A semi-automated iterative process for detecting feature interactions

Larissa Rocha
UFBA, Bahia, Brazil
UEFS, Bahia, Brazil
lrsoares@uefs.br

Ivan Machado
UFBA, Bahia, Brazil
ivan.machado@ufba.br

Eduardo Almeida
UFBA, Bahia, Brazil
esa@rise.br

Christian Kästner
Carnegie Mellon University,
Pittsburgh, USA
kaestner@cs.cmu.edu

Sarah Nadi
University of Alberta,
Edmonton, Canada
nadi@ualberta.ca

ABSTRACT

For configurable systems, features developed and tested separately may present a different behavior when combined in a system. Since software products might be composed of thousands of features, developers should guarantee that all valid combinations work properly. However, features can interact in undesired ways, resulting in failures. A feature interaction is an unpredictable behavior that cannot be easily deduced from the individual features involved. We proposed VarXplorer to inspect feature interactions as they are detected and incrementally classify them as benign or problematic. Our approach provides an iterative analysis of feature interactions allowing developers to focus on suspicious cases. In this paper, we present an experimental study to evaluate our iterative process of tests execution. We aim to understand how VarXplorer could be used for a faster and more objective feature interaction analysis. Our results show that VarXplorer may reduce up to 50% the amount of interactions a developer needs to check during the testing process.

CCS CONCEPTS

• **Software and its engineering** → **Feature interaction**; *Dynamic analysis*; Software product lines; Software testing and debugging; *Empirical software validation.*

KEYWORDS

Feature interaction, Runtime Analysis, Configurable Systems, Experimental Study

ACM Reference Format:
Larissa Rocha, Ivan Machado, Eduardo Almeida, Christian Kästner, and Sarah Nadi. 2020. A semi-automated iterative process for detecting feature interactions. In *34th Brazilian Symposium on Software Engineering (SBES '20), October 21–23, 2020, Natal, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3422392.3422418

SBES '20, October 21–23, 2020, Natal, Brazil

ACM ISBN 978-1-4503-8753-8/20/09…$15.00
https://doi.org/10.1145/3422392.3422418

1 INTRODUCTION

A software product might be seen as a configuration of features that need to be composed together without violating their particular requirements. In a highly configurable system, a feature is a product characteristic from user or customer views, which needs to be implemented, tested, delivered, and maintained [10, 16]. Although it might be relatively simple to specify the behavior of a feature in isolation, detecting interactions among features may not be a straightforward task.

A feature interaction is observed when the combined behavior of two or more features differs from the individual behaviors of both features [4]. Features are frequently combined to cooperate and contribute to an intended behavior (expected interactions). However, most interactions cannot be predicted upfront. Unexpected interactions can be classified as either problematic or benign to a system. On the one side, problematic are undesired interactions that may cause faulty or damaging system behavior, such as crashes. On the other side, most interactions, although unexpected, may result in a benign behavior that does not cause problems and might even be essential to coordinate the functionalities of multiple features.

Identifying and classifying feature interactions is challenging as they only appear in certain test cases and configurations (variants of a system composed of different feature combinations). Moreover, writing specifications requires human effort, and testing each feature combination may be unfeasible as the number of configurations and interactions grow exponentially to the number of features [7]. Many systems are composed of a very large number of features, e.g., Eclipse IDE and Linux. The Eclipse IDE[1] has more than 1,600 plugins [26] and the Linux kernel[2] has more than 15,000 configuration options [20]. The features of such products can be combined in different ways generating more potential interactions than there are atoms in the universe.

In an earlier investigation, we proposed an iterative approach to support the feature interaction analysis, named VarXplorer [30, 32, 33]. VarXplorer is an automated, tool supported, process to identify undesired interactions. In contrast to most of the literature approaches, which detects interaction based on upfront specifications [34], we proposed to inspect interactions as they are detected. Thus, we provided a runtime analysis and an inspection process that help developers distinguish intended from unintended interactions.

[1] https://www.eclipse.org/
[2] https://www.kernel.org/

Given a test case (system inputs), VarXplorer generates a *feature interaction graph* (FIG), a concise representation of all pairwise interactions among features. The FIG provides a visualization of which features interact, data context and features relationships, such as the suppression of one feature by another. Through an iterative process of interaction detection, developers and testers are able to analyze FIG from all test cases of a test suite. Furthermore, this process is incremental in the sense that, based on user inspection, the graph is automatically refined by removing benign interactions.

In this paper, we present an exploratory study to investigate the iterative detection process of feature interaction, proposed along with VarXplorer. We are interested in understanding whether the process improves and facilitates the identification of problematic interactions. Additionally, this study investigates how VarXplorer can be used to reduce the effort of identifying and judging interactions from a set of test cases. The effort to analyze interactions is related to the size of the graph: the bigger the graph, the more interactions the developer has to check.

We applied the VarXplorer iterative process on a test suite with 15 test cases. The tests were executed in sequence, from the smallest to the most complex one. Each test generates a FIG, in which developers should explore the interactions and mark them as either benign or problematic based on automatically identified relationships. The results show that the use of FIG reduces up to 50% the amount of information (e.g., interactions and variables) the developer has to analyze during a feature interaction analysis process. Moreover, 40% of the tests presented problematic interactions, which includes 5% of the *require* interactions and 44% of *suppress* interactions as problematic to the system.

2 FEATURE INTERACTION

A feature is developed to satisfy its requirements and should work properly in isolation. However, predicting its behavior when it comes together with other features might be a challenging task. A well-known example in the literature is the *fire and flood control* features for the building-automation system [11]. One the one hand, when a fire is detected, the *fire control* feature activates sprinklers. On the other hand, *flood control* closes off water supply when a given amount of water is detected on the floor. Although those features work fine isolated, together they present a dangerous interaction that may burn the building down. When sensors detect a fire, *fire control* activates sprinklers, but once there is water, *flood control* shuts off the main water supply. The interaction between *fire and flood control* is a pairwise interaction, aka. 2-way interaction.

Feature interaction problems are hard to be identified, especially those that do not lead the system to a crash, but cause a wrong behavior. To detect them, we would need to specify all system configurations. Usually, a test may pass to a given configuration and inputs, but it fails if a parameter is changed [6]. Next, we give more details on specifications and present an overview on how we detect interactions, earlier presented in [33].

2.1 Feature-based Specifications and Global Specifications

Providing specifications for all system configurations does not scale to real systems. The number of possible configurations can potentially be exponential to the number of features. A strategy is to specify features in isolation. A *feature-based specification* describes the behavior of a feature in isolation without any explicit reference to other features [36]. Such behavior is supposed to be preserved independent on other features in the system. Through feature-based specifications, one can detect interaction faults when a feature specification is violated in a configuration [36]. Nevertheless, an undesired interaction may occur without violating feature-based specifications, such as the building automation example illustrated in the previous section. Furthermore, from the whole set of features, it is not clear which combinations of features need to be verified; and developers often hesitate to describe or fail to meet specifications.

Conversely, *global specifications* describe properties for all system configurations. Generally, they represent specifications that fulfill certain functional requirements in all configurations [36]. However, global specifications are unable to define nuances of interactions to infer whether they can indeed affect the behavior of the feature. Moreover, it is difficult to find bugs caused by unintended interactions without any specification. Thus, recently published studies focused on detecting problematic feature interaction from global specifications. These approaches check global specifications based on different strategies, such as model checking [5, 17, 27], sampling [12, 14, 35], and variational execution [13, 22, 24].

Sampling approaches are able to detect configurations that fail, but do not detect unexpected and undesired behaviors. Static analysis tend to over-approximate potential interactions [1, 19]. They identify interactions based on estimated values, only. Other studies execute configurations separately and use symbolic execution to identify interaction problems on control flow [5, 9]. Frequently, configurable systems have bugs not covered by global specifications and bugs that do not result in a crash or other easily observable behavior. In addition, specifications at the feature level are usually missing and the above mentioned approaches may not detect all incorrect system behavior.

2.2 Variational Execution

Variational execution emerges to provide dynamic analysis of the software. It is able to present both control and data flow interactions, and shows concrete variable values to each possible configuration. Since executing one configuration at a time does not scale for large systems, variational execution approaches provide multi-execution to synchronize similar concrete executions [22]. It runs the program with multiple inputs to understand how the differences affect program behavior.

A variational execution aims to execute a test case exhaustively over all configurations of a software product (i.e., all combinations of features or inputs). Since many executions of a system are similar [24], it might not be necessary to run all the possible configurations individually. Variational execution scales to large configuration spaces due to its aggressive sharing abilities of redundancies among the executions of all configurations. As data and control flows are shared, we are able to observe feature interactions in the differences of the execution and assignments of data [22].

When executing a given test case, a variational execution tests all combinations among the features of that test in one single execution. For instance, if we consider a test with three features (A, B, and C), a single execution is able to test the program and all feature combination, enabling and disabling each feature. This process

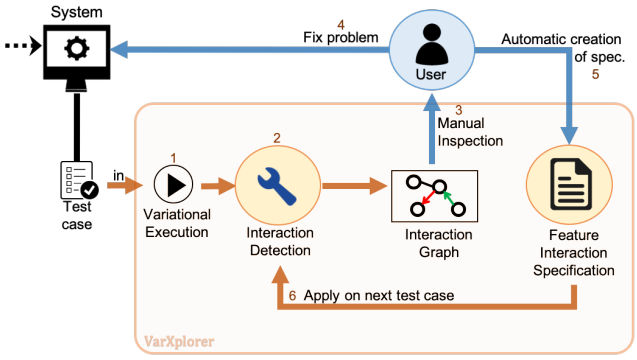


Figure 1: VarXplorer process overview.

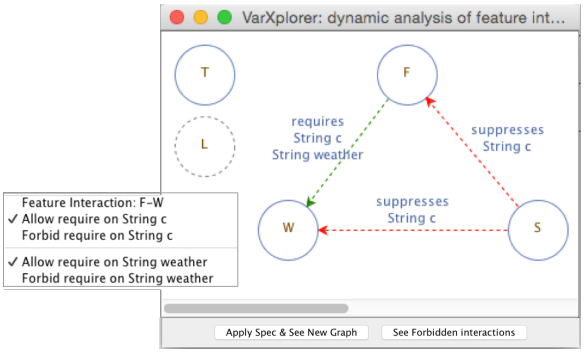


Figure 2: VarXplorer screenshot of the Wordpress graph.

is similar to execute the same test case seven times, one for each possibility among three features (i.e., A, B, C, AB, AC, BC, and ABC). Instead of creating seven tests, the variational execution tests all the seven possibilities in a single execution.

2.3 VarXplorer

VarXplorer [33] is a tool-supported iterative analysis to inspect feature interactions from the variational execution generated by VarexJ, a Java variational interpreter [22]. VarXplorer proposes to identify problematic interactions among pairs of features, which has been proved an effective and practical method to test software [3, 21]. Although empirical evidence shows that higher-order interactions may occur in practice, they are rare compared to pairwise interactions [15]. Especially with variational execution, most failures are related to individual features or pairs of features [22].

Identifying which features interact may not be enough to distinguish between intended and unintended interactions. For example, two features *A* and *B* may collaborate together to deliver some correct system behavior. However, under specific system inputs, the functionality provided by *B* may be suppressed by *A* in an undesired way. To understand the relationship between features, VarXplorer analyses control and data flow interactions to investigate the relation that a feature may have over others, such as suppressing or requiring another feature. We also mark each interaction with additional helpful information, e.g., the affected program variables. The different values that a given variable may assume can be a signal that something wrong occurred. This information is supported by a *feature interaction graph* (FIG), a concise representation of all pairwise interactions among features.

Figure 1 shows an overview of the VarXplorer approach. Based on variational execution, developers start the process by running a test case at a time (Step 1 of Figure 1). VarXplorer dynamically processes the interactions and generates the FIG (Step 2). The FIG shows the relationships among the features activated for that test, i.e., features responsible for the functionalities executed for that test case inputs. From a manual inspection, developers may indicate which relations are benign, those features that are intended to interact, and which ones present a suspicious interaction (Step 3). For unintended interactions, developers may want to fix the problem in the code (Step 4), and interactions marked as benign are removed from the graph. The graph is then refined as more test cases are run, while also taking into account the documented feature interaction specifications (Step 5), which are applied in the next test cases (Step 6).

Feature interaction specifications are automatically generated as developers mark interactions as either benign or problematic in the FIG. Global and feature-based specifications document the behavior of the system and features, respectively. Interaction specifications point out that there exists an interaction between two features, without the need to formally specify its behavior. Furthermore, they help developers focus on potential bugs by automatically removing benign interactions from the FIG. This process of removing benign interaction and highlighting problematic ones, when executing sequential tests, is what characterizes the VarXplorer iterative process. Next, we summarize our detection process and discuss how the feature interaction specifications assist our approach.

2.3.1 ***Feature Interaction Detection.*** In the interaction detection process (step 2, Figure 1), we identify and analyze all pairs of features that interact in a system. The input of the detection is a variational trace created from executing a test case (step 1), and the output is the FIG presenting all the interactions. The trace shows the differences among all configurations for a given test case (specific system inputs). The creation process of the FIG has two major steps: *pairwise detection* and *relationship analysis* [33]. First, we identify the pairs of features that interact from the variational execution and create a basic FIG. Then, we perform the relationship analysis and refine the basic FIG with the relationship between features, including the underlying variables they affect, to produce the complete FIG. A feature interaction occurs when the execution of a line of code depends on two or more features. If this execution leads to a bug, we are dealing with problematic interactions.

Figure 2 shows a screenshot of VarXplorer presenting a FIG generated from a code excerpt modeled after the WordPress system [33]. The FIG shows the two types of feature relationships that VarXplorer identifies, when a feature *suppresses* (red arrow) or *requires* (green arrow) another feature. In order to identify those relationships, VarXplorer collects, from the variational execution, the presence conditions that add or change any functionality during the execution. Presence conditions are propositional expressions over configuration options that determine when a specific code artifact is executed [23]. To analyze relationships, we investigate each pair of features to determine the effect one feature has on the

other. We defined two effects, *suppress* and *require* [33]. A *feature effect* specifies under which condition does a given feature have an effect on the trace. For example, the effect of a given feature *f1* is to have feature *f2* in the same execution (require relation); or for feature *f1* have an effect on the execution, it needs to not have feature *f2* (suppress relation).

The Algorithm on Listing 1 shows how we detect pairs of features and relationships [31]. VarXplorer identifies which features appear together for each presence condition to determine *suppress* and *require* relationships. First, we use binary decision diagrams (BDD) to get the set of valid features (Line 92 of Listing 1). From the set of expressions (gathered from VarexJ), we identify each pair of feature that interact (line 93). We also combine the features in pairs to look for relationships (lines 95 and 100). Then, for each feature, we look for its effect over others from the set of expressions (line 96). Finally, we check the implications [33] to detect relationships on a pair of features based on feature effects, either suppress or require, lines 101 and 102, respectively. This algorithm returns all pairs of features that interact and relationships, if any (line 115).

```
90  getInteractions(expressions){
91
92   featuresSet = BDD.getFeatures;//get set of features
93   exprPairs = getExpressionsPairs(expressions);
94
95   for each feature1 in featuresSet {//get 1st feature
         for pair comparison
96    unique = createUnique(feature1, expressions);//get
         effect of given feature1 on the expressions set
97    if (isContradiction(unique)) //when a feature does
         not appear in the expressions
98     continue;
99
100   for each feature2 in featuresSet {//get 2nd feature
101    suppressAnalysis = feature2.implies(unique.not());
102    requireAnalysis = feature2.not().implies(unique.not
          ());
103
104    pair = new PairExp(feature1,feature2);//creates pair
105    if (suppressAnalysis.isTautology())
106     phrase = "feature2 suppresses feature1";
107
108    if (requireAnalysis.isTautology())
109     phrase = "feature1 requires feature2";
110   }
111   if relationships were not found in the pair
112       phrase = "feature1 and feature2 do not interact";
113   interactionList.add(pair, phrase);
114  }
115  return interactionList;} //final list of relationships
```

Listing 1: Simplified algorithm for interactions detection

2.3.2 ***Feature Interaction Specification Language.*** The FIG shows all data and control flow interactions based on a test case, which represents a given set of system inputs. To better inspect all the possible interactions in a system, the feature interaction detection should be applied over different inputs, and different tests to achieve a high system coverage. However, when applied over real systems, the graphs may present a large amount of interactions and conditional variables, variables that depend of at least two features.

Nonetheless, different graphs from different test cases may share the same interactions. Although the input may be different, some pairs of feature may interact in the same way, as for example, overwriting the same variables with the same values. Hence, we proposed the *feature interaction specification language* to support developers to either *allow* or *forbid* interactions in a configurable system [33]. The interaction language is a lightweight strategy to indicate that there is an interaction among features. It does not require a formal description of the behavior of systems or features, as global and feature-based specifications do.

For example, users can right click on the graph to specify that an interaction is benign (*allowed*), which is then automatically added to the specification. When allowing, developers remove interactions from features that are intended to interact and present a benign behavior, which "cleans" the graph and facilitates finding interactions that may represent a bug. Otherwise, an interaction flagged as forbidden in a graph can be tracked throughout all test cases executions to point out the cases when it may occur. In particular, specifications can be created according to three parameters: type (*Allow, Forbid*), relationship (*Require, Suppress, Any*), and target (*Variable, Method, Class, Any*). The screenshot of Figure 2 shows the user possibilities to the *require interaction* between features *F* and *W*, i.e., either *allow* or *forbid* interactions on `String c` and `String weather`. Listing 2 shows an example of an automatically generated specification, created when the user *allows require* on `String c` and `String weather`.

```
1  <system name="WordPress">
2   <specification type="allow">
3    <require from="F" to="W">
4        <var name="weather"/>
5        <var name="c"/>
6    </require>
7   </specification>
8  </system>
```

Listing 2: Example of interaction specification

2.3.3 ***Iterative Process.*** From the test suite of a configurable system, VarXplorer proposes to run one test at a time starting from the smallest to the most complete one. Small tests produce small graphs which are probably easier to inspect than complex tests. VarXplorer proposes to facilitate the detection of problematic interactions. Thus, when developers get to execute more elaborate tests, many benign interaction would have already been removed, making complex graphs into graphs easier to analyze. The goal behind incremental removal of intended interactions is to make the analysis of tests simpler and faster, focusing on newly unintended interactions.

Hence, given a test case, VarXplorer generates the corresponding FIG and shows it to developers for manual inspection. From the graph, they can select an interaction, right click on the line that connects two features, to allow benign interactions or mark others as suspicious. From this process, developers automatically create feature interaction specifications that are documented and used for future test cases. Then, the FIG is automatically refined by removing benign interactions over test cases. After marking an interaction as benign, developer do not see that benign interactions again in future graphs. In case of finding problematic interactions (bugs), developers may want to solve the problem in the source code and also mark those interactions as suspicious in the graph, forbidding them to occur again. If previously analyzed suspicious interactions appear in future test cases, they are highlighted in the graph to alert users and make it easy to recognize them.

Table 1: Experiment Design

Preparatory Steps			
#	**Activities**	**Objective**	**Involved Subjects**
1	Practical Session	Configure environment	SPL developers and VarXplorer expert
2	Brainstorm	Define scenarios	SPL developers and VarXplorer expert
3	Implementation	Tests development	VarXplorer expert
4	Validation	Tests validation	SPL developers and VarXplorer expert
Actual Experiment			
1	Tests 1 to 7	Execution and analysis of tests 1 to 7	VarXplorer expert
2	Verification	Validation of bugs and interactions found	SPL developers and VarXplorer expert
3	Tests 8 to 15	Execution and analysis of tests 8 to 15	VarXplorer expert
4	Verification	Validation of bugs and interactions found	SPL developers and VarXplorer expert

3 EXPLORATORY STUDY

We performed an exploratory study to investigate how the iterative and interactive approach may support the discovery of suspicious interactions. We evaluate our proposed approach by answering the following research question:

RQ: How does the iterative process on individual test cases reduce the complexity of identifying interactions?

Iterativeness stands for the potential to optimize the feature interaction detection through short iterations in sequence and each iteration has a self-contained program scenario, composed of one test case analysis. The iterative analysis consists of executing in sequence all test cases of a test suite. A single test case is analyzed at a time, which produces one FIG with all possible interactions and relationships among the features for the given scenario. When looking at all tests at once one gets overwhelmed with warnings, but when looking at one test at a time the analysis remains reasonable and guides the effort.

In this study, we are interested in understanding how much effort we save using interaction specifications when executing test cases. Whenever a developer *allows* or *forbid* an interaction, the tool creates the corresponding specification and saves it in the system. When executing a given test case, VarXplorer always applies specifications created in previous FIG of that system to create the current FIG. Thus, we compare the size of FIG without applying specifications (complete graph) versus the reduced graph, when known interactions are removed. Consequently, we measure how many interactions are removed during the iterative process.

3.1 Subject System

We applied the VarXplorer on the RiSE Event SPL project [8], a SPL for papers/reports submission. The SPL was built based on the main features found on largely used conference management systems, such as: EasyChair[3], JEMS[4], and CyberChair[5]. The source code of the SPL project is annotated according to its features to generate different SPL products. VarXplorer reads the annotated code and collects the presence conditions that add or change any functionality during the execution. The RiSE Event SPL project was developed in Java, composed of 20 functional features, 26.457 Lines of Code, 1493 Methods and 496 Classes. The features are named as follows: *Event*, *User*, *Reviewer*, *Speaker*, *Author*, *Activity*, *ActivityTutorial*, *ActivityWorkshop*, *Registration*, *RegistrationSpeakerActivity*, *RegistrationUserActivity*, *CompleteSubmission*, *PartialSubmission*, *Review*, *Payment*, *PaymentCash*, *PaymentDeposit*, *PaymentCard*, *Assignment*, and *ConflictOfInterest*.

[3]www.easychair.org/
[4]jems.sbc.org.br
[5]www.borbala.com/cyberchair/

3.2 Study Overview

Since VarXplorer approach consists of executing test cases incrementally, the process starts by running a small test, which tends to generate a small graph. Then, we incrementally increase the test complexity (increasing number of features and test activities) until we have tested all system functionalities. VarXplorer proposes to run test cases in sequence, in which every test case is run after the analysis of its preceding case. Thus, benign interactions of previous tests are automatically applied on the current test, which may reduce the FIG and simplify its analysis.

We execute each test using VarXplorer and analyze the FIG. The developers should check each interaction and mark it as either benign or problematic. When they mark the graph, it automatically creates interaction specifications to document the benign and problematic interactions. VarXplorer also shows partial interactions, the ones related to variables, which can be analyzed separately for each variable. In case of finding problematic interactions in a graph, they may first need to fix the problem at source code and then, run the test again to check how the features are interacting. We repeat this process until we believe the test does not contain any other problematic interaction. The tests were executed using the Eclipse IDE (Oxygen version) and a laptop with 2.3GHz i5, and 16 GB DDR3.

3.3 Design and Procedure

Preparatory steps. Before running the actual study, we performed a set of preparatory steps, as Table 1 shows. The study was conducted by the main researcher of this paper, supported by the others. First, we had a practical session with the RiSE Event SPL developers to configure the Eclipse environment and database. The SPL project did not have any test cases available for this study. Thus, we performed a brainstorming session with the SPL developers to discuss the possible scenarios and how the tests would be built. The brainstorm session took around 2 hours. Then, we (researchers) implemented the suggested tests in Java and validated them with the developers to certify whether the tests would achieve the proposed goals (features coverage). The test suite was composed of 15 tests (T1-T15) developed in about 8 hours, and the validation session took 2 hours. Table 2 presents the list of tests.

To create the tests, we started with small scenarios composed of few features and incrementally increased the tests in terms of

Table 2: Details of the Test Case Suite

Test	F	Description
T1	4	It creates and updates events, and updates activities
T2	4	In addition to 1st test, it also creates activities
T3	3	It adds a registered speaker to a registered activity
T4	4	In addition to 3rd test, it creates new speakers
T5	4	In addition to 4th test, it adds new speaker to a registered activity
T6	2	It creates reviewers
T7	5	In addition to 6th test, creates a new submission
T8	6	In addition to 7th test, attributes new submissions to users
T9	7	In addition to 8th test, attributes submissions to authors
T10	7	In addition to 9th test, creates reviews to submissions
T11	4	It registers an user in an activity
T12	8	In addition to 11th test, registers a payment
T13	6	It creates assignments to different reviewers
T14	7	In addition to 13th test, checks conflicts between authors and reviewers
T15	8	In addition to 14th test, send messages to reviewers

F: number of features;

complexity and number of features, as Table 2 shows. For example, one test creates events, and the next one creates and updates events, in the same test scenario. The saturation was achieved when we had tested at least all features once and the creation of new test cases did not generate any new interaction. Thus, the test suite was composed of (i) a mix of tests with the same features, but different inputs; (ii) tests with features not tested in previous tests; and (iii) tests with a mix of non-tested features and features already tested. Moreover, the test suite does not need to contain one test to each configuration. VarXplorer extends a variational interpreter, VarexJ, which is able to exhaustively execute a test case over all configurations of a software product, sharing redundancies of the executions. Consequently, the variational execution reduces the need for additional tests.

Additionally, before running the tests, we asked the SPL developers to fill out the SPL database with some initial information. They created 4 events, 4 authors, 5 users, 4 activities, 2 speakers, 3 reviewers, 4 reviews, 4 submissions, 3 payments, 2 registrations, and 3 assignments. The test cases should manage this information in the database, searching, changing or creating new data.

Study execution. We first executed tests 1 to 7 (T1-T7) and then performed a verification session with the SPL developers to show them the information gathered, i.e., interactions, problems, causes, and solutions (fixes at source code). The same process was conducted for tests 8 to 15, we first executed and analyzed the tests and then validated them. This experiment took 40 hours and included: (i) the execution and analysis of the 15 FIG; and (ii) two validation sessions with the SPL developers, as Table 1 shows.

The analysis procedure consisted of running each test separately. The flowchart in Figure 3 synthesizes the sequence of steps we followed. When a test is executed, the corresponding FIG is created (Step 1 on Figure 3). To analyze interactions, developers have to check the graph and judge them as either benign or problematic (Step 2). If developers identify suspicious interactions, we suggest them to look for the causes of the problem (Step 3) and fix them in the source code (Step 4) before running the next test case. Then, they may have to run the test again (Step 1) and check the graph after the fix to guarantee the problem has been fixed. At this point, the graph created after the fix should not contain the same interactions, once the source code may have changed. Thus, it may contain new interactions, and prior interactions related to the fixed bug may

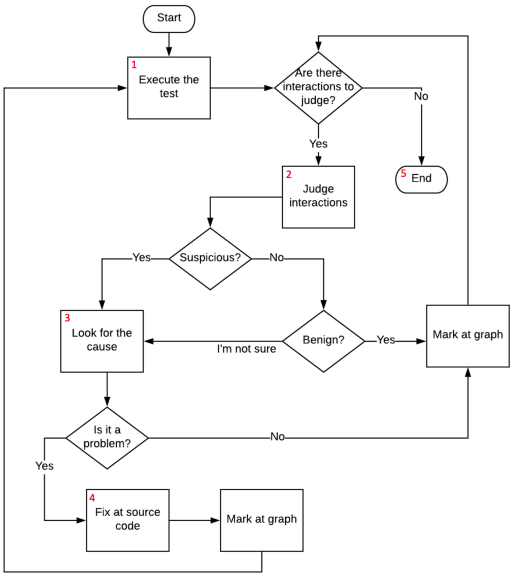


Figure 3: Flow chart of the process to analyze interactions.

Table 3: Interactions type, variables and time of analysis

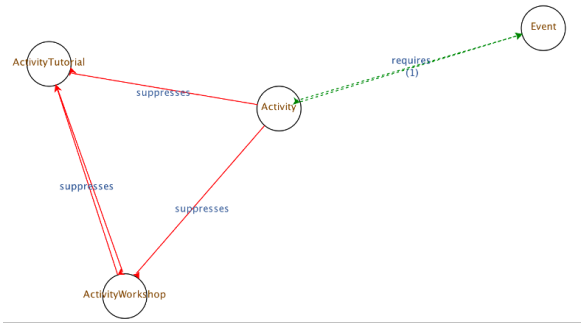
Test	Interactions				Time
	# Require	# Suppress	Total	Variables involved	
T1	2	4	6 (6)	17	8 hours
T2	6	1	7 (7)	7	1 hour
T3	4	0	4 (4)	6	30 min
T4	7	0	7 (8)	43	20 min
T5	10	4	14 (15)	56	20 min
T6	2	0	2 (2)	3	5 min
T7	4	2	6 (8)	32	10 min
T8	10	2	12 (14)	32	10 min
T9	4	2	6 (15)	22	5 min
T10	6	2	8 (19)	30	5 min
T11	12	0	12 (12)	30	15 min
T12	18	4	22 (32)	94	20 min
T13	8	1	9 (9)	13	5 min
T14	9	2	11 (20)	17	15 min
T15	16	1	17 (29)	29	10 min

#Require: number of interactions of *require* type; **#Suppress**: number of interactions of *suppress* type; **Total**: x (y), x means number of interactions shown in the graph after applying specifications, and y means number of interactions if no specification is applied; **Variables involved**: number of conditional variables; **Time**: Time spent to analyse each graph, judge as benign or problematic, and fix problems at source code (if any).

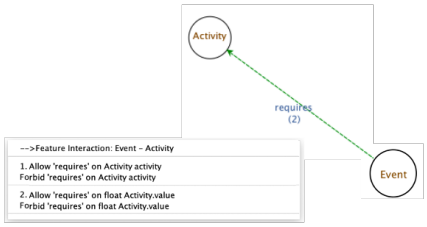
not appear anymore. Hence, this graph need to be checked again, and the process should restarts. The interactions of this new graph should judged until all interactions have been understood (Step 5).

4 RESULTS

Table 3 shows the type of interactions found for each test, conditional variables identified, and execution length. The tests ranged from 2 to 8 features each. The smallest graph (T6) has 2 interactions, and the biggest one (T12) has 32 interactions. For instance, T15 has 29 interactions in total, but since the graph was simplified based on the specifications built from T1 to T14 analysis, it was reduced to 17 interactions, in which 16 out of them are *require* interactions and 1 is a *suppress* interaction. The FIG of T15 also shows 29 conditional variables involved in the interactions.



(a) First Graph



(b) Graph after fixing the problem

(c) After applying specifications

Figure 4: Graphs generated for the analysis of T1.

The test suite presented a total of 431 conditional variables and 143 different interactions. 118 are *require* interactions and 25 are *suppress* interactions, scattered over 15 tests. Eleven *suppress* interactions and six *require* interactions were signal of bugs. For the RiSE Event test suite, each feature interacts at least with 3 other features, although others presented a higher level of interaction. The features `Activity` and `User` are the ones that interact the most, with 10 features each.

VarXplorer proposes to analyze the test cases incrementally, from the smallest test to the biggest one. Thus, we first executed T1. It aims to create events, update recently created events, and update the price of an activity. Figure 4a shows the execution of T1. Although the FIG shows 4 interacting features, based on the developers domain knowledge, T1 should only contain interactions between 2 features (`Activity` and `Event`). The other 2 features should neither has effect nor interact with others. Furthermore, the feature `Activity` should not suppress the behavior of any other feature of the system. The SPL Developers recognized those interactions as suspicious, presenting a high chance of representing a bug.

Although we are able to identify interactions as suspicious based on the FIG, it is not enough to understand what triggered the interactions. The FIG was proposed to support developers and provide a fast identification of problematic interactions, but the source code should still be used to fix the problem. To understand the cause of the suspicious interactions of T1, developers may want to analyze the execution trace and the source code, and also use the Eclipse debugging tool. For T1, we used all the available resources

Table 4: RiSE Event SPL interactions

Test	Bugs Fixed		Spec Cleaned	% of Reduction	
	Require	Suppress		Int.	Vars
T1	0	2	0	0,00%	0,00%
T2	0	1	5v (1i)	0,00%	45.45%
T3	0	0	0	0,00%	0,00%
T4	0	0	1i, 9v (3i)	12.5%	4.92%
T5	0	4	1i, 51v (8i)	6.66%	47.66%
T6	0	0	0	0,00%	0,00%
T7	0	1	2i, 3v (2i)	25,00%	27.27%
T8	0	0	2i, 32v (7i)	14.29%	47.76%
T9	0	0	9i, 66v (13i)	60,00%	70.21%
T10	0	0	11i, 58v (16i)	57.89%	44.96%
T11	0	0	0	0,00%	0,00%
T12	6	2	10i, 30v (12i)	31.25%	31.58%
T13	0	0	4i, 6v (4i)	30.77%	31.58%
T14	0	1	9i, 17v (11i)	45,00%	53.13%
T15	0	0	12i, 25v (14i)	41.38%	50,00%

Require: number of problematic interactions of type *require*; **Suppress**: number of problematic interactions of type *suppress*; **Spec Cleaned**: number of specifications removed from the graph after applying specifications; in which [*xi*, *yv* (*zi*)] means *x* interactions and *y* variables removed from *z* interactions; **Int**: percentage of interactions removed from the graph; **Vars**: percentage of variables removed from the graph.

to look for the problem, which was an incorrect expression in a source code annotation implemented by the SPL developers. Due to the wrong annotation, the program was calling unnecessarily routines and interactions. Since the RiSE Event is a configurable system, the wrong annotation would create invalid products.

T1 was the first test case of our analysis. It took longer to be understood than the other tests. After finding the problem and fixing it at source code, we came back to the graph and marked the suspicious interactions as *forbid*. In order to confirm that those interactions had been fixed, we ran the test case again. Figure 4b shows this new graph, which has one interaction: feature `Event` *requires* feature `Activity` regarding 2 variables (*activity* and `value`). According to our analysis, this interaction behaves as expected and then, it was marked as benign (*allow* in the graph). Benign interactions are automatically removed, as Figure 4c shows.

Listing 3 shows the set of specifications automatically created after finishing the analysis of T1: two interactions marked as *forbid* and one interaction that includes two variables marked as *allow*. At the end of the analysis of all tests, the set of specifications should contain all the specifications created from T1 to T15. The analysis of the RiSE Event SPL created a total of 286 specifications. This number is high because an interaction is created to each variable analyzed by the user.

```
1 <system name="RiSE Event SPL">
2  <specification type="Forbid"><suppress from="Activity" to="
       ActivityTutorial"/> </specification>
3  <specification type="Forbid"><suppress from="Activity" to="
       ActivityWorkshop"/> </specification>
4  <specification type="Allow"><require from="Event" to="Activity"><
       var name="Activity activity"/></require> </specification>
5  <specification type="Allow"><require from="Event" to="Activity"><
       var name="float Activity.value"/></require></specification>
6 </system>
```

Listing 3: Specifications automatically created for T1.

Before T1 be executed, VarXplorer had no feature interaction specifications, the first specification were created during the T1 analysis. From the second test case, VarXplorer had specifications and could apply them to reduce the next graphs. This process cleans all benign interactions, if repeated in the tests. For example, T9 has

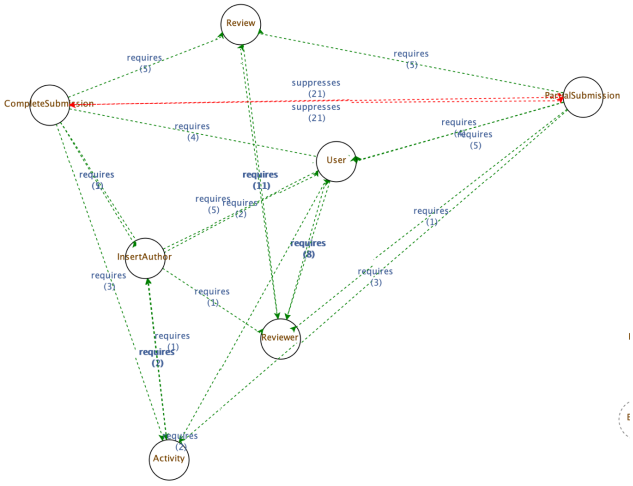


Figure 5: Complete graph of T10.

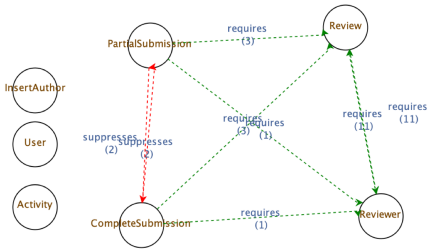


Figure 6: Reduced graph of T10.

15 interactions, but the graph showed 6 because 9 were removed from specifications created in previous tests (T1-T8).

Table 3 and Table 4 show the number of interactions and variables cleaned during the testing process (T1-T15). For instance, although the graph of T10 presents 19 interactions, 11 were removed from specifications. The developer had only 8 to judge in the reduced graph. At the same test, 58 variables of 16 different interactions (*11i, 58v (16i)*) were removed from specifications, as Table 4 shows. VarXplorer also allows the developer to see both graphs, the reduced and the complete one. For T10, the developer had 57.9% less interactions and 45% less variables to analyze when the specifications are applied.

Figures 5 and 6 show both graphs of T10, complete and reduced, respectively. The reduced graph is much smaller (less interactions and variables) than the complete one. It only shows the new interactions, which occurred for the first time in the iterative process. For example, all interactions related to `User`, `InsertAuthor`, and `Activity` were removed because they were judged as benign during the analysis of T9. We next summarize the RQ main findings:

RQ Findings: In general, by using the iterative process and the specifications, we had 45% less interactions to judge and 50% less variables, in average. Graphs with 0% of reduction (T1, T3, T6, and T11) represent new scenarios containing features not tested in previous tests during the analysis, consequently, interactions not seen in previous tests.

5 DISCUSSION

We executed a test suite with 15 different tests and scenarios. zAs test cases grow and features are repeated among tests, many interactions are repeated and, thus, they are no longer showed. The graph then shows only the new interactions, getting smaller and simpler to interpret and detect problematic interactions.

From the 143 interactions found in the SPL analysis, 17 were undesired according to the SPL developers (12%). Thus, 5% of the *require* interactions (6 out 118) and 44% of *suppress* interactions (11 out 25) were problematic to the system. The 17 undesired interaction were spread across 6 out of 15 test cases, which shows that 40% of the total of tests presented problems. Most of the problems were related to *suppress* interactions. Only 1 graph out of 15 presented problems related to *require* interactions (T12, as Table 4 shows).

Our results show that the feature relationships (require and suppress) may indicate the presence of undesired interactions, which assist developers during feature interaction analysis. The RiSE Event SPL used in this study is a stable and working project, which had been finished when we studied its interactions. Developers are expected to implement the features to interact in a desired way to meet SPL requirements. It was expected that most of the interactions of the FIG were benign.

The SPL presented 5 types of problems, as follows: (i) **Bad Annotation**, incorrect annotations in expressions, it creates wrong interactions and may generate malformed products; (ii) **Misplaced Object**, an object used in place of another. The project is based on MVC pattern and we found methods probably copied from other classes, in which a given part of it was incorrectly maintained; (iii) **Misplaced Variable Overwrite**, when the value of a variable is overwritten in a wrong place with a wrong value; (iv) **Conditional Statement**, incorrect implementation of conditional statements, using wrong or incomplete conditions; (v) **Spread Code**, piece of code spread over different features leading to unnecessary dependencies and code modularity issues.

We did not find any problem that could led the program to a crash. However, some problems, such as *misplaced object*, *conditional statement*, and *misplaced overwrite* may lead to a wrong behavior. For example, in T7, we observed that the system requires all article information before submitting it, although the program should allow partial submissions with incomplete information. This situation is contrary to the system requirements. Other problems may also provide unnecessary dependency and impair software maintenance/evolution.

VarXplorer proposes to execute ordered tests, from the smallest to the largest graph. Thus, from T1 to T15 for the RiSE Event project. However, after the analysis in sequence from T1 to T15, we created and run three other sets of the same tests randomly organized. We aim to analyze whether the order of the tests have any influence in the results. Table 5 shows the 3 sets and the additional effort (*Ad. Effort*) needed for each analysis, in which `R` means *random*. The value *zero (0)* in Table 5 stands for no additional effort used to judge the graph, i.e., either it has the same number or less interactions than the same graph of the ordered analysis.

For the original set of tests analyzed in an orderly way, each subsequent test is reduced due to benign interactions identified in previous tests. However, when we change the order, we do not

Table 5: Three random samples of the test suite

1st R set	Ad. Effort	2nd R set	Ad. Effort	3rd R set	Ad. Effort
T8	+2i 25 v	T10	+11i 58v	T8	+2i 32v
T14	+10i 15v	T6	0	T3	0
T3	0	T9	0	T12	+10i 30v
T10	+2i 10v	T3	0	T5	+4v
T2	+7v	T14	+9i 17v	T13	+4i 6v
T13	0	T2	+9v	T15	+5i 12v
T7	0	T15	0	T10	+9i 15v
T5	+7v	T7	0	T1	0
T9	0	T5	+36v	T6	0
T1	0	T11	0	T9	0
T15	0	T1	0	T11	0
T4	0	T4	0	T14	0
T11	0	T12	0	T2	0
T6	0	T13	0	T4	0
T12	0	T8	0	T7	0

guarantee that smaller tests come first and the analysis may have a higher initial effort. For example, T8 was the first test analyzed in the first random set (1st R set) (Table 5). For that analysis, T8 had 2 interactions and 25 more variables (+2i 25v) compared to the original test showed in Table 3. When the tests were executed incrementally, T5, T6 and T7 were executed before T8, which reduced the T8 graph.

When large graphs are executed first and out of order, users have initially much more information to check. The number of removed interactions is reduced and the complexity of the analysis increases, the developer may see much more interactions in a single graph. Although the initial effort may be higher, the overall effort was the same for all sets of tests, regardless the order. The effort is related to the number of interactions to analyze. Therefore, the amount of unique interactions and variables in each set of test cases maintained the same. We did not find any new interaction or problem due to variations on the order of execution.

6 THREATS TO VALIDITY

Construct Validity. This study was performed by one researcher using a third party system, with the support of two RiSE Event SPL developers. Before the tests execution, the researcher had unlimited access to the system requirements and source code for over two weeks. During those weeks, we performed several practical sessions with the presence of the developers. Regarding the creation of test cases, we asked the developers to list any unit tests they thought needed to be implemented. Then, we implemented the tests they suggested. In addition, the SPL developers were available most of the time to answer questions in person, email or chat. We also realized other meetings to present intermediary results and validate the identified interaction problems. The solutions to the problems at source code were also discussed with them.

Internal Validity. VarXplorer relies on running test suites to draw its conclusions. There is no systematic process to create a test suite to discover feature interactions with variational execution. The test suite of this study was defined with the developers, and the saturation was achieved when variations in the defined scenarios did not bring new interactions. Therefore, the test suite has reasonable coverage, but it may not be complete. Although we potentially may miss interactions that occur only with other specific scenarios, we executed the program with representative inputs, which covers all system requirements.

External Validity. The RiSE Event SPL had no test case implemented before this study. We implemented 15 test cases and found 11 suspicious interactions, all of them were fixed at source code. In this study, we did not compared VarXplorer to other tool. However, an earlier study concluded that developers were much faster at identifying interaction problems when using VarXplorer than with another similar tool [32]. Nonetheless, the reader must be careful when generalizing results beyond the studied system.

7 RELATED WORK

The literature presents three main categories of feature interaction approaches, i.e., detection, resolution, and analysis of interactions [34]. *Detection* approaches create strategies to identify feature interactions; resolution approaches focus on solving the interaction problem to deliver the desired products; and analysis approaches encompass the rest, any study that models, specifies, or discusses feature interaction. VarXplorer belongs to the detection group. Within this category, some approaches deal with feature interactions based on static analysis [2, 28, 29] instead of performing a variability-aware execution. Although recent efforts, static analysis of systems with high accuracy remains challenging [18]. Conversely, we use variational execution, a dynamic analysis that execute a test case exhaustively over all configurations of a software product, i.e., all combinations of features or inputs [22].

Similarly to our study, Nguyen et al. [25] propose iGen, a dynamic analysis tool to automatically discover feature interactions. From a sampling-based strategy focused on code coverage, they create a minimal set of configurations to detect the location on source code where features interact. However, sampling strategies execute the system only in selected configurations, they are not configuration complete, and may miss interactions [22]. Despite they are able to detect configurations that fail, they may not detect unexpected and undesired behaviors. Bugs that do not cause crashes might be as critical as the ones that lead the system to fail. Instead, we perform variational execution that explores all possible configurations from a test case and investigate interaction behaviors from feature relationships.

Regarding tools evaluation, both VarXplorer and iGen manually create test suites for the systems under study. Nonetheless, iGen only detects raw interactions, which may be not enough to support developers on identify whether an interaction is desired or represents a bug. VarXplorer provides the FIG, a representation of feature relationships, to support detecting problematic interactions.

Meinicke et al. [22] propose Varviz, a dynamic approach to test different executions focused on configuration complexity. Varviz identifies where features interact, but does not provide a way to distinguish which ones are problematic to the system. In addition, it presents much more information that is not related to the interactions. Features are likely to interact many times and in many different ways. Just showing that features interact does not provide sufficient insights to support developers to find and fix problems. As far as we know, no work provides explicit information about feature interactions, as we do with suppress and require relationships.

8 CONCLUDING REMARKS

VarXplorer provides a dynamic inspection process to detect any feature interaction problem that causes differences in control and

data flows of the system. For this study, we used VarXplorer to find feature interaction problems in a third part system. Most of the problems were related to *suppress* interactions. 40% of the tests presented feature interaction problems. However, as expected, the number of problematic interactions was much lower than benign interactions.

Most of the problems identified concern the lack of source code modularity and incorrect implementation, such as, incorrect variable overwrite and misalignment of *if* statements. When those problems only appear in the combination of features, they are harder to be identified by common strategies, because they need to test *all* interactions. The results confirm that the analysis of individual and ordered test cases and the use of reduced graphs (with automatic generated specifications) led to a reduction of 45% less interactions to judge in median; and 50% less variables when compared to complete graphs. Furthermore, 44% of suppress interactions presented problems, which show that the feature relationships of the FIG might be indicatives of buggy interactions. As future work, we intend to investigate tests generation approaches suitable for runtime analysis of feature interactions.

REFERENCES

[1] Florian Angerer, Andreas Grimmer, Herbert Prähofer, and Paul Grünbacher. 2016. Configuration-Aware Change Impact Analysis. In *30th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*, Vol. 00. 385–395.

[2] Sven Apel, Alexander Von Rhein, Thomas Thüm, and Christian Kästner. 2013. Feature-interaction detection based on feature-based specifications. *Computer Networks* 57, 12 (2013), 2399–2409.

[3] H. Avila-George, J. Torres-Jimenez, and I. Izquierdo-Marquez. 2018. Improved pairwise test suites for non-prime-power orders. *IET Software* 12, 3 (2018), 215–224.

[4] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. 2003. Feature interaction: a critical review and considered forecast. *Computer Networks* 41, 1 (2003), 115 – 141.

[5] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. 2011. Symbolic Model Checking of Software Product Lines. In *33rd Int. Conf. on Software Engineering* (Waikiki, USA) *(ICSE '11)*. ACM, New York, NY, USA, 321–330.

[6] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. 2007. Interaction Testing of Highly-configurable Systems in the Presence of Constraints. In *Int. Symposium on Software Testing and Analysis* (London, United Kingdom) *(ISSTA '07)*. ACM, New York, NY, USA, 129–139.

[7] M. B. Cohen, M. B. Dwyer, and J. Shi. 2008. Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach. *IEEE Transactions on Software Engineering* 34, 5 (Sept 2008), 633–650.

[8] P. A. d. M. S. Neto, T. L. d. Santana, E. S. d. Almeida, and Y. C. Cavalcanti. 2016. RiSE Events — A Testbed for Software Product Lines Experimentation. In *2016 IEEE/ACM 1st Int. Workshop on Variability and Complexity in Software Design (VACE)*. 12–13. https://doi.org/10.1109/VACE.2016.011

[9] M Hentschel, R Hähnle, and R Bubel. 2016. The Interactive Verification Debugger: Effective Understanding of Interactive Proof Attempts. In *31st IEEE/ACM Int. Conf. on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. ACM, New York, NY, USA, 846–851.

[10] Kyo C Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. 1998. FORM: A feature-; oriented reuse method with domain-; specific reference architectures. *Annals of software engineering* 5, 1 (1998), 143.

[11] Kyo C. Kang, Jaejoon Lee, and Patrick Donohoe. 2002. Feature-Oriented Project Line Engineering. *IEEE Softw.* 19, 4 (July 2002), 58–65.

[12] Chang Hwan Peter Kim, Don Batory, and Sarfraz Khurshid. 2010. Eliminating Products to Test in a Software Product Line. In *IEEE/ACM Int. Conf. on Automated Software Engineering* (Antwerp, Belgium) *(ASE '10)*. ACM, New York, NY, USA, 139–142.

[13] C. H. P. Kim, S. Khurshid, and D. Batory. 2012. Shared Execution for Efficiently Testing Product Lines. In *2012 IEEE 23rd Int. Symposium on Software Reliability Engineering*. 221–230.

[14] Chang Hwan Peter Kim, Darko Marinov, Sarfraz Khurshid, Don Batory, Sabrina Souto, Paulo Barros, and Marcelo D'Amorim. 2013. SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems. In *9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. ACM, New York, NY, USA, 257–267.

[15] D. R. Kuhn, D. R. Wallace, and A. M. Gallo. 2004. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering* 30, 6 (2004), 418–421.

[16] Kun Chen, Wei Zhang, Haiyan Zhao, and Hong Mei. 2005. An approach to constructing feature models based on requirements clustering. In *13th IEEE Int. Conf. on Requirements Engineering (RE'05)*. 31–40.

[17] Harry C. Li, Shriram Krishnamurthi, and Kathi Fisler. 2005. Modular Verification of Open Features Using Three-Valued Model Checking. *Automated Software Engg.* 12, 3 (July 2005), 349–382.

[18] Max Lillack, Christian Kästner, and Eric Bodden. 2014. Tracking Load-time Configuration Options. In *29th ACM/IEEE Int. Conf. on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. ACM, New York, NY, USA, 445–456.

[19] Max Lillack, Christian Kästner, and Eric Bodden. 2017. Tracking Load-Time Configuration Options. (2017).

[20] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. 2010. Evolution of the Linux Kernel Variability Model *(Int. Software Product Lines Conf.)*. Springer-Verlag, Berlin, Heidelberg, 136–150.

[21] S. Maity and A. Nayak. 2005. Improved test generation algorithms for pair-wise testing. In *16th IEEE Int. Symposium on Software Reliability Engineering (ISSRE'05)*. 10 pp.–244.

[22] Jens Meinicke, Chu Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. 2016. On Essential Configuration Complexity: Measuring Interactions in Highly-configurable Systems. In *31st Int. Conf. on Automated Software Engineering* (Singapore, Singapore). 483–494.

[23] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2014. Mining Configuration Constraints: Static Analyses and Empirical Results *(ICSE)*. ACM, New York, NY, USA, 140–151.

[24] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. 2014. Exploring Variability-aware Execution for Testing Plugin-based Web Applications. In *36th Int. Conf. on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. ACM, New York, NY, USA, 907–918.

[25] ThanhVu Nguyen, Ugur Koc, Javran Cheng, Jeffrey S. Foster, and Adam A. Porter. 2016. iGen: Dynamic Interaction Inference for Configurable Software. In *24th ACM SIGSOFT Int. Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. ACM, New York, NY, USA, 655–665.

[26] José A. Parejo, Ana B. Sánchez, Sergio Segura, Antonio Ruiz-Cortés, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2016. Multi-objective test case prioritization in highly configurable systems: A case study. *Journal of Systems and Software* 122 (2016), 287 – 310.

[27] Alexander Von Rhein, Sven Apel, and Franco Raimondi. 2011. Introducing Binary Decision Diagrams in the Explicit-State Verification of Java Code. In *In: Proc. Java Pathfinder Workshop*.

[28] Wolfgang Scholz, Thomas Thüm, Sven Apel, and Christian Lengauer. 2011. Automatic detection of feature interactions using the Java modeling language: an experience report. In *15th Int. Software Product Line Conf., Volume 2*. ACM, 7.

[29] Sven Schuster, Sandro Schulze, and Ina Schaefer. 2013. Structural Feature Interaction Patterns: Case Studies and Guidelines. In *8th Int. Workshop on Variability Modelling of Software-Intensive Systems* (Sophia Antipolis, France) *(VaMoS '14)*. ACM, New York, NY, USA, Article 14, 8 pages.

[30] Larissa Rocha Soares. 2018. Varxplorer: Reasoning about feature interactions. In *Proceedings of the 40th Int. Conf. on Software Engineering*. 500–502.

[31] Larissa Rocha Soares. 2019. *Feature Interactions In Highly Configurable Systems: A Dynamic Analysis Approach With Varxplorer.* Ph.D. Dissertation. Federal University of Bahia (UFBA).

[32] Larissa Rocha Soares, Jens Meinicke, Sarah Nadi, Christian Kästner, and Eduardo Santana de Almeida. 2018. Exploring feature interactions without specifications: A controlled experiment. In *Proceedings of the 17th Int. Conf. on Generative Programming*. 40–52.

[33] Larissa Rocha Soares, Jens Meinicke, Sarah Nadi, Christian Kästner, and Eduardo Santana de Almeida. 2018. Varxplorer: Lightweight process for dynamic analysis of feature interactions. In *Proceedings of the 12th Int. Workshop on Variability Modelling of Software-Intensive Systems*.

[34] Larissa Rocha Soares, Pierre-Yves Schobbens, Ivan do Carmo Machado, and Eduardo Santana de Almeida. 2018. Feature Interaction in Software Product Line Engineering: A Systematic Mapping Study. *Information and Software Technology* (2018).

[35] Sabrina Souto, Marcelo d'Amorim, and Rohit Gheyi. 2017. Balancing Soundness and Efficiency for Practical Testing of Configurable Systems. IEEE Press, 632–642.

[36] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1, Article 6 (June 2014), 45 pages.