikant, P. Shankar, The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition, 2nd Edition, CRC Press, Inc., Boca Raton, FL, USA, 2007.

[19] P. Ammann, J. Offutt, Introduction to Software Testing, 1st Edition, Cambridge University Press, New York, NY, USA, 2008.

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

[21] W. G. Griswold, D. Notkin, Automated assistance for program restructuring, ACM Trans. Softw. Eng. Methodol. 2 (3) (1993) 228–269. doi:10.1145/152388.152389.

[22] T. Mens, T. Tourwe, A survey of software refactoring, Software Engineering, IEEE Transactions on 30 (2) (2004) 126–139. doi:10.1109/TSE.2004.1265817.

[23] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[24] D. Batory, Feature models, grammars, and propositional formulas, in: H. Obbink, K. Pohl (Eds.), Proceedings of the 9th International Conference on Software Product Lines, Vol. 3714 of Lecture Notes in Computer Science, Springer-Verlag, 2005, pp. 7–20. doi:10.1007/11554844_3.

[25] C. Henard, M. Papadakis, G. Perrouin, J. Klein, Y. L. Traon, Assessing software product line testing via model-based mutation: An application to similarity testing, in: Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, ICSTW '13, IEEE Computer Society, Washington, DC, USA, 2013, pp. 188–197. doi:10.1109/ICSTW.2013.30.

[26] C. Henard, M. Papadakis, G. Perrouin, J. Klein, 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, IEEE Press, Piscataway, NJ, USA, 2013, pp. 1245–1248.

[27] D. Reuling, J. Bürdek, S. Rotärmel, M. Lochau, U. Kelter, Fault-based product-line testing: Effective sample generation based on feature-diagram mutation, in: Proceedings of the 19th International Conference on Software Product Line, SPLC '15, ACM, New York, NY, USA, 2015, pp. 131–140. doi:10.1145/2791060.2791074.

[28] C. Henard, M. Papadakis, Y. Le Traon, Mutation-based generation of software product line test configurations, in: C. Le Goues, S. Yoo (Eds.), Search-Based Software Engineering, Vol. 8636 of Lecture Notes in Computer Science, Springer International Publishing, 2014, pp. 92–106. doi:10.1007/978-3-319-09940-8_7.

[29] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, NuSMV Version 2: An Open-Source Tool for Symbolic Model Checking, in: Proceedings of the 14th International Conference on Computer Aided Verification, Vol. 2404 of

Lecture Notes in Computer Science, Springer-Verlag, London, UK, 2002, pp. 359–364.

[30] P. E. Ammann, P. E. Black, W. Majurski, Using model checking to generate tests from specifications, in: Proceedings of the Second IEEE International Conference on Formal Engineering Methods, ICFEM '98, IEEE Computer Society, Washington, DC, USA, 1998, pp. 46–.

[31] I. Beer, S. Ben-David, C. Eisner, Y. Rodeh, Efficient detection of vacuity in ACTL formulas, in: Proc. 9th International Computer Aided Verification Conference, no. 1254 in Lecture Notes in Computer Science, 1997, pp. 279–290.

[32] O. Kupferman, M. Y. Vardi, Vacuity detection in temporal model checking, International Journal on Software Tools for Technology Transfer (STTT) 4 (2) (2003) 224–233.

[33] M. C. Hansen, H. Yalcin, J. P. Hayes, Unveiling the ISCAS-85 benchmarks: a case study in reverse engineering, Design Test of Computers, IEEE 16 (3) (1999) 72–80. doi:10.1109/54.785838.

[34] T. Ball, R. Majumdar, T. Millstein, 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, ACM, New York, NY, USA, 2001, pp. 203–213. doi:10.1145/378795.378846.

[35] K. Kapoor, J. P. Bowen, Test conditions for fault classes in Boolean specifications, ACM Transactions on Software Engineering and Methodology 16 (3) (2007) 10. doi:10.1145/1243987.1243988.

[36] P. Arcaini, A. Gargantini, E. Riccobene, How to optimize the use of SAT and SMT solvers for test generation of Boolean expressions, The Computer Journal 58 (11) (2015) 2900–2920. doi:10.1093/comjnl/bxv001.

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

[38] M. Papadakis, Y. Jia, M. Harman, 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, ICSE '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 936–946.

[39] K. Aggarwal, Y. Singh, J. Chhabra, An integrated measure of software maintainability, in: Reliability and Maintainability Symposium, 2002. Proceedings. Annual, 2002, pp. 235–241. doi:10.1109/RAMS.2002.981648.

[40] R. P. Buse, W. R. Weimer, Learning a metric for code readability, Software Engineering, IEEE Transactions on 36 (4) (2010) 546–558. doi:10.1109/TSE.2009.70.

[41] S. K. Debray, W. Evans, R. Muth, B. De Sutter, Compiler techniques for code compaction, ACM Trans. Program. Lang. Syst. 22 (2) (2000) 378–415. doi:10.1145/349214.349233.

[42] R. Bodík, R. Gupta, Partial dead code elimination using slicing transformations, in: Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation, PLDI '97, ACM, New York, NY, USA, 1997, pp. 159–170. doi:10.1145/258915.258930.

[43] M. Consens, A. Mendelzon, A. Ryman, Visualizing and querying software structures, in: Proceedings of the 1991 Conference of the Centre for Advanced Studies on Collaborative Research, CASCON '91, IBM Press, 1991, pp. 17–35.

[44] R. C. Martin, Agile Software Development: Principles, Patterns, and Practices, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[45] K. J. Sullivan, W. G. Griswold, Y. Cai, B. Hallen, The structure and value of modularity in software design, in: Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-9, ACM, New York, NY, USA, 2001, pp. 99–108. doi:10.1145/503209.503224.

[46] J. Laval, J. Falleri, P. Vismara, S. Ducasse, Efficient retrieval and ranking of undesired package cycles in large software systems, Journal of Object Technology 11 (1) (2012) 1–24. doi:10.5381/jot.2012.11.1.a4.

[47] B. J. Grun, D. Schuler, A. Zeller, The impact of equivalent mutants, in: The 4th International Workshop on Mutation Analysis (Mutation 2009) - ICST, IEEE, 2009, pp. 192–199.

[48] T. Thum, D. Batory, C. Kastner, Reasoning about edits to feature models, in: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 254–264. doi:10.1109/ICSE.2009.5070526.

[49] P. Arcaini, A. Gargantini, E. Riccobene, A model advisor for NuSMV specifications, Innovations in Systems and Software Engineering 7 (2) (2011) 97–107. doi:10.1007/s11334-011-0147-2.

[50] J. Petke, M. Harman, W. Langdon, W. Weimer, Using genetic improvement and code transplants to specialise a C++ program to a problem class, in: M. Nicolau, K. Krawiec, M. I. Heywood, M. Castelli, P. García-Sánchez, J. J. Merelo, V. M. Rivas Santos, K. Sim (Eds.), Genetic Programming, Vol. 8599 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2014, pp. 137–149. doi:10.1007/978-3-662-44303-3_12.

[51] F. Wu, W. Weimer, M. Harman, Y. Jia, J. Krinke, Deep parameter optimisation, in: Proceedings of the 2015 on Genetic and Evolutionary Computation Conference, GECCO '15, ACM, New York, NY, USA, 2015, pp. 1375–1382. doi:10.1145/2739480.2754648.

[52] C. Simons, J. Singer, D. R. White, Search-based refactoring: Metrics are not enough, in: M. de Oliveira Barros, Y. Labiche (Eds.), Search-Based Software Engineering, Vol. 9275 of Lecture Notes in Computer Science, Springer International Publishing, 2015, pp. 47–61. doi:10.1007/978-3-319-22183-0_4.

[53] P. Arcaini, A. Gargantini, P. Vavassori, Validation of models and tests for constrained combinatorial interaction testing, in: The 3rd International Workshop on Combinatorial Testing (IWCT 2014) - ICST, 2014, pp. 98–107. doi:10.1109/ICSTW.2014.5.