Spectrum-Based Fault Localization in Software Product Lines

Aitor Arrieta[a], Sergio Segura[b], Urtzi Markiegi[a], Goiuria Sagardui[a], Leire Etxeberria[a]

[a]*Mondragon Unibertsitatea, Mondragon, Spain*
[b]*ETS Ingeniería Informática, Universidad de Sevilla, Spain*

Abstract

Context: Software Product Line (SPL) testing is challenging mainly due to the potentially huge number of products under test. Most of the research on this field focuses on making testing affordable by selecting a representative subset of products to be tested. However, once the tests are executed and some failures revealed, debugging is a cumbersome and time consuming task due to difficulty to localize and isolate the faulty features in the SPL.
Objective: This paper presents a debugging approach for the localization of bugs in SPLs.
Method: The proposed approach works in two steps. First, the features of the SPL are ranked according to their *suspiciousness* (i.e., likelihood of being faulty) using spectrum-based localization techniques. Then, a novel fault isolation approach is used to generate valid products of minimum size containing the most suspicious features, helping to isolate the cause of failures.
Results: For the evaluation of our approach, we compared ten suspiciousness techniques on nine SPLs of different sizes. The results reveal that three of the techniques (Tarantula, Kulcynski2 and Ample2) stand out over the rest, showing a stable performance with different types of faults and product suite sizes. By using these metrics, faults were localized by examining between 0.1% and 14.4% of the feature sets.
Conclusion: Our results show that the proposed approach is effective at locating bugs in SPLs, serving as a helpful complement for the numerous approaches for testing SPLs.

Keywords: Software product lines, spectrum-based fault localization, feature models, debugging.

1. Introduction

Software Product Line (SPL) engineering focuses on the systematic development of related software products from a set of reusable features [1]. A *feature* is defined as any increment in product functionality [2]. Features and their possible interactions are commonly depicted in a feature model. A *Feature Model (FM)* represents all the possible products of a SPL in terms of features and constraints among them [3]. In this context, a *product* is a set of features satisfying all the constraints of the FM. Figure 1 depicts a sample FM representing a simplified product line of mobile phones.

Most SPL testing approaches focus on deriving and testing each product individually [4, 5]. Since the number of potential products in a SPL is typically huge, several sampling techniques have been proposed to derive a manageable subset of products to be tested (e.g., [6, 7, 5]). Salient among them are *Combinatorial Interaction Testing (CIT)* techniques, whose goal is to select products where every combination of *t* features appears at least once, this is also called t-wise testing [8]. Another line of research addresses the problem of test case prioritization, where products are scheduled for testing in an order that attempts to increase their effectiveness at meeting some performance goal, typically detecting faults as soon as possible [9, 10, 11, 12]. Both strategies, sampling and prioritization, are complementary and are often combined.

Developing high-quality software requires not only effective testing methods to uncover failures, but also debugging techniques to locate and fix the bugs that trigger them. Debugging is mostly a manual process where testers must identify the defective code using techniques such as tracing, memory dumps or step-by-step execution. More sophisticated techniques include *Spectrum-Based Fault Localization (SBFL)*, which ranks code components (e.g., statements) according to their probability of having faults, so-called *suspiciousness* [13, 14, 15, 16].

Debugging SPLs is challenging due to the difficulty to find and isolate the faulty features in the SPL. Also, even if a suspicious feature or set of features are detected, it might still be difficult to generate small valid products (i.e., satisfying the constraints of the feature model) where the failure is reproduced and the defective assets can be pinpointed. Some works have proposed techniques based on machine learning to locate faults in configurable software in the past [17]. However, the recent advances on SPL testing contrast with the low number of studies that support SPL debugging, which remains a manual and time-consuming endeavour.

In this paper, we propose an approach to SPL debugging. The approach works in two steps. First, the outcomes of testing (test coverage and test outputs) are used to rank features according to their probability of having faults, so-called *sus-*

Email addresses: aarrieta@mondragon.edu (Aitor Arrieta), sergiosegura@us.es (Sergio Segura), umarkiegi@mondragon.edu (Urtzi Markiegi), gsagardui@mondragon.edu (Goiuria Sagardui), letxeberria@mondragon.edu (Leire Etxeberria)

piciousness score. The suspiciousness score of each feature (or set of features) is calculated using SBFL techniques adapted for the SPL domain. Then, a fault isolation approach is proposed to generate, by automatically analysing the feature model, products of minimum size containing the most suspicious features, in order to facilitate the isolation of the failure causes. A key contribution of our approach is the application of SBFL at the feature level (key atomic element in SPLs), rather than at the statement level, as in conventional SBFL approaches (e.g., [18, 13, 19, 14, 20]). In particular, we follow a model-based approach where the complexity of the code dependencies is managed through a simpler high-level representation of the features and the constraints among them: a feature model. This also permits to abstract the complexity of the underlying implementations such as the use of different programming languages or the combination of hardware and software features, e.g., cyber physical systems [21]. For the evaluation of the approach, we compared ten state-of-the-art SBFL techniques on nine SPLs of different sizes. Results reveal that SBFL performs well at locating faults in SPLs. More specifically, we found that three of the techniques under evaluation (Kulcynski2, Tarantula and Ample2) stand out over the rest, being able to localize the bugs by examining between 0.1% and 14.4% of the feature sets.

This paper is structured as follows. Section 2 presents general background related to SPL engineering and SBFL. Section 3 presents our approach for fault localization in SPL and the fault isolation algorithm. An empirical evaluation of our approach is performed in Section 4. Section 5 highlights the main issues that threatens our empirical evaluation. Section 6 positions our work with the current literature. Section 7 concludes the study and highlights future work.

2. Background

2.1. Feature models

Feature models (FMs) are the de-facto standard for modelling commonality and variability in SPLs [22, 3]. Structurally, a feature model is a tree-like structure in which nodes represent features and edges represent constraints among the features. A feature represents an increment in product functionality [2]. Each feature is related to a set of *assets* that implement the feature's functionality, i.e., code, documentation, test cases, etc. A *product* is a set of features satisfying the constraint of the feature model. Products are implemented by integrating the assets of the features that are part of them.

Fig. 1 depicts a sample feature model representing a SPL of mobiles phones. Child features can be divided into mandatory and optional features. *Mandatory* features must be included in all the products including its parent feature, e.g., all mobile phones in Fig. 1 must provide support for `Calls`. *Optional* features can be optionally included in those products containing its parent feature, e.g., phones can optionally provide support for `GPS`. Additionally, child features can be grouped into alternative and or relationships. A set of child features has an *alternative* relationship with their parent feature when only one of them can be selected when its parent feature is part of the product, e.g., phones can only support one type of screen: `Basic`, `Colour` or `High resolution`. Finally, in *or* relations at least one of the child features must be included in the products containing its parent feature, e.g., phones supporting media content must include the features `Camera`, `MP3` or both of them.

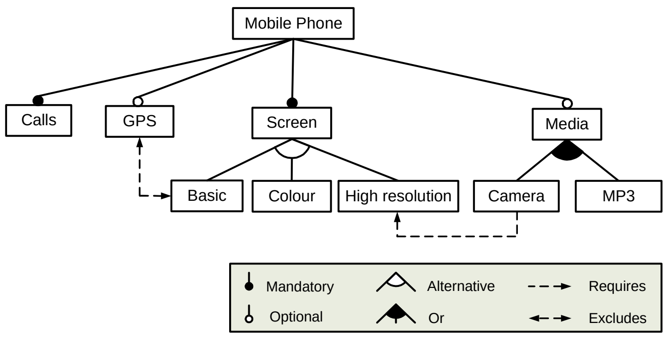

Figure 1: Example of a product line from the mobile phone industry [22]

In addition to the parental relationships among features, feature models can include cross-tree constraints among features. Typical constraints model dependencies such as "A *requires* B", indicating the products containing the feature A must also include the feature B, or "A *excludes* B", indicating that the features A and B cannot be part of the same product, i.e., they are incompatible features. In the example, phones including the feature `Camera` must include support for a `High resolution` screen.

The analysis of feature models deals with the automated extraction of information from feature models. The analysis is performed in terms of *analysis operations*. Among others, these operations allow finding out whether a feature model is void (i.e., it represents no products) whether it contains errors (e.g., dead features) or what is the number of products represented by the model. Catalogues with up to 30 different analysis operations on feature models have been reported in the literature [22]. A number of tools support the analysis of feature models including FaMa [23], SPLAR [24] and FeatureIDE [25].

In the following, we define some of the terms that will be used throughout the rest of the paper. For the definitions, let F be the set of features in a feature model.

- *Feature set.* Non-empty set of features S, $S \subseteq F$, with $|S| \geq 1$, e.g., S= {`Media`, `MP3`}.
- *Configuration.* A configuration is a 2–tuple of the form (S,R) such that $S, R \subseteq F$ being S the set of features to be selected and R the set of features to be removed such that $S \cap R = \emptyset$ and $S \cup R = F$. If $S \cup R \subset F$ the configuration is called *partial configuration* [22]. For instance, the following is a partial configuration of the model in Fig. 1: (S,R) = ({Media,MP3},{GPS}).
- *Product.* A product is equivalent to a configuration where only selected features are specified and omitted features are implicitly removed [22], e.g., see products in Table 1.
- *Product suite.* Set of products under test. Table 1 shows the set of products obtained when applying 2-wise test-

ing to the model in Fig. 1. The product suite is reduced from 13 products (total number of products in the SPL) to 8 products containing all the possible feature pairs, 41 in total.

- *Core features.* These are the set of features included in all the products of the SPL [22]. In the example the core features are `Mobile phone`, `Calls` and `Screen`.
- *Propagate operation.* This operation (also called dependency analysis operation [22]) receives a partial configuration as input, and it automatically selects and unselects the necessary features to create a valid product according to the constraints of the model (if such product exists). For example, suppose that we run the propagate analysis operation on the selected features {`GPS`, `Camera`}. The operation would propagate the decisions returning the product {`Mobile Phone`, `Calls`, `Screen`, `High resolution`, `Media`, `Camera`, `GPS`}. Notice that the product includes the core features, plus the features `Media` and `High resolution` (both required by `Camera`).

ID	Product
P1	{MobilePhone, Screen, Calls, High resolution}
P2	{MobilePhone, Screen, Calls, Colour, Media, MP3}
P3	{MobilePhone, Screen, Calls, Colour, GPS}
P4	{MobilePhone, Screen, Calls, High resolution, Media, MP3, Camera}
P5	{MobilePhone, Screen, Calls, High resolution, Media, Camera, MP3, GPS}
P6	{MobilePhone, Screen, Calls, Basic, Media, MP3 }
P7	{MobilePhone, Screen, Calls, Basic}
P8	{MobilePhone, Screen, Calls, High resolution, Media, Camera}

Table 1: Product suite (2-wise)

2.2. Spectrum-based fault localization

Spectrum-Based Fault Localization (SBFL) is a technique to assist on the location of program bugs [13, 26]. SBFL uses the results of test cases and their corresponding code coverage information to estimate the risk of each program component (e.g., statements) of being faulty. A *program spectrum* refers to a collection of data that provides a specific view on the dynamic behavior of a software program such as statement or branch coverage [13, 27]. Various forms of program spectra have been proposed [14]. For example, *block-hit* is a commonly used program spectra, where the program code is divided into statement blocks [26]. When SBFL with block-hit spectra is used, the result of the technique is an ordered list of code blocks sorted by their likelihood to cause the failure, so-called *suspiciousness score*.

Table 2 illustrates an example of SBFL with block hit spectra in a C program. To avoid confusion, we remark that Table 2 illustrates an example of SBFL, but in our case we do not apply this technique at the code level, but at the feature level; for further information of the application of SBFL at the feature level, refer to Section 3.1. Horizontally, the table shows the five code blocks in which the program has been divided, i.e., the components. Note that the code has a bug in block b_3. Vertically, the table shows four test cases of the program. For each test case (i.e., T_1, T_2, T_3 and T_4), a cell is marked with "•" if the program block of the row has been exercised by the test case of the column, creating what is known as the *coverage matrix* [18]. Additionally, the final row depicts the so-called *error vector*, which contains the outcome of each test case, either successful ("S") or failed ("F"). Based on this information, the suspiciousness score of each block can be calculated using more than 30 different techniques proposed in the literature [15]. One of the most well-known techniques to calculate the suspiciousness score is named *Tarantula*, which, for a program component (in our example a statement block), is computed as follows [26].

$$Suspiciousness(Tarantula) = \frac{\frac{N_{CF}}{N_F}}{\frac{N_{CF}}{N_F} + \frac{N_{CS}}{N_S}} \tag{1}$$

where N_{CF} is the number of failing test cases that cover the block, N_F is the total number of failing test cases, N_{CS} is the number of successful test cases that cover the block, and N_S is the total number of successful test cases. The suspiciousness score of each block is in the range [0,1], i.e., the higher the suspiciousness score of the block, the higher the probability of having a fault. The values of N_{CF}, N_{CS}, N_S, N_F and the Tarantula suspiciousness value of each code block are given in Table 2. The last column indicates the position of the statement in the suspiciousness-based ranking where top-ranked blocks are more likely to be faulty. In the example, the faulty block (b_3) is ranked first.

Suspiciousness techniques may often provide the same value for different components, being these tied for the same position in the ranking, e.g., blocks b_2, b_4, and b_5 in Table 2. Under this scenario, different approaches are applicable such as measuring the effectiveness in the best and worst scenarios, using an additional technique to break the tie, or using some simple heuristics such as alphabetical ordering [16].

3. Approach

In this section, we present a two-step approach for locating bugs in SPLs. First, SBFL-based techniques are used to calculate the suspiciousness of each feature set based on the testing outcomes, namely code coverage data and testing results (passes and failures). Second, the obtained suspiciousness scores are processed by a novel fault isolation approach to generate the smallest valid product containing the faulty feature set, helping to isolate the cause of the failure, and thus the bug causing it.

We may recall that this paper focuses on debugging and not testing. Thus, we assume the existence of a product suite (e.g., pairwise suite) and their corresponding testing results, obtained using any state-of-the-art testing technique, e.g., manual integration test cases. Note that a key requirement for the application of SBFL is that multiple failed and multiple successful test cases are available [16]. In what follows, our approach is

Table 2: An example showing the suspiciousness value computed using the Tarantula technique

ID	Program block	T_1	T_2	T_3	T_4	N_{CF}	N_{CS}	N_S	N_F	Suspiciousness	Ranking

b_1	int count n;
	Ele *proc;
	List *src_queue, *dest_queue;
	if (prio >= MAXPRIO) { /*MAXPRIO=3*/
		•	•	•	•	1	3	3	1	0.5	2

b_2	return;
	}
		•				0	1	3	1	0	3

b_3	src_queue = prio_queue[prio];
	dest_queue = prio_queue[prio + 1];
	count = src_queue->mem_count;
	if (count > 1) {
	/* **BUG: It should be if (count >= 1)** */
			•	•	•	1	2	3	1	0.6	1

b_4	n= (int) (count*ratio + 1);
	proc = find_nth(src_queue,n);
	if (proc) {
			•	•		0	2	3	1	0	3

b_5	src_queue = del_ele(src_queue,proc);
	proc->priority = prio;
	dest_queue = append_ele(dest_queue,proc);
	}
	}
			•	•		0	2	3	1	0	3

Execution results		S	S	S	F						

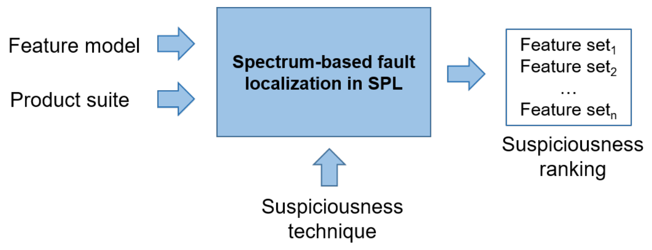

Figure 2: Overview of our approach on SBFL for SPL

described in detail, including the overall methodology for its application.

3.1. Spectrum-based fault localization in SPLs

We propose to adapt SBFL techniques to measure the suspiciousness score of each feature set in a SPL. Fig. 2 depicts the overview of the approach from a black-box perspective. Our approach receives a feature model and a product suite as inputs, and it returns a ranking of all the feature sets in the SPL, ordered by their suspiciousness value in descendent order, according to a given suspiciousness technique, e.g., Tarantula. The process to calculate the suspiciousness scores and to break ties in the final ranking is detailed next.

3.1.1. Constructing the coverage matrix and error vector

Based on the SBFL theory (explained in Section 2.2), we consider the SPL products under test as the test cases, and the feature sets as the components where faults must be located. As an example, consider the feature model in Fig. 1 and the product suite in Table 1. Table 3 depicts the coverage matrix, where the products under test are placed in columns, and the feature sets are listed in rows (note that a bug is simulated in the feature `MP3`). For the sake of simplicity, only feature sets composed of one or two features are considered, although the approach could be generalized to feature sets of any size. In the example, only some feature pairs are shown to keep this paper at a reasonable size. For each product under test (i.e., P_1, P_2,..., P_8), a cell is marked with "•" if it contains the feature set of the row. Additionally, the final row depicts the error vector, that is, the test outcome of each product, either successful ("S"), if all the test cases associated to the product passed, or failed ("F"), if at least one of the test failed. We may recall that test cases related to each product can be executed using any state-of-the-art testing technique, e.g., ASTERYSCO for CPS product lines [21]. Also, we reiterate that an underlying assumption in SBFL is that multiple failed and multiple successful test cases are available [16].

Based on the information collected in the coverage matrix and the error vector, the suspiciousness score of each feature set can be calculated using any of the state-of-the-art suspiciousness techniques proposed in the literature [15]. To this purpose, we propose a slight modification of the meaning of the classical notation used in SBFL formulas, where test cases are replaced by *products* and components are replaced by *feature sets*, namely:

N_{CF}	number of failed products that cover a feature set.
N_{UF}	number of failed products that do not cover a feature set.
N_{CS}	number of successful products that cover a feature set.
N_{US}	number of successful products that do not cover a feature set.
N_C	total number of products that cover a feature set.
N_U	total number of products that do not cover a feature set.
N_S	total number of successful products.
N_F	total number of failed products.

Table 3 shows the values of N_{CF}, N_{CS}, N_F and N_S for each feature set. Based on this information, the suspiciousness of each feature set using *Tarantula* is depicted in the column "Suspiciousness", followed by the position of each feature set in the ranking. As illustrated, the feature sets `MP3` (where a fault was seeded) and `MP3-Colour` are placed at the top of the ranking, followed by `Media` and `Camera`, with a suspiciousness score of 0.8 and 0.75 respectively. The rest of single features have a suspiciousness score of 0.5 according to Tarantula. Finally, the feature set `GPS-Colour` has a suspiciousness score of 0.

3.1.2. Breaking ties

The last column in Table 3 indicates the suspiciousness ranking of each feature set. As illustrated, the suspiciousness score of some feature sets are identical. We have taken three different strategies to break ties:

- *Core features:* If a core feature is faulty, all products will fail, and thus, for some techniques (e.g., Tarantula) all the feature sets will have the same suspiciousness score. If this occurs, our SBFL approach places core features at the top of the suspiciousness ranking. Notice that this does not happen with all techniques (e.g., Wong).

- *Feature interactions:* Faults in isolated features may distort feature groups suspiciousness scores. Take as an example the simulated fault in `MP3`. All feature sets including the feature `MP3` will fail, which will result, for some techniques (e.g., Tarantula), in all feature sets including `MP3` having the same suspiciousness score, e.g., the Tarantula scores of `MP3` and `MP3-Colour` in Table 3 are equal. Under this scenario, when a feature set S has the same suspiciousness that any of its feature subsets $S' \subset S$, then S' is ranked over S.

- *Parental relations:* If a parent feature is faulty, all the products containing one or more of its subfeatures will also be faulty, since parent and child features must appear together in products. Hence, for instance, a bug in the feature `Media` would make all the products including any of its child features to fail, that is, those including `Camera`, `MP3`, or both. To address this issue, when a parent feature has the same suspiciousness score as its child features, the parent feature is ranked first.

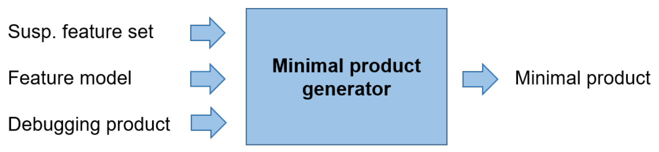

Figure 3: Overview of our approach for fault isolation in SPL

All ties obtained after applying the previous strategies are broken randomly. We remark, however, that other strategies would also be feasible and studying their effectiveness remains for future work.

3.2. Fault isolation

Even if we have a list of the most suspicious feature sets, it could still be challenging to find a product, hopefully as small as possible, where the fault can be easily located. This is the goal of techniques like delta-debugging [28], which aims to generate minimal inputs inducing the failure in the program under test (see related work section). Based on this idea, in this section, we present a debugging approach for the isolation of bugs in SPLs. The goal is to generate a minimal product, in terms of number of features, where the fault(s) can be easily located. For the generation of the product, we leveraged advanced tools for the automated analysis of feature models. More specifically, we used the analysis operations on feature models integrated into the tool SPLAR [24].

The overall overview of the approach for generating the minimal product is depicted in Fig. 3. The debugging approach receives a suspicious feature set (FS), a feature model (with a set of features F), and the failing product being debugged (P) as inputs. Then, a partial configuration is created in three steps, namely: (1) unselect the features that are not part of the product being debugged, (2) select the core features (C), and (3) select the features in the suspicious feature set (out of the remaining features). Formally, let S and R be the sets of selected and removed features in the partial configuration respectively. The partial configuration is defined as follows.

$$\begin{aligned} \forall f \in F \bullet \; f \notin P \;&\Rightarrow\; f \in R \;\wedge \\ f \in C \;&\Rightarrow\; f \in S \;\wedge \\ f \in FS \;&\Rightarrow\; f \in S \end{aligned} \tag{2}$$

The partial configuration is then provided as an input to the propagate operation, which generates a minimal valid product including the suspicious feature set. It is noteworthy that the minimal product generated is composed of a subset of the features in the product being debugged, and thus no new features are considered, which could result in unexpected results, e.g., new faults being introduced.

Continuing with the previous example, let us assume that the feature `MP3` has the highest suspiciousness score, and P5 = {`MobilePhone`, `Screen`, `Calls`, `High resolution`, `Media`, `Camera`, `MP3`, `GPS`} is the faulty product being debugged. A partial configuration would be created by unselecting the features not contained in the product (`Colour`, `Basic`), selecting

Table 3: An example showing the suspiciousness value computed using the Tarantula technique in the Mobile Phone SPL

ID	Feature Set	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	N_{CF}	N_{CS}	N_F	N_S	Suspiciousness	Ranking
F_1	MobilePhone	•	•	•	•	•	•	•	•	4	4	4	4	0.5	5
F_2	Screen	•	•	•	•	•	•	•	•	4	4	4	4	0.5	5
F_3	Calls	•	•	•	•	•	•	•	•	4	4	4	4	0.5	5
F_4	High resolution	•			•	•			•	2	2	4	4	0.5	6
F_5	Basic						•	•		1	1	4	4	0.5	6
F_6	Colour		•	•						1	1	4	4	0.5	6
F_7	GPS			•		•				1	1	4	4	0.5	6
F_8	Media		•		•	•	•		•	4	1	4	4	0.8	3
F_9	Camera				•	•			•	2	1	4	4	0.75	4
F_{10}	MP3 (**BUG**)		•		•	•	•			4	0	4	4	1	1
F_{11}	GPS-Colour			•						0	1	4	4	0	7
F_{12}	MP3-Colour		•							1	0	4	4	1	2
Execution results		S	F	S	F	F	F	S	S						

the core features (`Mobile phone`, `Calls`, `Screen`), and selecting the suspicious feature set (`MP3`). This configuration would be then provided as input to the propagate operation, together with the feature model and the product under debug (i.e., P5). The propagation function would return the following product {`Mobile phone`, `Calls`, `Screen`, `High resolution`, `Media`, `MP3`}. Note that the features `Media` and `High resolution` are automatically selected, whereas the feature `GPS` is not selected. On the one hand, the feature `Media` would be included because it is the parent feature of the `MP3` feature. On the other hand, the feature `High resolution` is selected because the product under debug employs this feature as an alternative child of the `Screen` feature, which is one of the core features of the product line. We may remark that the products generated by the propagate operation are always minimal, i.e., only those features strictly necessary to make a valid product are selected. Therefore, the debugger is provided with the smallest product including the suspicious feature set, contributing to reduce the effort required to locate the bug.

The reason for proposing an incremental approach instead of a decremental approach is complexity. Minimizing a product is an exponential problem: given a product P with t features, the potential number of sub-products of P (products composed of a subset of the features of P) is $2^t - 1$. Of course, not all feature combinations are valid and, thus, the feature model must be taken into account, which includes further constraints. In addition to this, a decremental approach may require re-testing many large products until finding the faulty feature(s). Instead, the incremental approach proposed would be faster since it requires a single SAT propagation and, more importantly, it guarantees that a minimal product is generated. As a further benefit, the generated products are as small as possible (since they only include the core features and the first suspicious feature set in the ranking), which is good in the case that several iterations are needed before locating the faulty feature sets (i.e., a product with less features requires less test effort [29, 30]).

3.3. Methodology

Figure 4 depicts the overall methodology to apply our SPL debugging approach. First, the suspiciousness scores of each feature sets are calculated based on the coverage information and test results, as explained in Section 3.1. Then, for each faulty product, the most suspicious feature set is selected and a minimal product is generated and tested. We reiterate that the tests can be performed using any state-of-the-art testing technique and it is out of the scope of this paper. If the test outcome is successful, the next most suspicious feature set is selected and another minimal product is generated. Conversely, if the product fails, the suspicious feature set is reported to the engineer to fix it. This process is repeated until all faults have been fixed. Notice that every time a faulty product is selected, the tests must be executed again to confirm that the product is still buggy, since the faults could have been fixed while debugging previous products. Finally, it is noteworthy that the calculation of the suspiciousness scores is only performed once, unlike related approach where it is calculated every time a bug is fixed [31]. Although this may affect the accuracy of our approach, we believe that this is a sensible strategy for SPLs where re-executing all the tests is usually very costly [29, 12, 32].

4. Evaluation

4.1. Research questions

In order to evaluate the effectiveness of feature-based SBFL in SPLs we aim to answer the following Research Questions (RQs):

RQ1: *What is the effectiveness of different state-of-the-art suspiciousness techniques at isolating the causes of failures in SPLs?*

RQ2: *How the size of the product suite affects the performance of the techniques under study?*

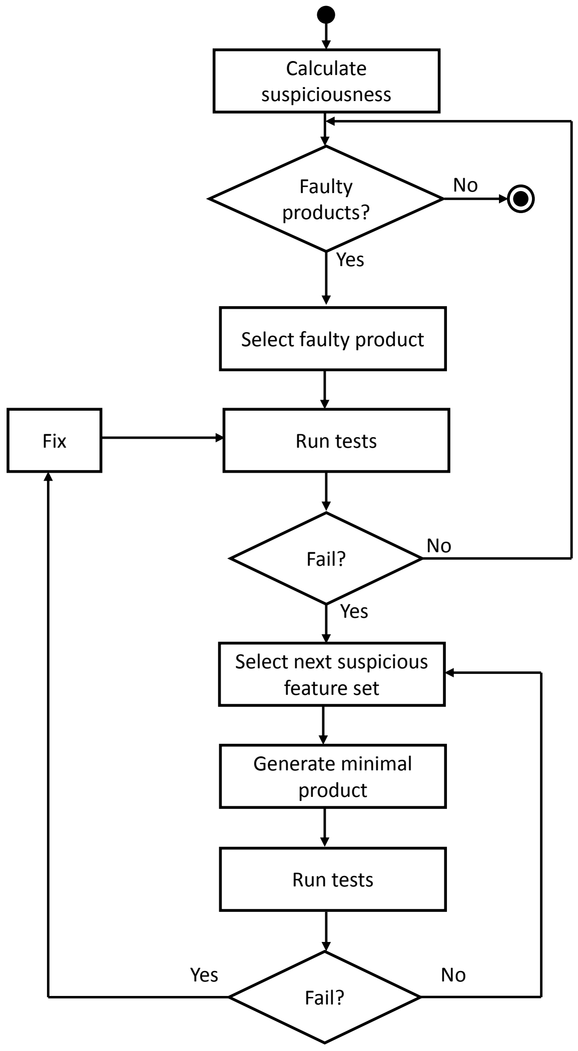

Figure 4: Overview of the methodology for SPL fault isolation

RQ3: *How the number and type of faults (single or interaction) affect the performance of the techniques under study?*

4.2. Experimental design

4.2.1. Subject models and product suites

We selected nine feature models representing SPLs of different sizes for the evaluation. Seven of the models were taken from the SPLOT repository [24]. Furthermore, we used the feature model of the Drupal framework, a realistic case study to evaluate variability testing techniques proposed by Sanchez et al. [33]. Additionally, we included a case study of an Unmanned Aerial Vehicle (UAV) that we previously used in other evaluations (e.g., [21, 30, 34]). For each subject model, the SPLCAT tool [35] was used to generate two product suites using 2-wise and 3-wise coverage criteria [36]. Table 4 depicts the characteristics of the selected models including number of features, number of cross-tree constraints (CTCs), total number of products, and number of products in the 2-wise and 3-wise product suites respectively.

Case Study	Features	CTC	Products	2-wise	3-wise
Drupal V3	21	9	96,768	11	37
Weather station	23	2	1,056	14	40
Eclipse	29	3	983,150	17	54
Android	45	5	36,240	18	67
UAV	46	4	2.3E6	22	74
Dell Laptop	47	109	2,319	47	142
Arcade	62	35	3.3E9	18	65
HIS	68	4	6,400	12	41
Model transformation	88	0	1.6E13	28	133

Table 4: Subject feature models

4.2.2. Fault seeding and test execution

We faced two obstacles in the selection of case studies for the evaluation of our approach. First, we found a lack of case studies with available feature models, source code, and test cases. Second, based on our experience with industrial partners [37], the execution of test cases in real setting is usually a time-consuming process, which hinders the use of real test cases in a large-scale evaluation as the one required in our paper. To address both obstacles, we resorted to a fault simulator in eight of the subject case studies (where no code nor test cases were available), as previously done in related papers [11, 38, 39, 9, 40]. Additionally, we used a real-world case study with available feature model, source code and test cases (Unmanned Aerial Vehicle), in order to evaluate the approach in realistic settings.

For the simulation of faults (in all case studies except UAV), we developed a fault generator to simulate different number and types of faults in the SPLs under test. The fault generator is based on the one proposed by Ensan et al. [38] and it has been used in several works to evaluate the fault detection rate of SPL test suites (e.g., [38][11][41]). The fault generator simulates faults in single features as well as faults caused by the interaction of two features. More specifically, our generator receives a feature model as an input and returns a random list of faulty feature sets as an output. For instance, the following list simulates two faults in the SPL in Fig. 1: {{`Colour`}, {`GPS`, `MP3`}}, a fault in the feature `Colour` and another fault caused by the interaction of the features `GPS` and `MP3`.

In addition to the fault simulator, we developed a test system to simulate the test outcomes of each product using a simple oracle: if a product contains any of the features labelled as faulty, the execution of the product is classified as failed, otherwise it is classified as successful. This is an intuitive approach that assumes that the test cases of each product are good enough to reveal failures in the products under test. Note that this is a key requirement for the application of SBFL: if test cases are not able to identify failures, they will certainly not be helpful in identifying faults. Both, the fault simulator and the test system, have been previously used in the literature [11].

As for the UAV case study, the experiments were performed employing a Simulink model in charge of simulating the UAV. We employed a test suite composed of 120 test cases. A test case in our case was a set of signals stimulating the inputs of the SUT over a specific amount of time. The test execution time for each test case lasts from 30 seconds to 3000 seconds.[1] Furthermore, we employed mutation testing to simulate faults. Mutation testing was employed since it has been demonstrated to be a good substitute of real faults [42]. For each fault in a specific feature set, a mutant was created, performing the mutation in one of the assets of that feature sets. This mutant was later selected when a product included the faulty feature set. We employed the mutation operators proposed by Hanh et al. for Simulink models [43]. To speed up the evaluation process, we prioritized the test cases with an additional greedy algorithm that used historical data of the test cases. This algorithm demonstrated to be effective in a previous work at detecting faults as fast as possible [32]. Since SBFL only uses information whether the test execution passed or failed, once the test cases detected a fault, the test execution was stopped with the aim of speeding up the evaluation process.

4.2.3. Suspiciousness techniques

We assessed the effectiveness of ten state-of-the-art suspiciousness techniques for the isolation of faults in SPLs. The chosen techniques were Tarantula, Ochiai, Dstar, Naish2, Wong and Russel-Rao, as proposed in [26]. We also included Kulcynski2, Arithmetic mean, Ample2 and M2, as they showed promising results in preliminary experiments [15]. The algebraic form of the chosen techniques are shown in Table 5 using the notation presented in Section 3.1. In the Dstar technique's formula, the * is an exponent of N_{CF}. We set * equal to 2 based on the original paper [20] and other relevant ones (e.g., [19]).

4.2.4. Evaluation metrics

The following metrics were used to measure the effectiveness of the approach.

Percentage of examined features (EXAMF). The EXAM score is one of the most common metrics to evaluate the effectiveness of fault localization techniques [16, 19, 44, 45]. It is calculated as the number of statements examined with respect to the total number of statements in the program. In our approach, the number of statements examined could be intuitively substituted by the number of feature sets examined, and the total number of statements by the total number of features sets. Given a product p being debugged and a faulty feature set f, we propose a variant of the EXAM score, called *EXAMF*, calculated as follows:

$$EXAMF(p, f) = \frac{NF_f}{NF_p} \times 100\% \tag{3}$$

Where NF_f is the number of feature sets examined to isolate the fault in f, and NF_p is the total number of feature sets in p. Since we are aiming at faults caused by a single feature or interaction between two features, NF_p is equal to all the valid possible combinations of one or two of the features of p. This was calculated using the SPLCAT tool. The lower the EXAMF score is, the more effective is the technique.

[1]Notice that this is the simulated test execution time

As an example, consider a fault in the feature `MP3`, and P5 = {`MobilePhone`, `Screen`, `Calls`, `High resolution`, `Media`, `Camera`, `MP3`, `GPS`} the faulty product being debugged. Let us suppose that `GPS` is the most suspicious feature and `MP3` the second most suspicious feature, according to a certain technique. Accordingly, the debugger would examine first the `GPS` feature, proceeding later to examine the `MP3` feature. Considering that the total number of valid feature sets (i.e., single features and pairs of features) in P5 is 28, this metric is calculated as $EXAMF(P5, \{MP3\}) = (2/28) \times 100 = 7.14$. This means that 7.14% of the feature sets in P5 had to be examined in order to locate the fault in `MP3`.

The EXAMF metric measures the effectiveness of a fault localization technique at detecting a single fault. In the cases where several faults are present, the effectiveness of each fault localization technique was evaluated as the average EXAMF score. Thus, the *average EXAMF* score for multiple faulty feature sets F in a product p that is being debugged is calculated as follows:

$$\frac{\sum_{i=1}^{|F|} EXAMF(p, F_i)}{|F|} \tag{4}$$

As an example, let us suppose two faults in the `MP3` and `GPS` features, and P5 the faulty product being debugged. Let us suppose that 4 feature sets were examined before isolating the fault in `MP3`, and 5 feature sets were checked before isolating the bug in `GPS`, i.e., $EXAMF(P5, \{MP3\}) = (4/28) \times 100 = 14.2$ and $EXAMF(P5, \{GPS\}) = (5/28) \times 100 = 17.8$. The average EXAMF is calculated as $(14.2 + 17.8)/2 = 16$. That is, 16% of the feature sets need to be examined on average to locate each faulty feature set in P5.

4.2.5. Experiments

In order to answer our research questions, we performed five independent experiments with different number and types of simulated faults. Each experiment was conducted on the subject models depicted in Table 4 assessing the effectiveness of the ten suspiciousness techniques depicted in Table 5. Table 6 shows the number of simulated faults in single and pairs of features in each experiment. As proposed by Sanchez et al. [11], the maximum number of faults in each model was set to n/10, being n the number of features in the SPL. For the fifth experiment, where faults due to single features and interaction of two features are combined, the distribution of the simulated faults was the same for both type of faults, as proposed in [11]. For each experiment and case study, five different distributions of faults were randomly generated, so-called *test scenarios*, in order to calculate averages. In total, 40 different test scenarios were run on each experiment and product suite: 8 case studies × 5 test scenarios.

4.3. Experimental results

4.3.1. Experiment 1: a fault in a single feature

This experiment aims at evaluating the approach when the SPL has one fault in a single feature. Tables 7 and 8 report the

Table 5: Algebraic form of the suspiciousness techniques under evaluation

Technique	Equation	Technique	Equation
Tarantula	$\frac{N_{CF}/N_F}{N_{CS}/N_S + N_{CF}/N_F}$	Russel-Rao	$\frac{N_{CF}}{N_{CF} + N_F - N_{CF} + N_{CS} + N_S - N_{CS}}$
Ochiai	$\frac{N_{CF}}{\sqrt{N_F(N_{CF} + N_{CS})}}$	Kulcynski2	$\frac{N_{CF}}{N_{UF} + N_{CS}}$
Dstar	$\frac{(N_{CF})^*}{(N_F - N_{CF}) + N_{CS}}$	Arithmetic mean	$\frac{2(N_{CF} \times N_{US} - N_{UF} \times N_{CS})}{(N_{CF} + N_{CS}) \times (N_{US} + N_{UF}) + (N_{CF} + N_{UF}) \times (N_{CS} + N_{US})}$
Naish2	$N_{CF} - \frac{N_{CS}}{N_{CS} + (N_S - N_{CS}) + 1}$	Ample2	$\lvert\frac{N_{CF}}{N_{CF} + N_F} - \frac{N_{CS}}{N_{CS} + N_{US}}\rvert$
Wong	N_{CF}	M2	$\frac{N_{CF}}{N_{CF} + N_S - N_{CS} + 2(N_F - N_{CF} + N_{CS})}$

Experiment	Single faults	Interaction faults
1	1	0
2	[2, n/10]	0
3	0	1
4	0	[2, n/10]
5	[1, n/5]	[1, n/5]

Table 6: Types of faults simulated in each experiment (n = number of features in the SPL)

EXAMF values of each suspiciousness technique under evaluation on the data collected from the 2-wise and 3-wise product suites respectively. The best value on each column is highlighted in boldface. We reiterate that the shown values are the average of five different scenarios with a randomly simulated fault on each of them. The EXAMF score ranged between 0.07% and 7.6% for the 2-wise product suite, and between 0.07% and 8.43% for the 3-wise suite. That is, both product suites yielded similar results, with only slight differences in favor of the 2-wise suite. This means that having more test data information was not necessarily helpful in this experiment.

The performance of all the techniques was consistent in all the case studies, and in both product suites. The faulty feature was successfully ranked as the most suspicious feature in 100% of the test scenarios for Ample2, Dstar, Kulcynski2, M2, and Ochiai, i.e., these were the techniques showing the best performance. In the case of Tarantula, the most suspicious feature was ranked first in 98% (88 out of 90) of the test scenarios. The technique performing worst was Arithmetic mean, followed by Naish2, Russel-Rao, and Wong.

4.3.2. Experiment 2: multiple faults in single features

This experiment evaluates the approach when the SPL contains multiple faults in two or more single features. Tables 9 and 10 show the average EXAMF value of each technique under evaluation on the data collected from the 2-wise and 3-wise product suites respectively. As illustrated, the results obtained with the 3-wise suite (between 0.10% and 8.89%) were slightly better than those obtained with the 2-wise suite (between 0.22% and 12.38%). More specifically, the EXAMF values of the 3-wise suite outperformed those of the 2-wise suite in 69 out of the 90 measures (10 techniques x 9 case studies). This means that the use of more test data improved the performance of the fault isolation techniques in this particular experiment.

Overall, the technique performing best with both product suites was Tarantula, followed by Kulcynski2, and Ample2. Conversely, Russel-Rao, Wong and Naish2, which showed exactly the same results in all case studies, resulted in the techniques with worst performance in this experiment.

4.3.3. Experiment 3: fault in a feature interaction

This experiment evaluates the approach under the presence of one fault due to the interaction of two features. Tables 11 and 12 show the mean EXAMF value of each technique over the five test scenarios. As in the previous experiment, the results obtained with the 3-wise suite were significantly better than those obtained with the 2-wise suite. More specifically, the EXAMF values of the 3-wise suite outperformed those of the 2-wise suite in 82 out of the 90 measures. Interestingly, the mean EXAMF values were significantly higher (up to 47.57%) than those observed in the previous experiments, which suggests that, as expected, locating bugs caused by the interaction of features is harder than isolating bugs in single features. Also, analogously to Experiment 1, where a single fault was also simulated, the performance of the techniques was consistent across all the case studies showing identical conclusions for both product suites. More specifically, the techniques performing best were Ample2, Dstar, Kulcynski2, M2, and Ochiai, all of them with the same average score. Conversely, the technique Arithmetic mean performed significantly bad in comparison with the rest of techniques, with a mean score over 21% with both product suites.

4.3.4. Experiment 4: multiple faults in feature interactions

This experiment aims to evaluate our approach when the SPL has multiple faults caused by feature interactions. Tables 13 and 14 show the average EXAMF values obtained in each of the case studies for five different test scenarios. As in the previous two experiments, the techniques showed significantly better performance with the 3-wise suite compared to the 2-wise suite. This improvement was significant in the case of Tarantula where the overall average EXAMF value decreased from 5.13% with the 2-wise suite to 0.93% with the 3-wise suite. Overall, the EXAMF values of the 3-wise suite outperformed those of the 2-wise suite in 68 out of the 90 measures. It is also noteworthy that the average EXAMF scores in this experiment are noticeably higher than in the previous ones. This suggests

Table 7: EXAMF scores obtained using the 2-wise product suite in Experiment 1. Best values on each column are highlighted in boldface

Technique	Drupal V3	Weather St.	Eclipse	Android	UAV	Dell L.	Arcade	HIS	Model T.	Mean
Ample2	**1.24**	**1.20**	**1.04**	**0.30**	**0.31**	**0.77**	**0.14**	**0.07**	**0.12**	**0.58**
Arithmetic M.	7.64	2.14	4.2	1.27	2.62	3.86	0.97	0.64	2.06	2.82
Dstar	**1.24**	**1.20**	**1.04**	**0.30**	**0.31**	**0.77**	**0.14**	**0.07**	**0.12**	**0.58**
Kulcynski2	**1.24**	**1.20**	**1.04**	**0.30**	**0.31**	**0.77**	**0.14**	**0.07**	**0.12**	**0.58**
M2	**1.24**	**1.20**	**1.04**	**0.30**	**0.31**	**0.77**	**0.14**	**0.07**	**0.12**	**0.58**
Naish2	1.73	**1.20**	1.94	**0.30**	0.36	1.71	0.24	0.16	0.34	0.88
Ochiai	**1.24**	**1.20**	**1.04**	**0.30**	**0.31**	**0.77**	**0.14**	**0.07**	**0.12**	**0.58**
Russel-Rao	1.73	**1.20**	1.94	**0.30**	0.36	1.71	0.24	0.16	0.34	0.88
Tarantula	**1.24**	**1.20**	1.17	**0.30**	**0.31**	**0.77**	**0.14**	**0.07**	**0.12**	0.59
Wong	1.73	**1.20**	1.94	**0.30**	0.36	1.71	0.24	0.16	0.34	0.88
Mean	2.03	1.29	1.64	0.34	0.56	1.36	0.25	0.15	0.38	0.90

Table 8: EXAMF scores obtained using the 3-wise product suite in Experiment 1. Best values on each column are highlighted in boldface

Technique	Drupal V3	Weather St.	Eclipse	Android	UAV	Dell L.	Arcade	HIS	Model T.	Mean
Ample2	**1.92**	**1.11**	**0.91**	**0.29**	**0.31**	**0.76**	**0.13**	**0.07**	**0.10**	**0.62**
Arithmetic M.	8.43	1.83	3.83	1.31	2.83	3.82	1.00	0.61	1.98	2.85
Dstar	**1.92**	**1.11**	**0.91**	**0.29**	**0.31**	**0.76**	**0.13**	**0.07**	**0.10**	**0.62**
Kulcynski2	**1.92**	**1.11**	**0.91**	**0.29**	**0.31**	**0.76**	**0.13**	**0.07**	**0.10**	**0.62**
M2	**1.92**	**1.11**	**0.91**	**0.29**	**0.31**	**0.76**	**0.13**	**0.07**	**0.10**	**0.62**
Naish2	2.48	**1.11**	1.73	**0.29**	0.36	1.71	0.24	0.16	0.28	0.93
Ochiai	**1.92**	**1.11**	**0.91**	**0.29**	**0.31**	**0.76**	**0.13**	**0.07**	**0.10**	**0.62**
Russel-Rao	2.48	**1.11**	1.73	**0.29**	0.36	1.71	0.24	0.16	0.28	0.93
Tarantula	**1.92**	**1.11**	0.99	**0.29**	**0.31**	**0.76**	**0.13**	**0.07**	**0.10**	0.63
Wong	2.48	**1.11**	1.73	**0.29**	0.36	1.71	0.24	0.16	0.28	0.93
Mean	2.74	1.18	1.46	0.39	0.58	1.35	0.25	0.15	0.34	0.94

Table 9: EXAMF scores obtained using the 2-wise product suite in Experiment 2. Best values on each column are highlighted in boldface

Technique	Drupal V3	Weather St.	Eclipse	Android	UAV	Dell L.	Arcade	HIS	Model T.	Mean
Ample2	3.27	1.37	6.26	0.90	0.77	1.11	1.27	0.60	1.42	1.88
Arithmetic M.	7.32	2.95	6.26	1.10	2.36	2.79	0.98	0.35	1.32	2.82
Dstar	3.74	1.41	11.50	3.00	3.42	2.04	2.98	0.89	4.99	3.78
Kulcynski2	**3.10**	**1.21**	3.74	0.40	**0.46**	**0.98**	**0.49**	**0.22**	0.39	1.22
M2	3.74	1.71	12.06	3.16	3.97	2.28	3.04	0.89	5.60	4.05
Naish2	4.59	2.01	12.38	3.33	4.57	2.54	3.10	0.94	5.79	4.36
Ochiai	3.58	1.41	10.13	2.48	2.69	1.40	2.66	0.89	3.89	3.23
Russel-Rao	4.59	2.01	12.38	3.33	4.57	2.54	3.10	0.94	5.79	4.36
Tarantula	**3.10**	**1.21**	**3.59**	**0.37**	**0.46**	**0.98**	**0.49**	**0.22**	**0.36**	**1.20**
Wong	4.59	2.01	12.38	3.33	4.57	2.54	3.10	0.94	5.79	4.36
Mean	4.16	2.73	9.07	2.14	2.78	1.92	2.12	0.69	3.53	3.12

Table 10: EXAMF scores obtained using the 3-wise product suite in Experiment 2. Best values on each column are highlighted in boldface

Technique	Drupal V3	Weather St.	Eclipse	Android	UAV	Dell L.	Arcade	HIS	Model T.	Mean
Ample2	**2.1**	**1.10**	1.67	0.72	0.62	1.12	1.14	0.34	0.80	1.06
Arithmetic M.	8.55	2.80	6.00	1.23	2.21	2.85	0.91	0.38	1.08	2.89
Dstar	3.66	1.20	7.14	2.65	2.66	2.23	2.91	1.22	4.41	3.12
Kulcynski2	2.50	**1.10**	1.33	0.34	**0.29**	0.95	0.38	**0.10**	0.22	0.80
M2	3.93	1.26	7.72	2.88	3.00	2.40	3.03	1.25	4.72	3.36
Naish2	4.13	1.66	8.89	3.09	3.36	2.60	3.10	1.27	5.24	3.71
Ochiai	2.61	**1.10**	5.37	2.20	1.91	1.47	2.62	1.11	3.55	2.43
Russel-Rao	4.13	1.66	8.89	3.09	3.36	2.60	3.10	1.27	5.24	3.71
Tarantula	2.50	**1.10**	**1.21**	**0.31**	**0.29**	**0.95**	**0.38**	**0.10**	**0.20**	**0.78**
Wong	4.13	1.66	8.89	3.09	3.36	2.60	3.10	1.27	5.24	3.71
Mean	3.82	1.46	5.71	1.96	2.11	1.98	2.07	0.83	3.07	2.56

Table 11: EXAMF scores obtained using the 2-wise product suite in Experiment 3. Best values on each column are highlighted in boldface

Technique	Drupal V3	Weather St.	Eclipse	Android	UAV	Dell L.	Arcade	HIS	Model T.	Mean
Ample2	**2.2**	**1.24**	**1.22**	**0.27**	**0.3**	**0.97**	**0.32**	**0.16**	**0.31**	**0.77**
Arithmetic M.	36.41	17.05	47.35	13.59	13.57	15.7	13.34	5.34	22.37	20.52
Dstar	**2.2**	**1.24**	**1.22**	**0.27**	**0.3**	**0.97**	**0.32**	**0.16**	**0.31**	**0.77**
Kulcynski2	**2.2**	**1.24**	**1.22**	**0.27**	**0.3**	**0.97**	**0.32**	**0.16**	**0.31**	**0.77**
M2	**2.2**	**1.24**	**1.22**	**0.27**	**0.3**	**0.97**	**0.32**	**0.16**	**0.31**	**0.77**
Naish2	13.89	5.28	23.2	3.16	4.98	11.59	2.44	2.19	6.51	8.14
Ochiai	**2.2**	**1.24**	**1.22**	**0.27**	**0.3**	**0.97**	**0.32**	**0.16**	**0.31**	**0.77**
Russel-Rao	13.89	5.28	23.2	3.16	4.98	11.59	2.44	2.19	6.51	8.14
Tarantula	4.93	4.2	3.27	1.29	0.65	1.66	2.56	0.34	1.06	2.21
Wong	13.89	5.28	23.2	3.16	4.98	11.59	2.44	2.19	6.51	8.14
Mean	9.40	4.32	12.63	2.57	3.07	5.70	2.48	1.30	4.45	5.10

Table 12: EXAMF scores obtained using the 3-wise product suite in Experiment 3. Best values on each column are highlighted in boldface

Technique	Drupal V3	Weather St.	Eclipse	Android	UAV	Dell L.	Arcade	HIS	Model T.	Mean
Ample2	**1.42**	**1.05**	**0.73**	**0.21**	**0.26**	**0.71**	**0.20**	**0.07**	**0.10**	**0.53**
Arithmetic M.	38.86	17.73	47.57	12.43	13.89	15.80	14.58	5.47	22.9	21.03
Dstar	**1.42**	**1.05**	**0.73**	**0.21**	**0.26**	**0.71**	**0.20**	**0.07**	**0.10**	**0.53**
Kulcynski2	**1.42**	1.05	**0.73**	**0.21**	**0.26**	**0.71**	**0.20**	**0.07**	**0.10**	**0.53**
M2	**1.42**	**1.05**	**0.73**	**0.21**	**0.26**	**0.71**	**0.20**	**0.07**	**0.10**	**0.53**
Naish2	5.94	3.14	5.30	2.01	1.68	6.35	1.49	1.16	1.19	3.14
Ochiai	**1.42**	**1.05**	0.73	**0.21**	**0.26**	**0.71**	**0.20**	**0.07**	**0.10**	**0.53**
Russel-Rao	5.94	3.14	5.30	2.01	1.68	6.35	1.49	1.16	1.19	3.14
Tarantula	**1.42**	1.86	**0.73**	**0.21**	**0.26**	**0.71**	0.47	**0.07**	**0.10**	0.65
Wong	5.94	3.14	5.30	2.01	1.68	6.35	1.49	1.16	1.19	3.14
Mean	6.52	3.43	6.78	1.97	2.05	3.91	2.05	0.94	2.71	3.37

that isolating multiple interaction faults imposes a significantly hard problem for the techniques under evaluation.

From the results, it is observed that Tarantula is the most effective technique to isolate multiple interaction faults, achieving the lowest average EXAMF value in 8 out of the 9 case studies with both test suites. Conversely, Arithmetic mean was the technique that showed the worst performance, followed by Russel-Rao, Naish2 and Wong.

4.3.5. *Experiment 5: faults in single features and feature interactions*

This experiment assessed the proposed approach in SPLs containing faults in single features as well as faults due to the interaction of two features. Tables 15 and 16 show the average EXAMF values obtained in this experiment for the eight case studies. As in the previous experiments, the overall performance of most techniques was better when using the 3-wise suite than when using the 2-wise suite. More specifically, the EXAMF values of the 3-wise suite outperformed those of the 2-wise suite in 62 out of the 90 measures. In contrast to the previous experiments, the results with each suite revealed slight differences, although they overall agree that the techniques performing best were Tarantula, Kulcynski2 and Ample2. Conversely, and in line with the previous experiments, the technique showing the worst performance is Arithmetic mean, followed by Russel-Rao, Naish2 and Wong.

4.3.6. *Statistical Analysis*

Results of the performed experiments were analyzed by means of statistical analysis. Specifically, for each experiment of each case study, each pair of the metrics were analyzed with a post-hoc analysis employing the Kruskal-Wallis test [46], which is a non-parametric method. This returned a p-value for each pair of metrics. The p-value indicates whether there is a statistically significant difference between two different SBFL techniques or not. As the statistical significance level was set to 95%, we considered that there was statistical significance between two different techniques when the $p\text{-value} < 0.05$. When the p-value of the Kruskal-Wallis test returned a value below 0.05, the Vargha and Delaney test was employed to obtain the $\hat{A}_{12}$ value [47][48]. The $\hat{A}_{12}$ value determines the difference between two techniques and see which of the two techniques is better.

Tables 17 and 18 summarize the results for the statistical analysis related to the performed experiments for the 2-wise and 3-wise suites. These tables indicate the number of times, out of 45 (5 experiments × 9 case studies), in which the technique in the row outperformed the technique in the column with statistical significance (i.e., $p\text{-value} < 0.05$ and the $\hat{A}_{12}$ in favor of the technique in the row). After the statistical analysis, it can be appreciated that the best metric was Kulcynski2. In fact, this metric was not statistically outperformed by any of the other metrics. However, the rest of metrics were outperformed by

Table 13: EXAMF scores obtained using the 2-wise product suite in Experiment 4. Best values on each column are highlighted in boldface

Technique	Drupal V3	Weather St.	Eclipse	Android	UAV	Dell L.	Arcade	HIS	Model T.	Mean
Ample2	**8.93**	3.93	14.47	2.35	3.69	2.38	4.93	1.49	5.15	5.25
Arithmetic M.	44.69	19.86	52.54	11.52	17.44	14.96	10.90	5.39	23.23	22.28
Dstar	13.06	7.39	27.70	4.97	6.17	8.85	7.14	3.40	15.06	10.42
Kulcynski2	14.92	5.19	14.49	3.23	2.27	1.79	3.69	1.48	5.38	5.83
M2	13.19	6.14	27.57	5.89	6.9	5.23	7.91	3.42	15.29	10.17
Naish2	23.13	13.01	37.55	8.72	12.45	9.53	9.86	4.20	18.56	15.22
Ochiai	11.91	4.51	23.26	3.52	3.78	1.96	5.99	2.97	11.16	7.67
Russel-Rao	23.13	13.01	37.55	8.72	12.45	9.53	9.86	4.20	18.56	15.22
Tarantula	12.96	**3.73**	**13.78**	**2.10**	**1.75**	**1.66**	**3.66**	**1.45**	**5.09**	**5.13**
Wong	23.13	13.01	37.55	8.72	12.45	9.53	9.86	4.20	18.56	15.22
Mean	18.90	8.98	28.65	5.97	7.93	6.54	7.38	3.22	13.60	11.34

Table 14: EXAMF scores obtained using the 3-wise product suite in Experiment 4. Best values on each column are highlighted in boldface

Technique	Drupal V3	Weather St.	Eclipse	Android	UAV	Dell L.	Arcade	HIS	Model T.	Mean
Ample2	**2.16**	1.28	2.03	0.96	0.69	1.43	2.36	0.47	1.56	1.44
Arithmetic M.	45.80	20.39	51.16	12.26	11.61	16.30	11.02	5.46	22.98	21.89
Dstar	6.14	2.50	18.51	5.44	4.27	3.37	10.95	3.73	14.09	7.67
Kulcynski2	3.06	1.29	2.11	0.83	0.55	0.87	0.98	0.25	0.39	1.15
M2	11.6	4.63	23.00	7.08	5.41	4.22	11.66	4.19	15.14	9.66
Naish2	18.33	7.63	28.92	8.95	6.85	6.59	12.36	4.92	12.37	13.06
Ochiai	2.18	1.54	7.71	2.78	2.22	1.28	8.03	2.86	4.19	4.44
Russel-Rao	18.33	7.63	28.92	8.95	6.85	6.59	12.36	4.92	12.37	13.06
Tarantula	2.17	**1.10**	**1.68**	**0.71**	**0.43**	**0.87**	**0.85**	**0.22**	**0.29**	**0.93**
Wong	18.33	7.63	28.92	8.95	6.85	6.59	12.36	4.92	16.81	12.37
Mean	12.81	5.56	19.30	5.69	4.57	4.81	8.29	3.19	11.40	8.40

Table 15: EXAMF scores obtained using the 2-wise product suite in Experiment 5. Best values on each column are highlighted in boldface

Technique	Drupal V3	Weather St.	Eclipse	Android	UAV	Dell L.	Arcade	HIS	Model T.	Mean
Ample2	11.29	**1.55**	4.66	**1.55**	0.97	1.79	**2.13**	1.28	2.71	3.10
Arithmetic M.	15.42	8.16	16.47	6.15	4.75	4.92	4.48	1.28	5.42	7.44
Dstar	10.21	3.68	9.94	4.41	3.99	3.15	4.71	1.85	7.47	5.48
Kulcynski2	**9.45**	1.85	4.71	2.01	0.82	1.09	2.26	**0.73**	**1.94**	**2.76**
M2	10.17	3.54	9.73	4.44	3.86	2.75	4.94	1.89	8.00	5.48
Naish2	9.70	4.57	13.31	5.36	5.04	4.05	5.3	1.98	8.73	6.44
Ochiai	11.68	2.95	7.92	3.31	2.37	1.85	4.02	1.83	5.98	4.65
Russel-Rao	9.70	4.57	13.31	5.36	5.04	4.05	5.30	1.98	8.73	6.44
Tarantula	12.06	1.85	**4.39**	2.01	**0.82**	**0.96**	2.26	**0.73**	**1.94**	3.00
Wong	9.70	4.57	13.31	5.36	5.04	4.05	5.30	1.98	8.73	6.44
Mean	10.94	3.73	9.77	4.00	3.27	2.87	4.07	1.55	5.96	5.12

Table 16: EXAMF scores obtained using the 3-wise product suite in Experiment 5. Best values on each column are highlighted in boldface

Technique	Drupal V3	Weather St.	Eclipse	Android	UAV	Dell L.	Arcade	HIS	Model T.	Mean
Ample2	2.46	**1.27**	1.99	**0.89**	0.5	1.22	**1.5**	0.74	1.69	1.36
Arithmetic M.	13.91	7.83	18.29	6.02	5.59	5.43	5.46	1.33	5.26	7.68
Dstar	4.92	3.05	8.13	4.51	3.61	3.52	5.45	2.45	8.24	4.87
Kulcynski2	**2.11**	**1.27**	1.97	1.08	**0.38**	0.91	1.58	**0.33**	**0.75**	1.15
M2	5.93	3.73	9.31	4.61	3.89	3.03	5.66	2.56	8.56	5.25
Naish2	7.58	4.22	12.2	5.14	4.44	3.97	5.85	2.61	9.26	6.14
Ochiai	3.57	1.72	4.08	3.21	2.2	1.49	4.86	2.07	6.40	3.28
Russel-Rao	7.58	4.22	12.2	5.14	4.44	3.97	5.85	2.61	9.26	6.14
Tarantula	2.35	**1.27**	**1.72**	1.08	**0.38**	**0.78**	1.58	**0.33**	**0.75**	**1.14**
Wong	7.58	4.22	12.2	5.14	4.44	3.97	5.85	2.61	9.26	6.14
Mean	5.80	3.28	8.21	3.68	2.99	2.83	4.36	1.76	5.94	4.31

Kulcynski2 at least in one of the experiments for both, the 2-wise and 3-wise suite.

Apart from Kulcynski2, two techniques can be considered as valid ones as compared to the rest for solving the fault localization problem in SPLs: Tarantula and Ample2. Kulcynski2 statistically outperformed Tarantula only in one test scenario for each of the product suites, whereas it statistically outperformed Ample2 in one test scenario for the 2-wise suite and in five test scenarios for the 3-wise suite.

4.4. Discussion

We now summarize the results and what they tell us about the research questions.

RQ1: Effectiveness of different suspiciousness techniques

The results of the experiments and the corresponding statistical analysis reveal that the approach is effective, with some of the techniques allowing to detect the faulty features by examining, on average, 5.13% feature set in the hardest scenarios (i.e., faults caused by multiple feature interaction). The results of the experiments and the corresponding statistical analysis of the data reveal that the techniques Kulcynski2, Ample2 and Tarantula are the most effective suspiciousness techniques for fault isolation in SPLs. It is remarkable that these three techniques showed a very stable performance with different types of faults and suite sizes. In contrast, the results of Ochiai, Dstar and M2 were more sensitive to the type of faults, and diverged significantly among the different experiments. The techniques Arithmetic mean, Russel-Rao, Naish2, and Wong performed badly in all experiments. In the light of these results, RQ1 is answered as follows:

Different suspiciousness techniques may perform very differently in the context of SPLs. Based on the results of our study, the most effective suspiciousness techniques are Kulcynski2, Tarantula and Ample2. Conversely, the techniques Arithmetic mean, Wong, Russel-Rao and Naish2 perform badly and they should be avoided.

RQ2: Size of the suite

The results obtained with the 3-wise suite were consistently better when compared with those obtained with the 2-wise suite. The only exception was Experiment 1 where both suites yielded similar results. We suspect that this was due to the simplicity of the problem, which made both suites to obtain the optimal result easily. Overall, however, the experimental results were expected and in line with the theory behind SBFL, which states that the accuracy of the techniques is better as the size of the test suite increases. Based on our results, RQ2 is answered as follows:

The accuracy of the fault localization techniques gets better as the number of products in the suite increases.

RQ3: Types and Number of Faults

The experimental results show that isolating a single fault (Experiments 1 and 3) is significantly easier than isolating multiple faults (Experiments 2, 4, and 5). This was expected because multiple faults may interfere among them making the results of the suspiciousness metrics less accurate. The results also suggest that detecting multiple interaction faults (Experiment 4) is significantly harder than detecting multiple single and interaction faults, either in isolation (Experiment 2) or combined (Experiment 5). In the view of these results, RQ3 is answered as follows:

The number and type of faults have a strong impact in the effectiveness of the suspiciousness techniques. Isolating single faults is significantly easier than locating multiple bugs. Locating multiple bugs caused by the interaction among different features is the hardest scenario.

5. Threats to validity

The factors that could have influenced our work are summarized in the following internal and external validity threats.

Internal validity: *Are there factors that might affect the results of this evaluation?* The number of simulated faults in each feature model could introduce a bias in our evaluation. To mitigate this threat, we experimented with different numbers of simulated faults, up to a maximum of 10% of the number of features, as proposed in [11]. Similarly, it could be the case that simulated faults affect different types of features differently, or that the debugging approach performs differently on products of different sizes. To address these threats, we created five different test scenarios with different simulated faults and two different product suites in each case study. Finally, another threat is related to the developed test system simulator, which assumes that test cases and test oracles are always capable of differentiating a faulty product from a non-faulty one. We reiterate, however, that a key requirement for the successful application of SBFL is that test cases are able to reveal the faults to be located. To mitigate this threat, we also evaluated our approach using a real-world case study with real test cases and mutation testing. The results are consistent with those obtained using simulated faults.
External validity: *What are the main limitations of the approach?* As mentioned in Section 3.1.2, if a core feature is faulty, all products will fail, and thus the results of suspiciousness techniques will not be accurate enough to locate the bug. This is an intrinsic problem of SBFL techniques which depend on the existence of both successful and failing tests to identify the suspicious components. To alleviate this threat, when all the products in the product suite fail, core features are placed at the top of the suspiciousness ranking. As another limitation, we considered faults in single features and faults caused by the interaction between two features, as these are common types of

Table 17: Summary of the Results for the Statistical Analysis for the pairwise suite

	Ample2	Arithmetic	Dstar	Kulcynski2	M2	Naish2	Ochiai	Russel-Rao	Tarantula	Wong
Ample2	–	36	13	0	12	28	8	29	1	29
Arithmetic	0	–	3	0	3	6	2	6	0	6
Dstar	0	28	–	0	0	9	0	9	1	9
Kulcynski2	1	39	12	–	12	29	9	29	1	29
M2	0	27	0	0	–	9	0	9	1	9
Naish2	0	7	0	0	0	–	0	0	0	0
Ochiai	0	32	1	0	0	16	–	16	1	16
Russel-Rao	0	7	0	0	0	0	0	–	0	0
Tarantula	1	33	12	0	13	22	9	21	–	21
Wong	0	7	0	0	0	0	0	0	0	–

Table 18: Summary of the Results for the Statistical Analysis for the 3-wise suite

	Ample2	Arithmetic	Dstar	Kulcynski2	M2	Naish2	Ochiai	Russel-Rao	Tarantula	Wong
Ample2	–	42	23	0	24	40	17	40	1	40
Arithmetic	0	–	5	0	5	7	4	7	0	7
Dstar	0	34	–	0	1	23	0	23	1	23
Kulcynski2	6	45	20	–	24	39	17	39	1	39
M2	0	33	0	0	–	17	0	17	1	17
Naish2	0	23	0	0	0	–	0	0	0	0
Ochiai	0	37	10	0	12	35	–	35	1	35
Russel-Rao	0	23	0	0	0	0	0	–	0	0
Tarantula	6	45	22	0	23	38	17	38	–	38
Wong	0	23	0	0	0	0	0	0	0	–

faults in software programs [49]. Thus, evaluating the effectiveness of the approach in isolating faults caused by the interaction among three or more features remains for future work.

In our evaluation, we assumed that the test suite of each faulty product is always able to reveal a failure. We think that this is sensible since the test suite of each product is typically composed of a large number of test cases. If at least one of the test cases exercising the faulty feature(s) reveals the failure, the product is correctly marked as faulty. Thus, we think that this is a minor threat since it is highly unlikely that none of the test cases exercising the faulty feature(s) reveal the failure.

To what extent is it possible to generalize the findings? We used eight case studies, which might not be enough to conclude that some techniques are better than others. To mitigate this threat, we chose case studies from different domains with different sizes and characteristics to assure a sufficient degree of heterogeneity.

Conclusion validity: A possible conclusion validity threat could be the configuration for the Dstar technique. Notice that this technique can be adjusted by setting the *, which is the exponent of N_{CF}. To reduce this threat, we set * to 2 based on previous studies [19].

6. Related Work

In this section, we overview those works closely related to our approach in the fields of SPL testing, SBFL and fault isolation.

6.1. Software product line testing

Recent surveys and mapping studies reveal an increasing interest in SPL testing [50, 51, 52, 6]. Lopez-Herrejon et al. conducted a systematic mapping study on combinatorial interaction testing for SPLs [8]. They identified over forty approaches using different techniques such as genetic and greedy algorithms. They also found that a majority of papers focused on deriving products from variability models (typically a FM) using pairwise testing [53, 5, 54]. Similar to those papers, we leverage the tools for the automated analysis of feature models. In particular, we propose to use the propagate analysis operation, typically used during product configuration, to generate minimal products including the suspicious feature set, easing the isolation of faults. In contrast with previous work, however, this paper focuses on debugging, not testing. Thus our approach does not aim to reveal failures but to locate the bugs that trigger them.

Similarly, a number of papers addressed the problem of product prioritization in SPLs. Most are based on the use of heuristic [11, 55] and search-based algorithms [56, 6, 57, 12] for reordering the products derived from a feature model according to different criteria (e.g., complexity of products). Others have focused on prioritization based on the dissimilarities of products [9][40], following the hypothesis that dissimilar products are better at finding faults. In our case we prioritize feature sets according to their suspiciousness score, which is calculated using state-of-the-art SBFL techniques.

In addition to product prioritization, our fault isolation approach also shares similarities with delta modeling [58]. Delta modeling is an approach used in SPL automated product derivation [58]. It consists of having a core product with a set of features as a basis [58]. To derive new products, different delta operations are applied to the core product [58, 59]. These delta operations consist of (1) adding new features, (2) removing features and (3) modifying features. Our algorithm adds suspicious features to the core features of the SPL and, subsequently, a propagation function adds required features in order to have a valid product. These operations can be considered as part of the delta modeling approach since our algorithm has an initial product composed of the SPL core features. The algorithm is designed this way so that the propagation function increases efficiency. Otherwise, every time the propagation function is called, the core features would be added to derive a valid product.

To the best of our knowledge, SBFL has been applied in the SPL context only in a recent study [31]. Li et al. proposed a search-based approach that generates application engineering level test cases that can be easily reused between different SPL products [31]. Their approach integrates fault localization techniques with the aim to generate more effective test cases when locating bugs. However, while Li et al apply SBFL at code level, in this study we proposed the application of SBFL at feature level in order to isolate feature sets containing faults.

Yilmaz et al. [60, 17] focused on the generation and scheduling of configurations in configurable software (e.g., Linux) for efficient fault characterization. To this end, they proposed two kinds of covering arrays, namely, fixed-strength covering arrays and variable-strength covering arrays [17]. Their empirical evaluation focuses on how different covering arrays perform in fault localization with two case studies. As expected, they found that higher strength covering arrays performed better than lower strength ones. In contrast with their approach, we propose a SBFL approach to locale faulty feature sets in SPLs following a model-based approach (using feature models). Additionally, we assess how different SBFL techniques perform in different test scenarios (i.e., different amount and types of faults, with different product suites). We think, however, that both approaches could be complementary: using their covering array algorithms to generate and prioritize product suites of different strengths, and allow for a faster fault localization in SPLs. Exploring this idea remains for future work.

6.2. *Spectrum-based fault localization*

Several empirical studies have been carried out to assess the performance between different SBFL techniques. Pearson et al. compared the performance of five SBFL techniques and two mutation-based fault localization techniques for both artificial and real faults from five open source projects (JFreeChart, Google Closure compiler, Apache Commons Lang, Apache Commons Math and Joda-Time) [19]. They found that Dstar outperformed the other techniques. They also found that while Tarantula does not perform better than other techniques (e.g., Ochiai) in artificial faults, in real faults there is not a statistically significant difference between Tarantula and the other techniques. Abreu et al. compared Ochiai with Tarantula in the Siemens set, finding that Ochiai performed better [18]. Ochiai is also found to be the best technique in the study performed by Le et al., where the Siemens set, together with NanoXML, XML-security and Space were employed as program subjects [61]. Wong et al. compared 38 techniques on different real-world programs (e.g., Siemens set, grep, make, gzip, etc.), finding that their proposed Dstar technique outperformed the other techniques [20]. Jones and Harrold compared five SBFL and slice-based technique in the Siemens set, finding that Tarantula was the best technique at finding faults [62]. The use of SBFL assumes the use of a test oracle, since the SBFL needs test results. However, as a test oracle is not always available, Xie et al. adapted SBFL to the metamorphic testing context by proposing an approach named metamorphic slice [63]. They compared their approach with three SBFL techniques (Tarantula, Ochiai and Jaccard) in nine programs and found that their approach is as effective as traditional SBFL.

The subject programs of previous studies have always been the source code of program with different languages (e.g., C or Java). In contrast, we propose the application of SBFL in SPLs at the feature level. This is a context in which this technique, to the best of our knowledge, has never been applied. We provided an empirical evaluation that compared ten different suspiciousness techniques in different fault scenarios, across eight case studies of different complexities. Unlike in previous studies, where Dstar, together with Ochiai has been found to be one of the best techniques, we found that in our context the best techniques are Tarantula, Ample2 and Kulcynski2. Moreover, we complement the use of SBFL in the SPL context with a fault isolation algorithm that provides the debugger with the smallest product to help isolate the faulty feature sets.

6.3. *Fault isolation*

Many studies have proposed different techniques for pinpointing faults in computer programs. The core idea of Delta Debugging is simplifying large test cases that produce a fault by removing irrelevant details [28]. The Delta Debugging algorithm has been extended in other studies, proposing a Hierarchical Delta Debugging approach where the input structure is taken into account [64]. This enables reducing the number of test cases and producing smaller outputs. Similar techniques to the Delta Debugging have been proposed for minimizing the constraints on the input parameters to isolate the cause of faults of web applications [65], or for isolating C compiler bugs [66]. Other traditional techniques include program slicing, where irrelevant parts of a failing program are removed [67].

Our fault isolation approach builds upon the aforementioned approaches by employing an incremental approach to build the minimal product (i.e., the core features are taken as a baseline, and the most suspicious feature sets are included to form the smallest product possible) instead of a decremental approach (i.e., isolating fault by making the input space smaller). Moreover, our approach is designed for the SPL context, a context where debugging has been paid little attention. This SPL context faces several idiosyncrasies, such as the use of feature models to manage the variability and the use of reasoning techniques (e.g., SPLAR) to derive valid products.

7. Conclusion and Future Work

In this article we presented a debugging approach for SPLs using SBFL techniques. Based on the features included on each product under test and the test outcomes, it is possible to identify which feature sets were involved in a failure, and which ones did not, narrowing the search for the faulty feature set that made the execution fail. As a result, feature sets are ranked according to their suspiciousness score, assisting debuggers on the localization of bugs. Additionally, we propose to exploit the techniques for the automated analysis of feature models to generate minimal valid products containing the suspicious feature sets, contributing to reduce the effort required to isolate and locate faults. We empirically evaluated our approach by comparing the effectiveness of ten SBFL techniques on eight case studies. Results show that the approach is effective, with the techniques Tarantula, Kulcynski2 and Ample2 showing a good and stable performance with different number and types of faults. We also found that the effectiveness of the technique increases with the number of products under test. This work complements the extensive corpus of papers on SPL testing, and paves the path for new contributions on fault localization in SPLs.

In the future we would like to compare other techniques for fault localization, such as machine learning-based fault localization (similarly as proposed by Yilmaz et al. [17]). In addition, we would like to compare our incremental fault localization approach with the decremental. Furthermore, it could be nice to expand on the empirical evaluation by including more case studies with real faults. In addition, an empirical evaluation involving a controlled experiment with humans could be interesting to better assess our approach in practice. Last, as previously mentioned, our approach is black box. In the future a nice complement to our study could be to use white box information of test cases by using the traceability between feature sets and test cases, which could lead to further benefits.

Experimental results

Experimental results and statistical analysis scripts in R are publicly available at http://bit.ly/IST2018AArrieta

Acknowledgements

Aitor Arrieta, Urtzi Markiegi, Goiuria Sagardui and Leire Etxeberria are part of the embedded systems group of Mondragon Goi Eskola Politeknikoa, supported by the Department of Education, Universities and Research of the Basque Government. Sergio Segura is jointly supported by the European Commission (FEDER) and Spanish Government under CICYT project BELI (TIN2015-70560-R), and the Andalusian Government project COPAS (P12-TIC-1867).

References

[1] P. Clements, L. Northrop, Software Product Lines: Practices and Patterns, SEI Series in Software Engineering, Addison–Wesley, 2001.

[2] D. Batory, Feature models, grammars, and propositional formulas., in: Software Product Lines Conference (SPLC), Vol. 3714 of Lecture Notes in Computer Sciences, Springer–Verlag, 2005, pp. 7–20. doi:10.1007/11554844_3.

[3] K. Kang, S. Cohen, J. Hess, W. Novak, S. Peterson, Feature–Oriented Domain Analysis (FODA) Feasibility Study, Tech. Rep. CMU/SEI-90-TR-21, SEI (1990).

[4] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, D. Beyer, Strategies for product-line verification: case studies and experiments, in: International Conference on Software Engineering, 2013.

[5] G. Perrouin, S. Sen, J. Klein, B. Baudry, Y. le Traon, Automated and scalable t-wise test case generation strategies for software product lines, in: Conference Software Testing, Verification and Validation, 2010.

[6] R. E. Lopez-Herrejon, L. Linsbauer, A. Egyed, A systematic mapping study of search-based software engineering for software product lines, Information and Software Technology 61 (2015) 33 – 51. doi:http://dx.doi.org/10.1016/j.infsof.2015.01.008.

[7] D. Marijan, A. Gotlieb, S. Sen, A. Hervieu, Practical pairwise testing for software product lines, in: Proceedings of the 17th International Software Product Line Conference, SPLC '13, ACM, New York, NY, USA, 2013, pp. 227–235. doi:10.1145/2491627.2491646.

[8] R. E. Lopez-Herrejon, S. Fischer, R. Ramler, A. Egyed, A first systematic mapping study on combinatorial interaction testing for software product lines, in: Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015, 2015, pp. 1–10.

[9] M. Al-Hajjaji, T. Thüm, J. Meinicke, M. Lochau, G. Saake, Similarity-based prioritization in software product-line testing, in: Proceedings of the 18th International Software Product Line Conference - Volume 1, SPLC '14, ACM, New York, NY, USA, 2014, pp. 197–206. doi:10.1145/2648511.2648532.

[10] R. E. Lopez-Herrejon, J. Ferrer, F. Chicano, E. N. Haslinger, A. Egyed, E. Alba, A parallel evolutionary algorithm for prioritized pairwise testing of software product lines, in: D. V. Arnold (Ed.), GECCO, ACM, 2014, pp. 1255–1262.

[11] A. B. Sánchez, S. Segura, A. Ruiz-Cortés, A comparison of test case prioritization criteria for software product lines, in: IEEE International Conference on Software Testing, Verification, and Validation, 2014, pp. 41–50.

[12] S. Wang, D. Buchmann, S. Ali, A. Gotlieb, D. Pradhan, M. Liaaen, Multi-objective test prioritization in software product line testing: An industrial case study, in: Software Product Line Conference, 2014, pp. 32–41.

[13] R. Abreu, P. Zoeteweij, R. Golsteijn, A. J. van Gemund, A practical evaluation of spectrum-based fault localization, Journal of Systems and Software 82 (11) (2009) 1780 – 1792. doi:http://dx.doi.org/10.1016/j.jss.2009.06.035.

[14] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, L. Yi, An empirical investigation of the relationship between spectra differences and regression faults, Software Testing Verification and Reliability 10 (3) (2000) 171–194.

[15] X. Xie, T. Y. Chen, F.-C. Kuo, B. Xu, A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization, ACM Trans. Softw. Eng. Methodol. 22 (4) (2013) 31:1–31:40. doi:10.1145/2522920.2522924.
URL http://doi.acm.org/10.1145/2522920.2522924

[16] W. E. Wong, R. Gao, Y. Li, R. Abreu, F. Wotawa, A survey on software fault localization, IEEE Transactions on Software 42 (8) (2016) 707–740.

[17] C. Yilmaz, M. B. Cohen, A. A. Porter, Covering arrays for efficient fault characterization in complex configuration spaces, IEEE Transactions on Software Engineering 32 (1) (2006) 20–34.

[18] R. Abreu, P. Zoeteweij, A. J. Van Gemund, On the accuracy of spectrum-based fault localization, in: Testing: Academic and Industrial Confer-

ence Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007, IEEE, 2007, pp. 89–98.

[19] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, B. Keller, Evaluating and improving fault localization, in: Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on, IEEE, 2017, pp. 609–620.

[20] W. E. Wong, V. Debroy, R. Gao, Y. Li, The dstar method for effective software fault localization, IEEE Transactions on Reliability 63 (1) (2014) 290–308.

[21] A. Arrieta, G. Sagardui, L. Etxeberria, J. Zander, Automatic generation of test system instances for configurable cyber-physical systems, Software Quality Journal 25 (3) (2017) 1041–1083. `doi:10.1007/s11219-016-9341-7`.

[22] D. Benavides, S. Segura, A. Ruiz-Cortés, Automated analysis of feature models 20 years later: A literature review, Information Systems 35 (6) (2010) 615 – 636. `doi:10.1016/j.is.2010.01.001`.

[23] FaMa Tool Suite. `http://www.isa.us.es/fama/` (Accessed November 2013).

[24] M. Mendonca, M. Branco, D. Cowan, S.P.L.O.T.: Software Product Lines Online Tools, in: Companion to the 24th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM, Orlando, Florida, USA, 2009, pp. 761–762. `doi:10.1145/1639950.1640002`.

[25] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, T. Leich, Featureide: An extensible framework for feature-oriented software development, Science of Computer Programming 79 (2014) 70–85. `doi:10.1016/j.scico.2012.06.002`.

[26] T.-D. B. Le, D. Lo, F. Thung, Should i follow this fault localization tool's output?, Empirical Software Engineering 20 (5) (2015) 1237–1274. `doi:10.1007/s10664-014-9349-1`.

[27] T. Reps, T. Ball, M. Das, J. Larus, The use of program profiling for software maintenance with applications to the year 2000 problem, in: Proceedings of the 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC '97/FSE-5, Springer-Verlag New York, Inc., New York, NY, USA, 1997, pp. 432–449. `doi:10.1145/267895.267925`.
URL `http://dx.doi.org/10.1145/267895.267925`

[28] A. Zeller, R. Hildebrandt, Simplifying and isolating failure-inducing input, IEEE Trans. Softw. Eng. 28 (2) (2002) 183–200. `doi:10.1109/32.988498`.

[29] S. Wang, S. Ali, A. Gotlieb, Minimizing test suites in software product lines using weight-based genetic algorithms, in: Proceedings of the 2013 Genetic and Evolutionary Computation Conference, Amsterdam, Netherlands, 2013, pp. 1493 – 1500.

[30] A. Arrieta, S. Wang, G. Sagardui, L. Etxeberria, Search-based test case selection of cyber-physical system product lines for simulation-based validation, in: Proceedings of the 20th International Systems and Software Product Line Conference, 2016, pp. 297–306.

[31] X. Li, W. E. Wong, R. Gao, L. Hu, S. Hosono, Genetic algorithm-based test generation for software product line with the integration of fault localization techniques, Empirical Software Engineering (2017) 1–51`doi:10.1007/s10664-016-9494-9`.
URL `http://dx.doi.org/10.1007/s10664-016-9494-9`

[32] A. Arrieta, S. Wang, G. Sagardui, L. Etxeberria, Test case prioritization of configurable cyber-physical systems with weight-based search algorithms, in: Proceedings of the Genetic and Evolutionary Computation Conference 2016, GECCO '16, ACM, New York, NY, USA, 2016, pp. 1053–1060. `doi:10.1145/2908812.2908871`.
URL `http://doi.acm.org/10.1145/2908812.2908871`

[33] A. B. Sánchez, S. Segura, A. Ruiz-Cortés, The drupal framework: A case study to evaluate variability testing techniques, in: Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS '14, ACM, New York, NY, USA, 2014, pp. 11:1–11:8. `doi:10.1145/2556624.2556638`.
URL `http://doi.acm.org/10.1145/2556624.2556638`

[34] U. Markiegi, A. Arrieta, G. Sagardui, L. Etxeberria, Search-based product line fault detection allocating test cases iteratively, in: Proceedings of the 21st International Systems and Software Product Line Conference - Volume A, SPLC '17, ACM, New York, NY, USA, 2017, pp. 123–132. `doi:10.1145/3106195.3106210`.
URL `http://doi.acm.org/10.1145/3106195.3106210`

[35] M. F. Johansen, O. Haugen, F. Fleurey, An algorithm for generating t-wise covering arrays from large feature models, in: Proceedings of the 16th International Software Product Line Conference - Volume 1, SPLC '12, ACM, New York, NY, USA, 2012, pp. 46–55. `doi:10.1145/2362536.2362547`.

[36] M. Johansen, Software product line covering array tool (2017).
URL `http://martinfjohansen.com/splcatool/`

[37] G. Sagardui, L. Etxeberria, J. Agirre, A. Arrieta, C. F. Nicolás, J. M. Martín, A configurable validation environment for refactored embedded software: an application to the vertical transport domain, in: ISSRE 2017 (Industry Track): IEEE International Symposium on Software Reliability Engineering, 2017.

[38] F. Ensan, E. Bagheri, D. Gašević, Evolutionary search-based test generation for software product line feature models, in: Proceedings of the 24th International Conference on Advanced Information Systems Engineering, CAiSE'12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 613–628. `doi:10.1007/978-3-642-31095-9_40`.
URL `http://dx.doi.org/10.1007/978-3-642-31095-9_40`

[39] X. Devroey, G. Perrouin, M. Cordy, P.-Y. Schobbens, A. Legay, P. Heymans, Towards statistical prioritization for software product lines testing, in: Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive (VAMOS), 2014.

[40] M. Al-Hajjaji, T. Thüm, M. Lochau, J. Meinicke, G. Saake, Effective product-line testing using similarity-based product prioritization, Software & Systems Modeling (2016) 1–23.

[41] E. Bagheri, F. Ensan, D. Gasevic, Grammar-based test generation for software product line feature models, in: Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research, CASCON '12, IBM Corp., Riverton, NJ, USA, 2012, pp. 87–101.
URL `http://dl.acm.org/citation.cfm?id=2399776.2399785`

[42] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, G. Fraser, Are mutants a valid substitute for real faults in software testing?, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2014, pp. 654–665.

[43] L. T. M. Hanh, N. T. Binh, K. T. Tung, A novel fitness function of metaheuristic algorithms for test data generation for simulink models based on mutation analysis, Journal of Systems and Software 120 (C) (2016) 17–30.

[44] X. Ju, S. Jiang, X. Chen, X. Wang, Y. Zhang, H. Cao, Hsfal: Effective fault localization using hybrid spectrum of full slices and execution slices, Journal of Systems and Software 90 (2014) 3–17.

[45] W. E. Wong, V. Debroy, B. Choi, A family of code coverage-based heuristics for effective fault localization, Journal of Systems and Software 83 (2) (2010) 188–208.

[46] A. Vargha, H. D. Delaney, The kruskal-wallis test and stochastic homogeneity, Journal of Educational and Behavioral Statistics 23 (2) (1998) 170–192.

[47] A. Arcuri, L. Briand, A practical guide for using statistical tests to assess randomized algorithms in software engineering, in: Software Engineering (ICSE), 2011 33rd International Conference on, IEEE, 2011, pp. 1–10.

[48] A. Vargha, H. D. Delaney, A critique and improvement of the cl common language effect size statistics of mcgraw and wong, Journal of Educational and Behavioral Statistics 25 (2) (2000) 101–132.

[49] R. Kuhn, R. Kacker, Y. Lei, J. Hunter, Combinatorial software testing, Computer 42 (2009) 94–96.

[50] P. A. da Mota Silveira Neto, I. do Carmo Machado, J. D. McGregor, E. S. de Almeida, S. R. de Lemos Meira, A systematic mapping study of software product lines testing, Information & Software Technology 53 (5) (2011) 407–423.

[51] I. do Carmo Machado, J. D. McGregor, Y. C. Cavalcanti, E. S. de Almeida, On strategies for testing software product lines: A systematic literature review, Information and Software Technology 56 (10) (2014) 1183 – 1199. `doi:http://dx.doi.org/10.1016/j.infsof.2014.04.002`.

[52] E. Engström, P. Runeson, Software product line testing - a systematic mapping study, Information and Software Technology 53 (1) (2011) 2–13.

[53] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Budry, Y. le Traon, Pairwise testing for software product lines: comparison of two approaches, Software Quality Journal.

[54] M. B. Cohen, M. B. Dwyer, J. Shi, Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach, IEEE Trans. Softw. Eng. 34 (5) (2008) 633–650. doi:10.1109/TSE.2008.50.
URL http://dx.doi.org/10.1109/TSE.2008.50

[55] A. B. Sánchez, S. Segura, J. A. Parejo, A. Ruiz-Cortés, Variability testing in the wild: the drupal case study, Software & Systems Modeling (2015) 1–22.

[56] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, Y. Le Traon, Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines, IEEE Trans. Softw. Eng. 40 (7) (2014) 650–670. doi:10.1109/TSE.2014.2327020.
URL http://dx.doi.org/10.1109/TSE.2014.2327020

[57] J. A. Parejo, A. B. Sánchez, S. Segura, A. Ruiz-Cortés, R. E. Lopez-Herrejon, A. Egyed, Multi-objective test case prioritization in highly configurable systems: A case study, Journal of Systems and Software (2016) –doi:http://dx.doi.org/10.1016/j.jss.2016.09.045.

[58] D. Clarke, M. Helvensteijn, I. Schaefer, Abstract delta modeling, in: Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE '10, ACM, New York, NY, USA, 2010, pp. 13–22. doi:10.1145/1868294.1868298.

[59] R. Lachmann, S. Lity, S. Lischke, S. Beddig, S. Schulze, I. Schaefer, Delta-oriented test case prioritization for integration testing of software product lines, in: Proceedings of the 19th International Conference on Software Product Line, SPLC '15, ACM, New York, NY, USA, 2015, pp. 81–90. doi:10.1145/2791060.2791073.

[60] C. Yilmaz, M. B. Cohen, A. Porter, Covering arrays for efficient fault characterization in complex configuration spaces, in: ACM SIGSOFT Software Engineering Notes, Vol. 29, ACM, 2004, pp. 45–54.

[61] T.-D. B. Le, F. Thung, D. Lo, Theory and practice, do they match? a case with spectrum-based fault localization, in: Software Maintenance (ICSM), 2013 29th IEEE International Conference on, IEEE, 2013, pp. 380–383.

[62] J. A. Jones, M. J. Harrold, Empirical evaluation of the tarantula automatic fault-localization technique, in: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, ACM, 2005, pp. 273–282.

[63] X. Xie, W. E. Wong, T. Y. Chen, B. Xu, Metamorphic slice: An application in spectrum-based fault localization, Information and Software Technology 55 (5) (2013) 866–879.

[64] G. Misherghi, Z. Su, Hdd: Hierarchical delta debugging, in: Proceedings of the 28th International Conference on Software Engineering, ICSE '06, ACM, New York, NY, USA, 2006, pp. 142–151. doi:10.1145/1134285.1134307.
URL http://doi.acm.org/10.1145/1134285.1134307

[65] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, M. D. Ernst, Finding bugs in web applications using dynamic test generation and explicit-state model checking, IEEE Transactions on Software Engineering 36 (4) (2010) 474–494.

[66] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, X. Yang, Test-case reduction for c compiler bugs, in: ACM SIGPLAN Notices, Vol. 47, ACM, 2012, pp. 335–346.

[67] M. Weiser, Programmers use slices when debugging, Commun. ACM 25 (7) (1982) 446–452. doi:10.1145/358557.358577.
URL http://doi.acm.org/10.1145/358557.358577