Fault model-based variability testing

Ivan do Carmo Machado[1], **Eduardo Santana de Almeida**[1]

[1]Computer Science Department – Federal University of Bahia (UFBA)
Salvador – BA – Brazil

{ivanmachado,esa}@dcc.ufba.br

Abstract. *Software Product Lines (SPL) testing techniques are commonly focused on handling variability from a high level abstraction perspective, despite the importance of understanding the nature of issues emerging from source code that could affect the overall quality of products. In this investigation, we present a framework aimed to handle such a neglected issue by augmenting an SPL testing process with fault modeling support. Fault modeling is an strategy employed to capture the behaviour of the system against faults. By understanding the nature of faults before developing the tests might improve the likelihood of finding particular classes of errors. The proposed framework encompasses test assessment, to evaluate the effectiveness of existing test suites, and test design, by focusing on fault-prone elements. We carried out a controlled experiment to assess the test effectiveness of the proposed framework. Software engineers from an industrial partner acted as subjects. The assessment has shown promising results that confirm the hypothesis that combining fault models in an SPL testing process performs significantly better on improving the quality of test inputs.*

1. Introduction

Software Product Lines (SPL) engineering has proved to be an important strategy to cope with the diversity of customer needs. Owing to the use of mechanisms to implement variability, instead of offering a single product as a compromise of the varying needs, SPL engineering enables companies to offer several products with slightly varying capabilities. Instead of developing products independently, the products of a product line are developed by reusing existing product line assets in a prescribed way; these assets include software components, requirements, test cases, and other reusable artifacts [Clements and Northrop 2001].

A required activity in SPL engineering is to ensure that an artifact holds an adequate level of quality, as it is likely to be used in a range of different product configurations [Pohl et al. 2005]. In this effect, SPL engineering demands cost-effective quality assurance techniques that attempt to minimize the overall effort, while improving, or at least not worsening, fault detection rates. There are many well-known quality assurance techniques such as reviews or testing. However, in industry, software testing is still the most prevalent quality assurance technique [Ammann and Offutt 2008].

Although not as mature as in single-system development, software testing has evolved in SPL engineering, in a range of topics [Machado et al. 2014a]. The demand for specific SPL testing approaches has led the research community to increasingly propose novel techniques, methods, and tools, as earlier discussed in [Neto et al. 2011]. When analyzing existing literature on SPL testing [Machado et al. 2014a], we might observe

two main *groups of interests*: ***(i)*** **Selecting instances of products for testing -** verifies if the features of a product work properly together. It handles how the assets can be combined so that valid products can be composed; and ***(ii)*** **Testing actual products -** verifies if the features fulfill their specifications, i.e., it deals with the actual functionalities of built assets.

Both interests are important and one should not advance completely towards one overlooking the other. Notwithstanding, in practice it is not always observed. The research community has largely investigated the former, with a range of formalized techniques, some of them even including strong evidence from large-scale SPL projects [Machado et al. 2014a]. In the latter, techniques do not take into consideration fine-grained variability issues, when desigining testing assets. The overall focus has been on modeling variability in the problem domain, leaving aside the variability support at the solution domain [Machado et al. 2014b].

We might enlist two main likely reasons to explain this phenomenon: *(i)* the underlying assumptions of most SPL testing techniques is that, handling tests at source code level is a straighforward task, and as such, techniques from single system development suffices; and *(ii)* the challenge of handling variabilities is more easily addressed at higher levels of abstractions (e.g., through models), rather than at low levels (i.e., at source code). Given that models are more abstract and less detailed than source code, variability is likely less scattered, what simplifies its management.

However, while such a simplification shall facilitate the understanding of how to compose features in order to build a product instance, it is rather important to manage variability at source code level, given that it holds an important role in establishing variable behavior. Indeed, no empirical evidence can be found in the literature to ensure that such a statement is not a simple guess [Machado et al. 2014a, Neto et al. 2011].

By considering existing testing support for SPL engineering, and the SPL demands, the central problem we seek to address in this investigation is the *lack of adequate support for the low-level variability testing in SPL engineering*. More than verifying whether the features fulfill their specifications, or whether the features of a product work properly together, what usually do not have to do with source code, but high-level models instead, the challenge is to establish an understanding on how testing an SPL could benefit from the variability awareness. That is, not only using variability to define high-level tests, but also to deal with the source code particularities, so as to enable better fault coverage still while developing the artifacts.

In this sense, we developed a framework for building fault models to support testing in SPL engineering, surrounding low-level variability implementation issues. The goal is to establish an affordable strategy to design effective test cases that prioritize the fault-prone parts in a system implemented relying on different variability mechanisms. In general, a fault model is an engineering model of something that could go wrong in the construction or operation of a piece of equipment, structure, or software [Martin and Xie 2007]. In SPL testing, the objective is to modeling source code statements that could be faulty when implementing variations in an SPL project.

2. Background

Experienced software engineers, when designing test sets, may have in mind where the *hotspots* in the project are. *Hotspots* are herein considered as the points in the software where most problems are likely to occur, and as a consequence, where most effort should be expended. In this sense, while it is possible that test design may be carried out without any fault modeling support, it is not true that a common testing strategy does not involve fault models, at least implicitly.

It is rather important to build models that reflect the occurrence of faults, so that testing activities can be planned more realistically and effectively. In line with information on the nature of faults, another testing strategy may emerge. It consists of *fault modeling*. It is tied to the idea of building fault dictionaries to augment the efficacy of testing, thus improving the overall quality of the delivered software.

Due to the complexity present in the cause-effect relationship of faulty software components, it becomes apparent that a fault model has a broader meaning. In this work, we consider that a fault model is *a description of the behavior of, and assumptions about, how components in a faulty system behave*. A fault model provides testers with specific fault types for which to search based on the types of technologies used and the activities that have preceded the tests, such as requirements specification, design, and implementation. Tests or review scenarios are written to search for each possible type of fault.

A fault model describes the space of erroneous behaviors which can be expected as a result of an error. Such a description consists of a fault list or dictionary. From the fault list/dictionary, faults can be selected, and test inputs can be developed [McGregor 2008]. It is an effective means to design test cases that have a high probability of revealing faults [Martin and Xie 2007]. The tester with access to a fault model and the frequency of occurrence of fault types could use this information as the basis for generating fault hypotheses and test cases, aiming at building a minimal complete test suite.

Determining the frequency occurrence should rely on the extensive analysis of historical data, encompassing a collection of experience, in terms of common faults, about the scenario under analysis in a range of criteria, e.g., application domain, programming language, structural complexity of the software, etc. In this effect, it is unlikely that a generic fault model may emerge, that could be used in all scenarios. Fault models are usually dependent on particular domains. However, when understanding the issues of a particular domain, the interest is on the behaviour of the system under various common faults. While the fault model support cannot guarantee the absence of a specific type of fault, it can be used in assurance arguments that specific procedures have been used to search for specific faults.

A measurement of the efficiency of a fault model can be defined as the ratio of existing actual faults covered by testing the modeled faults. As more thorough fault models need higher test effort, because more possible faults have to be considered, a trade-off between quality and cost of a test must be found in practice.

2.1. Fault models in the Software Development Life Cycle

Different fault models helps detecting defects that were introduced during different phases, so that they could be fixed in the earlier stages of the projects avoiding rework.

Thus, a fault management process could be defined as a means to determine where faults are found and where they are introduced.

A template for a development phase description can be defined to support the process. It comprises four elements of interest for the development phase, as follows:

- **Faults expected** - When defining a process each phase will propagate some faults on to the next phase. The activities in the second phase should be created to identify and handle these faults.
- **Faults eliminated** - The activities in each development phase may eliminate faults that have been previously introduced.
- **Faults introduced** - Each phase of development has the potential to inject faults into the product or its supporting artifacts. The nature of the phase determines what is possible.
- **Faults propagated** - Faults that are either anticipated or introduced must either be eliminated or they will be passed on to the following phase.

Artifacts created at one phase are passed on to, and used by, later phases. As a consequence, any fault injected and not detected by the verification activities in that particular phase is still in the artifact when it is used by the later phase. This action is referred to as *fault propagation*.

When a fault is injected into a development artifact, such as architecture or program code, that fault remains until it is recognized and removed. As the development proceeds and the artifacts are used by later phases, a fault may cause errors that result in additional faults being created in other artifacts. For example, a faulty requirement may result in a fault in the architecture and several faulty test cases thereupon.

The Verification & Validation (V&V) activities of a development process are intended to identify those faults. These activities should be planned with specific faults in mind. Developing the V&V plan for a project is the point at which specific fault types related to the technologies being used are mapped to the development process. This is the point at which fault modeling becomes particularly interesting. At each phase, the V&V activities will be defined in conjunction with the fault models. A process definition will be a composition of phase definitions, as Figure 1 shows. The V&V activities in each phase are the first line of defense. Expected fault types and introduced fault types are searched for. Figure 1 depicts a classical waterfall process only for the purpose of presentation, as a more iterative and incremental approach is more encouraged.

3. Conceptual Framework

SPL testing is focused on maximizing the quality of products delivered to customers while reducing the cost of testing. Associated to the idea that a single problem within a feature may have on the set of products that make use of it, the ultimate goal of an SPL testing strategy is to avoid retesting feature in every single product instance derived from the SPL. Thus, we may weight the importance of keeping features, both individually and in conjunction, at an adequate level of quality.

Recall the generic fault model-based process earlier introduced in Figure 1. It is possible to incorporate fault models in each SDLC phase, in order to guide the design of artifacts, having in mind the likely mistakes an engineer can make, when in charge of such

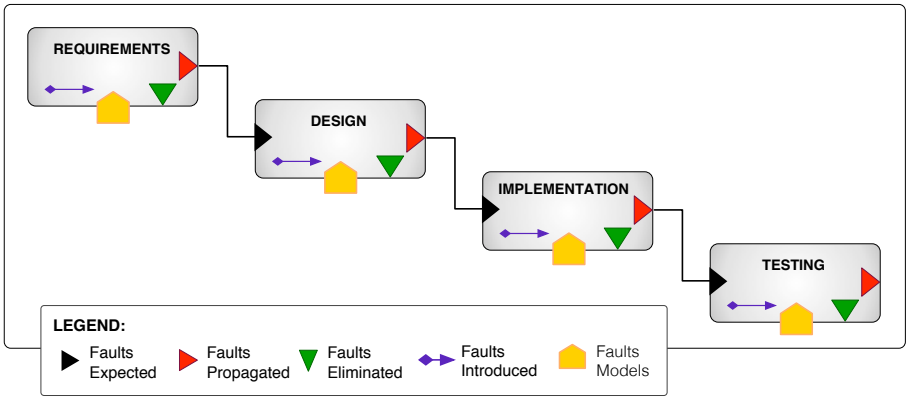

Figure 1. Software development life cycle enhanced with fault model support.

a task. The goal is to minimize the amount of faults that are propagated from one phase to another. Indeed, later phases will also search for those faults that are expected to be propagated from earlier ones. However, this is where the exponential problem comes into play. Considering that a specification has a problem that had not been identified before leaving to the design phase, and such a problem was then modeled as an architectural design, and, the problem earlier introduced in the specification phase had not been identified again. Proceeding in the life cycle, the implementation is going to further materialize this problem, that much probably will only be identified when testing is carried out.

Notwithstanding, while a complete fault model approach should consider all the SDLC phases, the approach proposed in this present investigation narrowed down the focus to investigate the problems that are likely to occur when implementing either the source code or the test scripts. The preceding phases, such as requirements and design, as well as other important SPL-related concerns (product derivation, process and management aspects) are left aside in this investigation. The literature on SPL engineering has provided researchers with a large set of explicit activities and operations to follow in the formulation of both specifications - comprising features and requirements, e.g., [Souza et al. 2013], and design - encompassing activities to assess the quality of the product line architecture, e.g., [Nakagawa et al. 2011].

Figure 2 shows the overall SPL testing schema, enhanced by the fault modeling support. It shows the relationship between the fault models and the test process. The solid arrows represents a relationship among test tasks (design, execution, and reporting), and between test phases and other elements (design, source code, fault models, test cases, and knowledge base), highlighting which ones provide input to others. There are two more kinds of relationships, represented by dashed arrows: *(i) produces*, that particularly indicates that *test cases* are produced by the task *test design*, and *(ii) uses*, indicating that some test phase/element makes use of any other element.

The fault models, by means of their associated fault dictionaries, will be mainly used to support the design of *test cases*. The dictionaries pinpoint which faults are more likely to occur, given a particular *variability mechanism*, involved in the *source code implementation*. To this end, fault models should be aware of variability mechanisms used in the project.

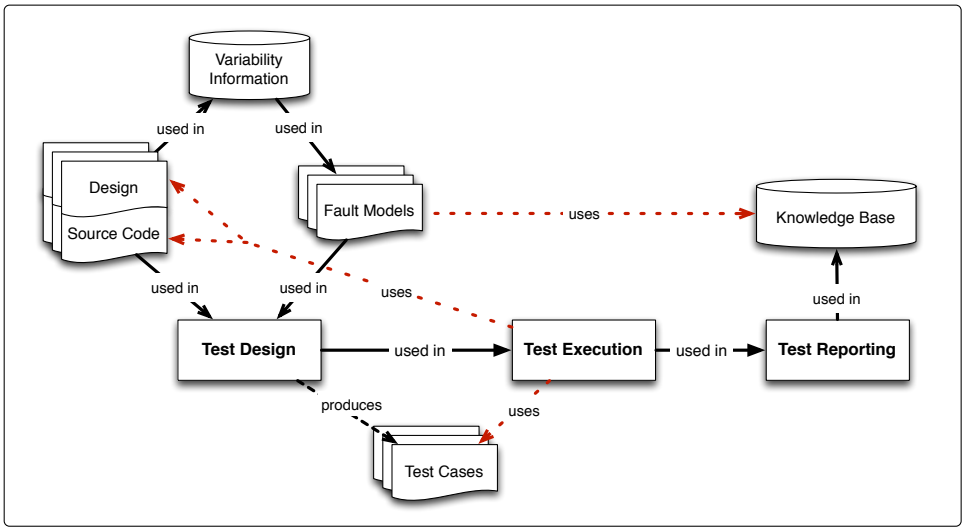

Figure 2. Overview of the SPL testing schema.

As a living document, as new faults are found that have not been listed in the fault model, an analysis on the problem should be undertaken, so as to enable its incorporation. The feedback arrow from the *test reporting* task to the *knowledge base* repository illustrates such a task. The idea behind a *knowledge base* comes from the fact that all fault models can evolve to include novel constructs. It might be compared to design patterns [Gamma et al. 1995], where known problems are modeled to avoid repeating the problem. Likewise *design patterns*, in which developers are provided with usage scenarios, every fault model is expected to accomplish a real application scenario.

A fault model contains historical data about commonly occurring errors, with data coming from two main sources: *(i)* data from the same project under evaluation, in case past test runs have provided the knowledge base (c.f. Fig. 2) with appropriate feedback, or *(2)* from a historical database, which encompasses knowledge from other projects implemented in similar conditions, in terms of project domain, size, programming language, etc. All such information might influence the fault distribution and occurrence, and should be taken into account accordingly.

3.1. Fault modeling for test suite evaluation

In this perspective, we assume that an SPL project already comes with a test suite. As a consequence, every product instance may comprise a subset of the SPL test suite. Considering we are strictly working in testing variability mechanisms implemented in Java programming language, we could consider test suites as a set of test scripts, implemented in any widely used test automation framework such as the JUnit.

This first perspective is aimed at employing fault models to evaluate the effectiveness of the existing test sets.

Let S be a set of N tuples $\{(F_0, T_0), (F_1, T_1), ..., (F_{n-1}, T_{n-1})\}$, where F is the set of features and T the set of test suites of an SPL, and F_i and T_i are valid subsets of F and T, respectively. A program $P_i \in S$ is a runnable instance of this SPL. P_i is not an actual product instance, but it is rather a valid subset of features $\Phi \subseteq F$, or even a single feature

$f_j \in F$, that can be tested as an isolated instance. Hereinafter, each P_i will be referred to as a program i under test (PUT_i).

Each feature $f_j \in F$ may be associated to a set of tests $t_j \in T$. Let R be a subset of T, then there is a function $X(f_j) : F \rightarrow R$, which represents the set of test cases that are suitable to a feature f_j. Each feature f_j also holds information about the variability mechanisms $VM = \{vm_0, vm_1, ..., vm_n\}$ employed to implement variability in the feature. As we intend to take into account variability information, we consider that a variability implementation mechanism $vm \in VM$ can be associated to a set of fault models $FM = \{fm_0, fm_1, ..., fm_p\}$, so that $Y(vm) : VM \rightarrow S_{FM}$, where S_{FM} is a subset of FM.

A fault model fm_i subsumes information about fault types to search for when testing a program P which uses it. Building a fault model consists of analyzing historical data to understand which fault types, and associated faults, are occurring in a given variability mechanism, and in which frequency range.

Now, let S be the program specification S represented schematically as $S \vdash \{\forall inputs, \exists output \mid spec(input, output)\}$, where $input$ is a vector of arguments, $output$ is an expected result and $spec$ is a proposition function describing the required relation between them. Hence, a program P under test (PUT) is a 3-tuple (X, Y, S). Testing P consists of checking that the behavior of an implementation, its *actual output* is conform to its specification, namely its *expected output* (the $output$ from function $spec$ above), given a set of $inputs$.

Hence, to carry out a test evaluation, a subset of features Φ will be selected. Each $f_i \subseteq \Phi$ is associated to a set R of test cases. These are *developer tests*, i.e., those developers design to test their code as they write it, as opposed to the tests carried out by third-party testers. Developer testing, often in the form of *unit* or *integration* tests, helps developers to both gain high confidence in the program unit (e.g., a class) under test while they are writing it, and reduce fault-fixing cost by detecting faults early when they are freshly introduced in the program unit [Xiao et al. 2012]. However, as we are dealing with features composed of one or more classes, we also consider the integration test level. While a unit test is usually a sequence of method calls on an object instance, therefore the main components are method and constructor calls, an integration test involves interfaces between components, and, from a broader perspective, the likely interaction between features, whenever a feature depends upon calling an external object.

Furthermore, as each $f_i \subseteq \Phi$ holds information about the variability implementation mechanism, then let $faultModels(vm)$ be a function that returns a list containing the classes of errors of the fault models that are appropriate to the mechanism vm used to implement f_i. Besides, let $\tau(P, R)$ be a function that executes test cases $t_i \subset R$ on program P against the specifications S and returns the outcome of the test execution.

In the assessment schema, we include a task called *fault injection*, and a repository called *mutation operators*. We can use a set of mutation operators to describe all expected errors, and therefore defines the behavioral fault model. The proposed approach does not make distinctions, for the time being, between *static* and *dynamic* fault injection. Given that we combine mutation analysis and fault injection techniques, we could be inclined to narrow down the focus to the former. However, it is important to mention that a wide

variety of faults listed in the fault models can be dynamically emulated as well.

Given a set of test cases T, the fault models may indicate some mutation operators to program P to produce a modified version, a mutant P'. A set of representative faults, suggested by the fault models, are injected into the code of P. Hence, let function $\tau(P, T)$ be executed. Then, it will be possible to measure the adequacy of test cases, i.e., a test case is *adequate* if it is effective at detecting faults in the program [Offutt et al. 2001].

The mutants are run with $input$ data from a given test set T. If a test set can distinguish a mutant P' from the original program P, i.e., it produces a different $output$, the mutant P' is said to be killed. Otherwise, the mutant is called as a live mutant. That is, if after modifying the source code, with the set of mutatns, the same output is observed, it means that the test cases are not *adequate* enough. Conversely, a test set which can kill all non-equivalent mutants is said to be *adequate*. That can be explained by the mutant score calculation, as follows.

Mutation score $ms(P, T)$ is defined as the ratio between the number of mutants detected and the total number of mutants minus the equivalent ones [Jia and Harman 2011]. A mutant is said to be equivalent if it syntacticaly differs from the original program, but semantically the mutation can not be detected. A test set T is *mutation adequate* if its mutation score is 100%. The score can be calculated as follows:

$$ms(P, T) = 100 * \frac{DM(P, T)}{MT(P)}$$

where $DM(P, T)$ is the number of mutants killed by T; and $MT(P)$ is the total number of mutants generated from P.

Fault models are expected to increase the probability of finding a given fault as the associated metric. However, this attribute should really reflect the percentage of faults that the technique can detect.

3.2. Fault modeling for test suite design

The second perspective encompasses the fault modeling support for test design purposes. Relying primarily on a prioritization strategy, test design seeks to identify *what to look at prior to testing*.

There exists a bunch of formalized test prioritization techniques, as deeply discussed in [Catal and Mishra 2012]. Most of them are concerned about establishing effective means to improve test case selection aimed at regression testing. They are usually concerned about analyzing past historical data about modules structure (e.g., size, coupling, etc), bugs reported, bug fixing modifications, and general maintenance modifications. Based on such data, some heuristics are calculated that enable prioritization.

Our goal is not to propose a novel test case prioritization strategy. Indeed, this perspective aims at prioritizing the units for testing, not the test cases themselves, as they do not exist yet.

Figure 3 shows an overview of the test design schema. There is a task called *prioritize units*. We followed the prioritization principles first introduced in [Rothermel et al. 2001]. The heuristic defines that each unit in the source code will be

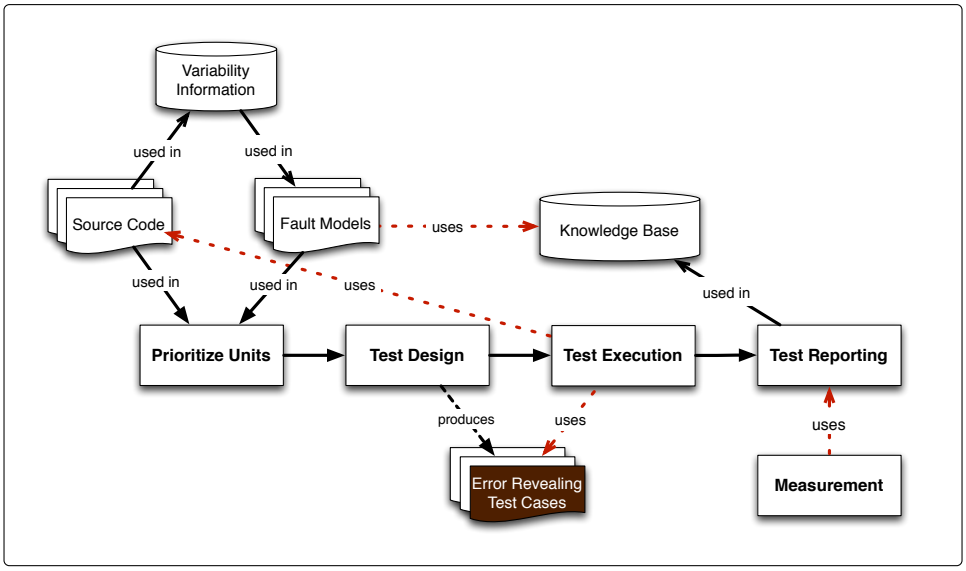

Figure 3. Schema for test set design.

attributed to a *weight*, namely an integer value. In this sense, a feature f_i is composed of a set of units $u_i \in U$, where U is the set of all implementation units of the SPL. Following the ideas of generating unit tests described by [Fraser and Zeller 2012], and tailoring it to encompass integration concerns, as it may illustrate how variable entities should interact with each other, let a unit be a 6-tuple $u = (cs, ms, fs, ps, ci, ce)$, with all values of type integer, where:

- cs is the number of *constructor statements* in a unit;
- ms is the number of *method statements*;
- ps is the number of *primitive statements*;
- ci is the number of *calls to internal units*, i.e., units from the same feature;
- ce is the number of *calls to external units*, i.e., units from other features.

The prioritization strategy consists of analyzing every attribute and weighting every unit. We leverage a feature weight by summing all the values of each u element. The principle is that the element with maximum weight is taken first, followed by the element with the second highest weight, and so on. The running example presented in the next Section illustrates how it can be done.

4. Evaluation

This section describes a controlled experiment aimed at **evaluating the strength and significance of the proposed fault modeling framework**. We analyzed how the fault models could support the testing of variable features in an SPL project. The experiment was carried out in an industrial setting, involving seven experienced software engineers, all of them familiar with the development of variant-rich software systems.

4.1. Experiment planning

This experimental study focused on answering the following research questions:

Table 1. Hypothesis formulation

Null Hypothesis	*Alternative Hypothesis*
$H_{01} : \mu_{TCE_{FM}} \leq \mu_{TCE_{AH}}$	$H_{11} : \mu_{TCE_{FM}} > \mu_{TCE_{AH}}$
$H_{02} : \mu_{TCov_{FM}} \leq \mu_{TCov_{AH}}$	$H_{12} : \mu_{TCov_{FM}} > \mu_{TCov_{AH}}$
$H_{03} : \mu_{MS_{FM}} \leq \mu_{MS_{AH}}$	$H_{13} : \mu_{MS_{FM}} > \mu_{MS_{AH}}$

Legend: AH - adhoc (without the support of fault models) — FM - fault model supported test design

- **RQ1. Does the use of fault models lead to best SPL testing results?** This question aims at investigating whether the fault model solution is worthwhile to be used by practitioners.
- **RQ2. Is the fault modeling approach helpful to uncover the prescribed faults?** Not only finding more errors, but finding those that were actually pointed out by the fault models might be another measure of effectivenes.

We applied the *Mutation Score **(MS)*** calculation, to assess whether the fault models help identifying the expected fault types, as earlier explained, and also the *Test Case Effectiveness **(TCE)***, *Test Coverage **(TCov)*** metrics [Machado et al. 2012a], as explained next.

TCE: The more defects test cases find, the more effective they are. It is defined as the ratio of defects found by test cases to the total number of defects reported during a test cycle. We tailored such measure to our context, so that (TCE) is defined as the ratio of the amount of defects (D_{tot}) reported to the total number of test cases (N_{tc}). This value provides insights on the effectiveness of functional test cases. TCE calculations can be defined as: $TCE = \frac{D_{tot}}{N_{tc}}$.

TCov: It gives the fraction of all code elements covered by a selected number of test cases or a complete test suite. We assume C_{ov} as the *basic blocks* coverage generated by the Eclemma[1] code coverage tool.

Hypotheses. The use of fault models to anticipate *hotspots* for testing is assumed to yield better fault coverage, improving the software testing activity. Thus, we formalized the definition of the *null* and *alternative* hypotheses that drive this investigation. Table 1 shows both the null (H_{0n}) and alternative (H_{1n}) hypotheses.

Variables. As **independent variables** we considered the designed test cases and the background experience of the participants. The **dependent variables** are the set of uncovered faults.

Subjects. The subjects were chosen based on *convenience* sampling. We randomly selected a set of software engineers from a partner company from Salvador, Brazil[2], a 15-year-old software house, mostly working in the domain of learning environments and educational systems.

[1] `http://www.eclemma.org/`

[2] Recôncavo Institute of Technology. `http://reconcavotecnologia.org.br/`.

We randomly selected seven software engineers, experienced in software development with Java programming language. All of them had at least three years of industry experience. We provided them with training sessions on the topic, in which they could learn from us how to systematically reuse their developed software artifacts, and next, some of them were selected to serve as subjects in this evaluation.

Instrumentation. The instruments used are the consent form, background and feedback questionnaires, and the source code and documentation of the SPL project under evaluation. We used a Java-based SPL, called *PL_SimElevator*. The project consists of an SPL aimed at simulating the operation of an *elevator controller*. Variability in the SPL was implemented using the following variability mechanisms [Svahnberg et al. 2005]: *inheritance, polymorphism* and *encapsulation*. The project comprises over 3,500 lines of code, 12 packages, 32 classes, and 11 optional features. From this SPL, distinct variants of elevator controllers can be generated, to meet different product configurations. Some of the variants are bound at compile-time, while others necessarily at runtime.

Design. We employed a completely randomized design, with *one factor with two treatments*. We compared the two treatments against each other, namely test design with and without the support of fault models. The participants were randomly arranged into groups. The control group applied software testing techniques they commonly use in their daily tasks as software developers (unit and integration tests). The experiment group used the fault lists/dictionaries from the fault models available to them.

4.2. Experiment operation

The participants were all given an 1-hour-introductory lecture.The participants also had a 3-hour-introductory lecture on SPL engineering. In addition, a 4-hour-training session was undertaken, when they could become familiar with the SPL project. All participants had the opportunity to practising their own test design strategy.

During the experiment session, the participants had about four hours to complete the task, which consisted of designing and implementing JUnit test cases.They had to analyze the source code, and implement unit and integration test cases from the source code.We intended to simulate a real testing environment, hence the participants were told to implement both *unit* [Machado et al. 2011] and *integration* tests [Machado et al. 2012b]. These are usually tests a *developer* implement to either execute a specific functionality in the code (unit) or test the behavior of a component or the integration between a set of components (integration). The participants could implement mocks whenever a method depended on other parts of the system, not available to them.

The participants had to design test cases to handle the SPL. In this sense, their must cover the more variation points they could, from the set of packages previously mentioned. The idea was to design the test cases thinking of their further reusability. The test effectiveness would be measured by considering the capability of a test case to be reused in other product instances, and also its capability of uncovering defects. Hence, they were recommended to design domain test cases, to cover commonalities and proceed with the capabilities of each variability mechanism to enable further specializations when a variant was present.

Fault injection. In order to measure the effectiveness of designed test cases, we emulated faults in the source code.They represent some of the commonly occurring

problems when implementing variability using Java constructs. The selected modules represent suitable locations for emulating some of the desired types of faults. It means a fault was identified and classified according to an operator library of type of faults. Indeed, the faults should only be emulated where they could actually exist in the compiled code, considering the logic of programming structures.

4.3. Analysis and interpretation

This section presents the statistical analysis of the gathered data, collected from the test cases implemented by the participants, and the defect log report they filled out during the experiment session. Each defect found in each log was analyzed to check whether they represented an actual defect. False positives were discarded from the final analysis.

4.3.1. Does the use of fault models lead to best variability testing results?

By applying a randomized design, we assigned the participants to two different groups: *Group A* (ad-hoc testing): **P3, P4, P5, P6**; *Group B* (fault model support): **P1, P2, P7**. As we observed an expertise homogeneity, and especially for the small number of participants, we did not make the effort to achieve an even distribution of participants.

The results indicate a higher *mean* value for the **TCE** calculations for the group which employed the notion of fault models for test case design: 0.504 with a *sd* of 0.182, against 0.343, with a *sd* of 0.078. When the participant implemented a test case, he should have in mind its reuse potential. That is, a test case aimed at a class c can be reused in any product configuration containing such a class. Hence, the results may demonstrate the reuse capability of the set of designed test cases.

The descriptive statistics for each product is as follows. **Premium**: Group A (*mean*: 29.525, *sd*: 19.145), Group B (*mean*: 36.500, *sd*: 22.035); **Enhanced**: Group A (*mean*: 28.450, *sd*: 18.034), Group B (*mean*: 26.167, *sd*: 16.614); **Simple**: Group A (*mean*: 22.000, *sd*: 14.288), Group B (*mean*: 23.833, *sd*: 37.302). Group B reached better mean for the first two products, while Group A for the third one. It may indicate that the more variability the source code includes the more useful the fault model can be.

4.3.2. Is the fault modeling approach helpful to uncover the prescribed faults?

The ultimate goal of using fault models is to reduce the number of test cases designed. Besides, the idea is to uncover the more defects with less test cases. In order to investigate the effects of using a fault modeling approach, we calculated the accuracy of the fault models at uncovering the fault types they were expected to. By analyzing the *mean* values of groups A and B, respectively 6.675 (*sd: 2.735*) and 17.800 (*sd: 1.905*), the group of participants using a fault model had better results.

4.3.3. Hypothesis testing

The hypotheses were tested using a standard paired t-test with a 95% confidence level. We calculated the t-test to compare the two treatments against all metrics. Table 2 shows

Table 2. t-test results.

metric	t	df	p-value	n
TCE	1.434	2.549	0.262	
$TCOV$ Premium	0.438	4.042	0.684	
$TCOV$ Enhanced	0.173	4.667	0.870	7
$TCOV$ Simple	0.081	2.445	0.942	
ms	6.338	4.999	0.001	

the results for all three metrics. We may observe a significant difference between the ms score of both groups A and B. It may indicate that programmers using fault models can achieve higher accuracy on finding the faults they are searching for. Such an observation enable us to refute the null hypothesis H_{03}. However, the same does not hold true for the remainder hypothesis. Although descriptive statistics showed better results for the group undertaking the task with the fault model support, we have not found evidence of a statistically significant difference for the hypotheses H_{01} and H_{02}, what prevent us to make any conclusions about such findings.

4.4. Threats to Validity

A general threat to *conclusion validity* observed in the experiment is the small number of samples, which may reduce the ability to reveal real patterns in the data as well as to establish the true strength of the relationship between variables. Such an observation may also reduce the statistical power. The analysis and interpretation of the results was described using descriptive statistics and a parametric test was employed. While the choice of a nonparametric statistic over the parametric one should be used, especially for the small sample, the literature indicates the choice of the latter as being more robust considering the data type collected in the experiment [Briand et al. 1996], and the kind of analysis to be applied. It is worth mentioning that no extreme violation of the assumptions was identified in the collected data.

Regarding *internal validity*, the instruments used in the experiment may emerge as a source of a discussions. We should consider the project employed in the analysis as a factor that may have affected the results. For instance, the source code might contain an additional set of issues, other than those injected faults (expected to be uncovered by the tests), so that those were also considered as valid defects. Regarding subjects selection, as we carried out the experiment inside a software company, with software engineers we were not aware of, prior to the study, we mitigated such a likely threat.

In our observation, we could consider two main threats that may affect the *construct validity*. The first threat is that the measurements as defined may not be appropriate. Besides, the pre-chosen fault models may not be representative or good enough for the scenario under testing. These may limit the scope for the conclusions made to the use of the fault modeling approach.

Besides, despite we counted on experienced software engineers as subjects, their small number may limit generalisations of findings. This may pose a threat that affects the *external validity*. However, as the study settings resembled a real testing scenario, the

inferences about the use of fault models in SPL testing might be strengthen.

5. Concluding Remarks

Most existing SPL testing techniques are focused on handling variability testing at high level of abstraction, e.g., through the analysis of feature models. Conversely, in this investigation we focused on issues emerging at source code level. That is, we handle variability testing at a lower level of abstraction, where the implementation mechanisms are used consistently.

The contribution of this investigation consists of designing a framework to build fault models capable of providing SPL testing with an adequate support, from two main perspectives: *test assessment* - focused on the evaluation of the effectiveness of existing test suites; and *test design* - focused on the construction of test sets, by leveraging fault-prone elements. The goal of employing fault models is to support test case design with a prioritization strategy, to focus on the *hotspots* in the source code, and consequently anticipate the likely faults. Another role a fault model may perform is to evaluate existing test suites effectiveness.

As a means to evaluate the capability of the proposed framework, we carried out an empirical study, in which experienced software engineers could make use a set of fault models to design a set of developer tests. The evaluation showed promising results. We understand that there are many promising improvements opportunities. Hence, as future work, we plan to investigate aspects of automation, addressing challenges of generating desirable test inputs and checking the behavior of the features under test. In addition, we plan to carry out empirical studies to better understand the role of fault models in other domains and scenarios.

References

Ammann, P. and Offutt, J. (2008). *Introduction to software testing*. Cambridge University Press.

Briand, L., Emam, K., and Morasca, S. (1996). On the application of measurement theory in software engineering. *Empirical Software Engineering*, 1(1):61–88.

Catal, C. and Mishra, D. (2012). Test case prioritization: a systematic mapping study. *Software Quality Journal*, pages 1–34.

Clements, P. and Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, USA.

Fraser, G. and Zeller, A. (2012). Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 38(2):278 –292.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

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

Machado, I. C., Almeida, E. S., Gomes, G. S. S., Neto, P. A. M. S., Novais, R. L., and Neto, M. G. M. (2012a). A preliminary study on the effects of working with a testing

process in software product line projects. In *IX Experimental Software Engineering Latin American Workshop*, ESELAW, Buenos Aires, Argentina.

Machado, I. C., McGregor, J. D., Cavalcanti, Y. C., and Almeida, E. S. (2014a). On strategies for testing software product lines: A systematic literature review. *Information & Software Technology*, 56(10):1183–1199.

Machado, I. C., Neto, P. A. M. S., and Almeida, E. S. (2012b). Towards an integration testing approach for software product lines. In *IEEE 13th International Conference on Information Reuse and Integration (IRI)*, pages 616–623.

Machado, I. C., Neto, P. A. M. S., Almeida, E. S., and Meira, S. R. L. (2011). RiPLE-TE: A Process for Testing Software Product Lines. In *Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering (SEKE)*.

Machado, I. C., Santos, A. R., Cavalcanti, Y. C., Trzan, E. G., Souza, M. M., and Almeida, E. S. (2014b). Low-level variability support for web-based software product lines. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM.

Martin, E. and Xie, T. (2007). A fault model and mutation testing of access control policies. In *Proceedings of the 16th international conference on World Wide Web*, WWW, pages 667–676, Banff, Alberta, Canada. ACM.

McGregor, J. D. (2008). Toward a fault model for software product lines. In *Proceedings of the 12th International Conference on Software Product Lines*, SPLC, pages 157–162, Limerick, Ireland. IEEE Computer Society.

Nakagawa, E. Y., Antonino, P. O., and Becker, M. (2011). Reference architecture and product line architecture: A subtle but critical difference. In *Proceedings of the 5th European Conference on Software Architecture*, ECSA, pages 207–211.

Neto, P. A. M. S., Machado, I. C., McGregor, J. D., Almeida, E. S., and Meira, S. R. L. (2011). A systematic mapping study of software product lines testing. *Information & Software Technology*, 53(5):407–423.

Offutt, J., Alexander, R., Wu, Y., Xiao, Q., and Hutchinson, C. (2001). A fault model for subtype inheritance and polymorphism. In *12th International Symposium on Software Reliability Engineering (ISSRE)*.

Pohl, K., Böckle, G., and van der Linden, F. J. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag.

Rothermel, G., Untch, R. H., Chu, C., and Harrold, M. J. (2001). Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948.

Souza, I. S., Gomes, G. S. S., Neto, P. A. M. S., Machado, I. C., Almeida, E. S., and Meira, S. R. L. (2013). Evidence of software inspection on feature specification for software product lines. *Journal of Systems and Software*, 86(5):1172 – 1190.

Svahnberg, M., van Gurp, J., and Bosch, J. (2005). A taxonomy of variability realization techniques: Research articles. *Software Practice & Experience*, 35(8):705–754.

Xiao, X., Thummalapenta, S., and Xie, T. (2012). Advances on improving automation in developer testing. *Advances in Computers*, 85:165–212.