Exploring Feature Interactions without Specifications: A Controlled Experiment

Larissa Rocha Soares
Federal University of Bahia, Brazil

Jens Meinicke
University of Magdeburg, Germany

Sarah Nadi
University of Alberta, Canada

Christian Kästner
Carnegie Mellon University, USA

Eduardo Santana de Almeida
Federal University of Bahia, Brazil

Abstract

In highly configurable systems, features may interact unexpectedly and produce faulty behavior. Those faults are not easily identified from the analysis of each feature separately, especially when feature specifications are missing. We propose VarXplorer, a dynamic and iterative approach to detect suspicious interactions. It provides information on how features impact the control and data flow of the program. VarXplorer supports developers with a graph that visualizes this information, mainly showing suppress and require relations between features. To evaluate whether VarXplorer helps improve the performance of identifying suspicious interactions, we perform a controlled study with 24 subjects. We find that with our proposed feature-interaction graphs, participants are able to identify suspicious interactions more than 3 times faster compared to the state-of-the-art tool.

CCS Concepts • **Software and its engineering → Feature interaction**; Reusability; • **General and reference →** *Experimentation*;

Keywords Highly Configurable Systems, Feature Interaction, Controlled Experiment

ACM Reference Format:
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 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '18), November 5–6, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3278122.3278127

1 Introduction

A *feature* describes a unit of functionality of a software system that satisfies a requirement [2]. Highly configurable systems can be composed by selecting features from a set of thousands of features (aka. *configuration options*) [39, 51]. For example, the Linux kernel has more than 15,000 configurable options [30, 42]. This large set of options may be combined in different ways, and developers must guarantee that all valid combinations work properly. The *interaction* among features is a common problem in highly configurable systems, which may result in unexpected behavior that is not easily deduced from the analysis of each feature separately [3, 45]. A system may behave as expected most of the time, but it may present unexpected and problematic interactions only under specific feature combinations.

Determining the influence of feature interactions on a system's behavior is challenging. Anticipating and specifying all likely consequences of each potential feature interaction may not be possible, mainly due to the fact that (i) the number of configurations and feature interactions grows exponentially in relation to the number of features [19]; (ii) the behavior of some interactions may be unknown and unpredictable in advance [3]; and (iii) human effort is required, but people usually do not like writing specifications. To address these challenges, recent analyses focus on detecting feature interaction bugs from *global specifications*. Those are specifications that all configurations of a configurable system need to fulfill, such as requiring the system to not crash [49]. Usually, these approaches check global specifications based on systematic sampling [25, 27, 46], combinatorial interaction testing [18, 31, 38], model checking [5, 15, 17, 28, 50], or variational execution [9, 26, 33, 36, 52].

However, the problem is that features may interact in many ways, for example by triggering events that enable other features, having control over the same variables, and enforcing conditions that suppress other features [8]. Additionally, although some faults lead directly to crashes (e.g., conflicting function names [36]), others may cause more subtle problems. For instance, in the simplified WordPress example of Listing 1, the options *weather* and *smiley* interact in an unintended way, although they do not crash the system. When they are used together in the same system, the temperature is not shown and the system presents an unexpected output: instead of replacing the "[:weather:]" tag with the current temperature (e.g., 70°F), it is rewritten to "[:weather☺".

Since specifications at the feature level are usually missing, the above mentioned approaches may not detect all incorrect system behavior, especially bugs not covered by global specifications and bugs that do not result in a crash or other easily observable behavior. Instead of upfront specifications, we propose to inspect feature interactions as they are detected and incrementally classify them as intended or problematic. We present feature-interaction graphs to facilitate the identification of unintended interactions. A *feature-interaction graph* is a concise visualization that shows all pairwise interactions observed in an execution, presenting the relationships that a pair of features may hold. Hence, we provide an inspection process that helps developers to distinguish intended interactions from interactions that may lead to bugs.

To detect feature interactions in a test execution (without knowing whether they are benign), we use the variational interpreter VarexJ [33, 36]. It performs variational execution to simultaneously execute all system configurations equivalent to aligning the traces of executing all configurations separately. An interaction is represented as a control-flow or state difference in the system that depends on two or more options.

Towards the identification of problematic interactions, we recently proposed VarXplorer [44], an iterative and interactive analysis that inspects feature interactions from the variational execution generated by VarexJ. Figure 1 shows an overview of the approach: given a configurable system and a set of test cases, it detects interactions and provides an incremental analysis of the relationship among features, illustrated through a feature-interaction graph. Relationships provide details on how features interact to support developers in identifying unintended interactions. In addition to feature interaction relationships, the graph shows additional indicators, such as the suppression of one feature by another, that are identified through analyzing control and data flow interactions. The feature-interaction graph is presented to developers for manual inspection where they can use the presented information to determine, and specify, intended versus forbidden interactions. The graph is then refined as more test cases are run, while also taking into account the documented interaction specifications. Unlike global and feature-based specifications, interaction specifications do not specify the behavior of the system or feature. Instead, they help developers focus only on potential bugs by automatically removing benign interactions from the graph.

In general, VarXplorer [44]: (i) determines the relationships between features and presents two classes of interactions, namely *suppress* and *require* interactions; (ii) implements *feature-interaction graphs*, a concise visual representation of feature interactions identified at runtime using variational execution; (iii) proposes a *feature interaction specification* language to allow and forbid interactions on data and control flow; and (iv) presents an *iterative and interactive*

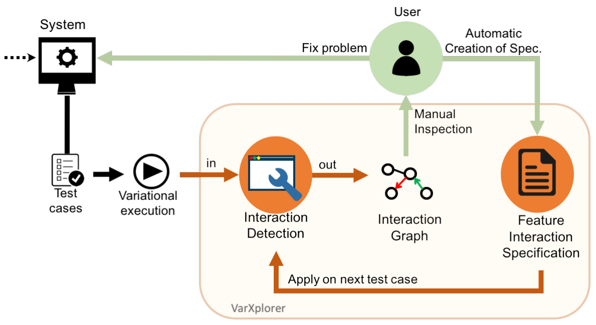


Figure 1. Feature interaction detection with VarXplorer.

approach to refine feature-interaction graphs using feature interaction specifications.

In this paper, we contribute an empirical evaluation of feature-interaction graphs to determine if they do help developers identify problematic interactions. We conduct a controlled experiment with 24 participants from different universities and companies. We measure the effort to identify a buggy interaction based on the information provided by the feature-interaction graph. We also perform an in-depth qualitative analysis based on video and audio recordings, and post-treatment interviews. The results show that participants using VarXplorer outperformed participants using the state-of-the-art tool. They are at least 3 times faster when using VarXplorer to identify suspicious interactions. From the qualitative analysis, we also identify and discuss 5 observations, including how the feature relationships support identifying bugs.

In summary, we make the following contributions:

1. A *user study* showing that feature-interaction graphs improve the efficiency of understanding feature interactions compared to the state of the art.
2. An in-depth qualitative analysis showing advantages of the graph components towards the detection of suspicious interactions.
3. An Eclipse plug-in to generate *feature interaction specifications* and remove interactions that do not represent a bug, allowing the developer to focus only on suspicious cases.

2 How Does a Feature Interact?

A software product can be seen as a configuration of features[1] that need to be composed together without violating their particular requirements. On the one hand, it is relatively simple to specify the behavior of a feature in isolation. On the other hand, specifying and detecting interactions

[1]Henceforth, we use the term feature to refer to any configuration option, module, or component in a configurable system.

Listing 1. Simplified WordPress example [33].

```
boolean STATISTICS, SMILEY, WEATHER, FAHRENHEIT,SECURE_LOGIN;

    void createHtml(String c) {
        c = wpGetContent();
        if (SMILEY)
            c = c.replace(":]", getSmiley(":]"));
        if (WEATHER) {
            String weather = getWeather();
            c = c.replace("[:weather:]", weather);
        }
        if (STATISTICS) {
            int time = getCurrentTime();
            printStatistics(time);
        }
    }

    String getWeather() {
        float temparature = 30;
        if (FAHRENHEIT)
            return (temperature * 1.8 + 32) + "°F";
        return temperature + "°C";
     }
```

among features may not be a straightforward task. The feature interaction problem has been a challenging subject for decades [13].

A *feature interaction* is observed when the combined behavior of two or more features differs from the individual behaviors of both features [16, 22, 53]. In a pair of features that interact, a feature might enable, require, overwrite variables, or even block the effect of another feature.

Next, we present more details on feature interactions, types, and detection strategies available on literature. Besides, we give an overview of our approach and how we detect interaction without specifications, previously presented in [44].

2.1 Feature Interactions

In general, a feature interaction can be classified in terms of its behavior: *intended* interaction and *unexpected* interaction. Intended interaction corresponds to a desired and upfront predicted behavior. However, many interactions cannot be identified in early phases of the software development; they usually result in unexpected behavior. In configurable systems, benign and problematic interactions are different types of unexpected interactions [45]. Although unexpected, a portion of the interactions may pose no risk to the system, features might be combined together to deliver useful and benign functions. Conversely, problematic interactions are harmful and may inject faults to a system behavior.

In Listing 1, we show a code snippet modeled after WordPress. For this example, the current temperature is presented (in either Celsius or Fahrenheit) from the benign interaction between the features `weather` and `fahrenheit`. On the other hand, when the features `smiley` and `weather` interact, the system displays an incorrect output because `smiley` incorrectly replaces part of the *weather* variable. As a result, the system unexpectedly shows an HTML tag instead of the current temperature. Hence, *weather* loses its effect, which is to show the temperature.

Classifying interactions as benign or problematic is not a straightforward task. It requires detecting the unexpected interaction, which often only appears in specific test cases of one or a small set of product variants.[2] One strategy to detect interactions is to provide specifications for all variants. However, this does not scale due to the large number of possible system configurations [39, 51]. Alternatively, specifying each single feature may require less effort. A *feature-based specification* describes the behavior of a feature in isolation without any explicit reference to other features [49].

In a different strategy, developers write specifications that must hold for all configurations. Global specifications represent a common way to reduce the effort of creating specifications for individual features. A typical example corresponds to a requirement that fulfills certain functional requirements in all configurations. However, global and feature-based specifications cannot describe interactions, or distinguish intended from unintended interactions. Besides that, specifications are usually rare in practice.

In our work, we approach the challenge of identifying feature interactions without upfront existing specifications. Although there are many studies to detect and resolve faults caused by feature interaction problems [5, 9, 15, 17, 18, 25–28, 31, 33, 36, 38, 46, 50, 52], identifying unexpected feature interactions that do not lead to a crash, but that cause faulty behavior (such as, the problematic interaction between the features `smiley` and *weather* in Listing 1), remains an open challenge.

2.2 On Detecting Feature Interactions without Specifications

Feature interactions can be detected by comparing the executions of *all* system configurations. Variational execution is an efficient method to compare executions [9, 26, 33, 36, 52]. It is able to share redundancies among executions, which significantly reduces the total number of executions [37]. In this paper, we use the variational interpreter VarexJ [33]: it is a dynamic analysis for Java that tracks interactions on data and control flow during execution. VarexJ provides all interactions on data and control flow for a single test case in all configurations. It also identifies the presence conditions that add or change any functionality during the execution. *Presence conditions* are propositional expressions over options that determine when a specific code artifact is executed [35].

In Figure 1, we present an overview of our process to incrementally analyze feature interactions, proposed previously [44]. Given a configurable system, we execute test cases (system inputs) looking for feature interactions. The developer then explores which interactions are problematic. We support them in the process with a *feature interaction graph*, a concise representation of all (pairwise) interactions

[2]A variant can be referred as a configuration, i.e., a system composed of different feature combinations [22].

among features. Based on the variational execution of a system, the graph provides a visualization of which features interact, and presents their relationships and data context.

Indicating which features interact (raw interactions) does not provide sufficient insights for the developer to identify whether a certain interaction is benign or represents a bug. To understand the relationship between features, we investigate the relation of a feature to the others, such as suppressing or requiring. In addition, interaction relationships may be associated with the data context of the interaction (e.g., the variables involved in the relation). The different values that a given variable may assume can be a signal that something wrong occurred. Highlighting the variables involved might help developers to identify problematic interactions.

Our interaction detection process is incremental in the sense that, based on user inspection, the graph is automatically refined by removing interactions previously indicated as benign by the developers. This refinement is supported through a *feature interaction specification language* and ensures: (i) that the user does not see benign interactions again in future iterations (i.e., for other test cases); and (ii) that any newly detected unintended interactions are flagged in the future. The goal is to incrementally remove intended interactions to focus on unintended interactions. To make specifications easy to create, developers can mark interactions as either allowed or forbidden in the graph [44]. For completeness, we next present our interaction detection, more details can be found in our recent workshop paper [44].

2.2.1 Interaction Detection

In the interaction detection process, we identify and analyze all pairs of features that interact in a system. The input of the detection is the information gathered after executing a test case (presence conditions and variables), and the output is the interaction graph presenting all the interactions.

The creation process of the interaction graph has two major steps: *pairwise detection* and *relationship analysis*. First, we identify the pairs of features that interact and create a basic interaction graph. Then, we perform the relationship analysis and refine the basic graph with additional information about the relationship between features, including the underlying variables they affect, to produce the complete interaction graph.

Pairwise Detection. For pairwise detection, we collect a set $\mathbb{PC}$ with all the presence conditions that occur in the data and control flow of the program. Control flow conditions are path conditions (possible ways an execution can go), and data flow conditions are formed by the conditions on each variable. From $\mathbb{PC}$, we identify all pairs of features that interact by finding features that occur together in the same condition. Given a pair of features (*f1*, *f2*), we assume that there is an interaction between *f1* and *f2* if there is at least one presence condition $p \in \mathbb{PC}$ in which *f1* and *f2* occur simultaneously

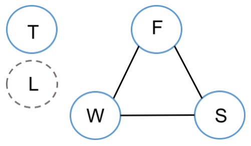


Figure 2. Basic feature-interaction graph for WordPress.

as literals in p:

$$f \blacktriangleright p \ := \ f \text{ occurs as literal in } p \tag{1}$$

$$\mathbb{I} = \{(f_1, f_2) \mid p \in \mathbb{PC} \wedge (f_1 \blacktriangleright p) \wedge (f_2 \blacktriangleright p)\} \tag{2}$$

From Equation 1 and 2, we are able to collect all pairwise interactions. We use them to create the *basic feature-interaction graph*, a simple visualization of all interactions. For example, our running example has five features: `SMILEY` *(S)*, `STATISTICS` *(T)*, `WEATHER` *(W)*, `FAHRENHEIT` *(F)*, and `SECURE LOGIN` *(L)*. Based on the above equations, we identified three pairs of interactions $\mathbb{I}_{wp} = \{(F, W), (S, F), (S, W)\}$ in the entire set of presence conditions $\mathbb{PC}_{wp}$, as follows[3]:

$$\mathbb{PC}_{wp} = \{S, \neg S, W, \neg W, T, \neg T, W \wedge F, W \wedge \neg F,$$
$$W \wedge F \wedge \neg S, W \wedge \neg F \wedge \neg S, \neg S \wedge \neg W\}$$

Figure 2 shows the basic graph for our running example, illustrating the interactions in $\mathbb{I}_{wp}$. Although the program of Listing 1 contains five features, only three of them interact with each other: *F, W,* and *S*. The other two are non-interacting features; they either do not interact with any other feature during system execution or are not executed in any configuration related to the current test case. Although the basic graph shows which features interact with each other, it does not provide enough insight on *how* features interact. We further investigate pairs of features to determine relationships that further describe the interaction. To support users in identifying problematic interactions, we also analyze the variables involved.

Relationship Analysis. We provide two complementary analyses to inspect each pair to determine the effect of a feature on the other: $\mathbb{PC}$*-based analysis* and *data-based analysis*. In the former, we explore presence conditions on control and data flow to identify which relation a feature may have over the other (i.e., either suppress or require other features). The latter is responsible for investigating variables that are controlled by more than one feature. Thus, we identify feature relationships exclusive to variables. For example, a feature *f1* may not present an overall suppression on the feature *f2*, but *f1* may suppress *f2* in relation to a given variable.

A *feature effect* specifies under which conditions a given feature has an effect on the execution. If a feature *f1* has no effect, then the selection of *f1* never adds or changes any functionality that was not present before [35]. In the basic graph of Figure 2, the dashed feature L is not active

[3] $\mathbb{I}_{wp}$: interactions for the WordPress (*wp*) example. $\mathbb{PC}_{wp}$: set of all presence conditions for the *wp*.

and, therefore, L has no effect in the WordPress execution. Inactive features never have an effect. For the other features, we investigate each presence condition to detect the effect one feature may have on the other.

The feature effect is given by analyzing the effect of a given feature on the set of presence conditions. Formally, we say the effect of f on a condition p is given as the function $\mathbb{U}(f,p)$, as follows[4]:

$$\mathbb{U}(f,p) = (f \leftarrow True) \oplus (f \leftarrow False) \tag{3}$$

A feature f has no effect on p if enabling (f as $True$) or disabling (f as $False$), does not affect the value of p, then f does not have an effect on selecting the corresponding code fragment under the condition p. Otherwise, a feature f has an effect on p when enabling and disabling the feature in p leads to different result at least for one configuration, which means that different code fragments are executed. Based on the feature effect, we identified two types of relationships, *suppresses* and *requires*, as follows:

Definition 1. *Let f_1 and f_2 be the two features of an interaction pair. We say that f_1* suppresses *f_2 when the suppressed feature f_2 has no effect if the feature f_1 is selected.*

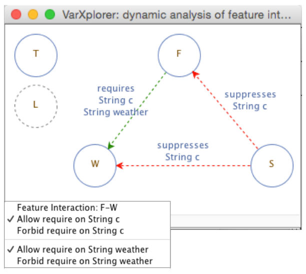
Definition 2. *A feature f_1* requires *feature f_2 when f_1 has an effect only if the feature f_2 is selected.*

Formally, we say f_1 *suppresses* f_2 with presence conditions $\mathbb{PC}$ iff the result of the implication $f_2 \implies \neg\mathbb{U}(f_1, \mathbb{PC})$ is a tautology. Otherwise, we say f_1 *requires* f_2 iff the result of $\neg f_1 \implies \neg\mathbb{U}(f_2, \mathbb{PC})$ is a tautology. For example, the effect of the feature `FAHRENHEIT` (F) on the WordPress execution results in $\mathbb{U}(F, \mathbb{PC}_{wp}) = W$, that is, feature F only has an effect iff W is selected. Thus, F *requires* W in order to have an effect on the system (i.e., $\neg W \implies \neg\mathbb{U}(F, \mathbb{PC}_{wp})$ is a tautology). This behavior can be observed in Listing 1: Line 20 is only executed when the decision in Line 7 is true, which calls the method *getWeather()* in Line 8. Then, we see that F is a sub-feature of W. From the domain knowledge, we know that this is an intended cooperation in terms of a *require* relationship between those two features.

In contrast, if F would only have an effect iff $\neg W$, then W would *suppress* F (i.e., F would be blocked by W, which would be a bug). We perform the same analysis for each pair of interaction to determine the effects of features in a pair. This analysis identifies all cases of suppress and require relationships between features, which may support the user to find faulty behaviors.

To further explore additional relationships between features, we complement the flow analysis with a data analysis. Features that do not directly interact on the system flow may still interact by controlling the same variables. *Conditional variables* are those in which the values depend on more than one feature. Unexpected data values may reveal bugs from unintended interactions on variables. This analysis supports developers in a low-level inspection. For example, from the relationship analysis, we found that S *suppresses* F in data (variable c). Thus, when both F and S are selected, the variable c is not overwritten by F. This case may be an example of a bug because wrong information is displayed to the user. Instead of seeing the current temperature, users see the tag "[:weather☺".

[4]Feature effect is also known as unique existential quantification [35].



Feature Interaction: F–W
✓ Allow require on String c
Forbid require on String c
✓ Allow require on String weather
Forbid require on String weather

Figure 3. VarXPlorer screenshot of the Wordpress graph.

For completeness, our detection strategy should be complemented with a larger iterative process of detecting and documenting evaluated interactions. We described such a process as well as described a corresponding specification language in prior work [43, 44]. In a nutshell, the *feature interaction specification language* helps developers to either allow or forbid interactions in a configurable system [44]. For example, guided by the visualization provided by Figure 3, the user can automatically allow the benign data interaction between F and W for the variables `c` and `weather`. Thus, the intended interaction will not be shown again in the analysis of future test cases. Conversely, for the other two interactions of the WordPress example (S-W and S-F), one of the features in each interaction is being suppressed by the other. In case of bugs, the user may want to fix the problem directly in the code and also mark those interactions as suspicious in the graph, by means of the *forbid* specification, as Figure 3 shows.

3 Experimental Evaluation

We performed a controlled experiment to understand how feature interaction graphs help users identify suspicious interactions. We investigate and compare the ability of users to identify problematic interactions with and without VarXplorer, in a setting with different tasks and systems. We are interested in the time spent to detect suspicious interactions and in the perception of the participants about the tool.

3.1 Research Questions

We investigate the usefulness of interaction graphs as a strategy to identify feature interaction bugs in programs. VarXplorer is an Eclipse plug-in that abstracts from multiple executions to show feature relationships. To the best of our knowledge, there is currently no comparable tool that is able to detect suspicious interactions based on a dynamic analysis of relationships between features, without specifications.

One possible baseline could be to compare VarXplorer with a traditional source-code inspection or the standard Eclipse debugger. However, we assume that inspecting the code to detect interactions is hard, slow, and possibly a tedious work. On the other hand, the standard debugger is a general-purpose tool that is not specifically designed with feature interactions in mind. Recent work already shows that Varviz, an Eclipse plug-in that provides a variability aware execution trace of the code, outperforms the standard debugger for comprehension tasks involving interactions [32]. Based on such previous findings, we compare VarXplorer to the current state-of-the-art tool in this area, Varviz.

We aim to answer the following main question: *Does VarXplorer help developers to identify suspicious feature interactions?*, which we split into two concrete research questions:

- RQ1: Does VarXplorer improve the performance of identifying suspicious interactions compared to Varviz?
- RQ2: How does the interaction graph presented by VarXplorer help understand the suspicious interactions in a program?

RQ1 is related to the effort required to identify suspicious interactions. We measured the time spent to detect interactions in two setups: using VarXplorer and Varviz. For each tool, we created two tasks for two different systems. We measured how long participants take to identify and understand a suspicious/buggy interaction from the information provided by (i) the graph generated with VarXplorer; versus (ii) the execution trace created by Varviz.

To answer RQ2, we analyze what information helps participants understand and identify suspicious interactions. In addition, we want to know how the graph components (relationships, variables arrows, and colors) can help developers with the detection of buggy interactions.

3.2 Experiment Overview

We designed our experiment as a within-subjects study. For this design, the same group of participants receives more than one treatment [23]. In this way, all participants perform tasks using both tools, VarXplorer and Varviz. The tools are the treatments of our experiment.

Within-subjects designs have greater statistical power than between-subjects designs: we need fewer participants in the study to find statistically significant effects, because each participant is tested under all treatments. Within-subjects designs also represent a good strategy when it is difficult to recruit participants [20].

The experiment consists of two tasks: participants first start with one tool and they have to identify interactions in a given system. After finishing the first task, they start the activities with the second tool and another system. For each tool, they use a different task to reduce learning effects.

While the participants are working on the tasks, we ask them to verbalize their thoughts and tell us what they are doing (think-aloud protocol [11]). When necessary, we also ask them why they are doing a particular activity. The think-aloud protocol makes the process as explicit as possible during the tasks, because it captures preference and performance data simultaneously, rather than waiting until the experiments finishes to ask all the questions. In addition, we record the screen and audio to collect supporting data for analyzing the time and strategy used by participants to find interactions. We run the experiment for one participant at a time.

We complement the above setup with a pre-survey and a post-interview. Before the experiment, we ask them to answer an online pre-survey, which we used to collect background data about their experience, mainly with Java and the Eclipse IDE.[5] We create balanced groups of participants based on their experience. For the post-interview, we asked two questions: (1) which tool is easier to understand a feature interaction? and (2) what makes this tool easier in comparison to the other one? We triangulate the gathered answers with the data we obtain from the think-aloud protocol.

3.3 Pilot study

Before the main experiment, we conducted two pilot studies with 8 graduate students from two universities in different countries. We used the pilot study results to determine the amount of time needed to execute our tasks. This allowed us to estimate and plan the number of participants we needed in the main study. We found a large effect size between the participants who used VarXplorer (3 min on average) versus the ones who used Varviz (13 min on average), which suggests that we do not need a large group of participants. The pilot study also allowed us to assess whether the participants could properly understand the subject systems and the tasks they should perform, as well as to train the researcher who overlooked the experiment. We do not consider the results of the pilot in our analysis.

3.4 Participants

After the pilot, we recruited 24 participants (excluding pilots). To recruit them, we sent emails to professors in two universities, from different computer fields, to suggest ex-students (developers) and current students.

We received 24 emails from three different profiles: undergrad students, graduate students, and professional developers. Furthermore, some of the students are also developers. The participants experience regarding Java and Eclipse IDE varied from few months to more than 10 years.

Table 1 shows the participants involved in the experiment: 7 undergrad students (Un), 9 master students (M), 4 PhD. students (PhD), and 7 developers (Dev). The students are from two different universities (U1 and U2) and the developers

[5]It is a Google Form available at: https://goo.gl/forms/F4a6e7K0agpLp0Hv2

Table 1. Participants

Partic.	Institution	Position	Group	Exp. (years)
1	U1	M	1	>= 5 and <10
2	U1	PhD	1	>= 10
3	U2	Un & dev	1	>= 1 and <5
4	U1	M	1	<1
5	U1	PhD	1	>= 1 and <5
6	U2	Un	1	>= 1 and <5
7	U1	PhD	2	>= 10
8	U1	M & dev	2	>= 5 and <10
9	U1	Un	2	>= 1 and <5
10	C1	Dev	2	>= 1 and <5
11	U1	M	2	>= 1 and <5
12	U1	M	2	>= 1 and <5
13	U1	M	3	>= 5 and <10
14	C2	Dev	3	>= 10
15	U1	M	3	>= 5 and <10
16	C3	Dev	3	>= 1 and <5
17	U1	Un	3	>= 1 and <5
18	U1	Un	3	>= 1 and <5
19	U1	M	4	>= 5 and <10
20	U1	M	4	>= 1 and <5
21	U2	Un & dev	4	>= 1 and <5
22	C4	Dev	4	>= 5 and <10
23	U1	Un	4	>= 1 and <5
24	U1	PhD	4	>= 10

work in four different companies (C1, C2, C3, and C4). According to our design, we created four groups with a similar background distribution of participants.

3.5 Experimental Material and Tasks

We used two product lines as the evaluation material: Elevator and Telephone.

The elevator system has been proposed by Plath and Ryan [40]. It is an extensible elevator model whose features are designed to highly interact. For example, the elevator needs to stop if it is empty or priority service for a special floor is activated. Although this system has only 1046 LOC and 6 features, it is hard to understand the impact of its features due to the interactions. Furthermore, it has been frequently used in the literature [4, 6, 10]. We used the Elevator Java version from the SPL2go repository.[6]

The telephone system has been widely discussed in the literature due to the Feature Interaction Detection Contest that was held in 1998 and 2000 [21]. The contest aimed to compare various methods and tools for detecting feature interactions. To enable a comparison, the objective was to detect interactions among a given set of features for a given telephone system. The telephone system was designed to present many interactions. Based on the specification from the contest, we created a Java implementation for the telephone system. We implemented 6 features and 1005 LOC.

We design two tasks, one for each system. The tasks were designed to be similar in size, number of features, and time to be executed. The pilots served to align them. In general, we asked the participants to use the tool given to them (either VarXplorer or Varviz) to identify suspicious interactions on the systems for a given test case. The tasks were designed to

[6]http://spl2go.cs.ovgu.de

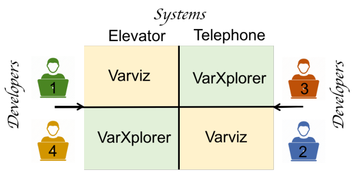


Figure 4. Latin Square to our treatments.

present just one suspicious interaction for each system and a couple of benign interactions. We provide the participants with the description of each feature in the target system, test case scenario documentation, and the system's source code.[7] From those artifacts, they get the domain knowledge about the systems. Thus, the participant role in the experiment is to identify the problematic interaction in each system. We next describe the details of the two tasks.

Task 1. According to the features specification of the elevator system, when the elevator has two thirds of the maximum weight, it should not attend to calls until it delivers passengers, making the weight be less than two thirds. However, because of a problematic interaction between two features (*Executive Floor* and *Two Thirds Full*), the elevator goes to pick a passenger up even though it has already achieved two thirds of the capacity, which forces the elevator to not close the door until someone leaves it. In this situation, the feature *Executive Floor* is blocking the execution of the feature *Two Thirds Full*. In this task, the participants should figure out that this interaction leads the system to a wrong behavior. They have to identify the suspicious interaction using either Varviz or VarXplorer, depending on the group they were assigned. We request them to identify the problem, but we do not require them to fix the problem in the source code.

Task 2. The contest instructions describe all the feature specifications [16], such as: (i) *Call Forward on Busy*, all calls to the subscribing line are redirected to a predetermined number when the line is busy; and (ii) *Call Waiting*, allows the subscriber to be noticed that another party (incoming call) is trying to reach his number while her line is busy. However, when the line is already busy and another number is trying to reach that line, because of a precedence in the implementation, the telephone system is always forwarding the call, even if the person wants to put the incoming call on waiting. Again, one feature has its behavior suppressed by another and the participants should identify that this is the suspicious interaction, by using one of the two tools.

3.6 Design

The tool used during the experiment represents an independent variable with two levels, VarXplorer and Varviz. We also distinguish participants related to the systems they use: Elevator system and Telephone system. Furthermore, to analyze the performance of the tools (RQ1), we measured the time

[7]The documentation and tasks description is available at https://goo.gl/57XwtQ

spent by participants to find the suspicious interaction in each system. Time is a dependent variable of our evaluation.

Latin Square Design. Since participants perform two tasks, one followed by the other, there can be problems of carryover effects. Thus, each measurement may depend not only upon the treatment given but also on the preceding treatment [20]. To avoid those kind of effects, we use a Latin Square design [14]. It represents a method of placing treatments so that they appear in a balanced fashion within a square block. Latin Square is an useful design where the experimenter desires to control variation in two different directions. In this way, treatments should appear once in each row and column.

In addition to the standard Latin Square, we use three strategies to avoid learning effects: (i) we have every treatment preceding every other treatment the same number of times (counterbalanced Latin Squares); (ii) we change the order participants use the tools; and (iii) participants do not repeat the same tool or the same system in different tasks.

Figure 4 was inspired by the Latin Square to show the distribution of the population to our experiment. The columns are labelled with the two subject systems (Elevator and Telephone). The rows correspond to the developers. The 4 squares (cells) contain the two treatments (Varviz and VarXplorer). Then, we allocated one group of participants to each cell. Based on this design, each participant received the two treatments listed in a given row for the two subject system listed in the corresponding columns.

Clearly, we can only have a participant looking for interactions in a given program once, otherwise there would be a learning effect on subsequent attempts. Following the strategies of our design, participants are using different tools and systems for each task and they have never used neither the tools nor the systems before the experiment. Moreover, since we permute the order in which they perform the activities, we create 4 groups, as Figure 4 shows. We balanced the groups based on the participants experience. The order of each group is described as follows:

- Group 1: first Varviz-Elevator, then VarXplorer-Telephone
- Group 2: first Varviz-Telephone, then VarXplorer-Elevator
- Group 3: first VarXplorer-Telephone, then Varviz-Elevator
- Group 4: first VarXplorer-Elevator, then Varviz-Telephone

3.7 Procedure and Execution

Before the participants receive their tasks, we first introduced the experiment with a *tutorial* about feature interactions. The tutorial took 10 minutes on average. Then, each participant had two tasks to accomplish, with descriptions and instructions provided for each task.

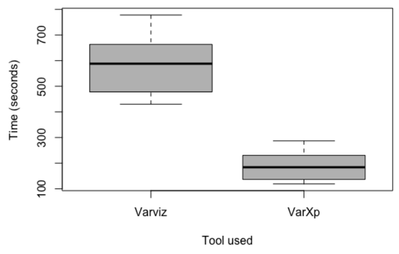


Figure 5. Time results for the tools

Before each task, we conducted a *warm-up section* to introduce the tool (either Varviz or VarXplorer) using a third system, the mock WordPress shown on Listing 1). For the first warm-up, we give them Eclipse with the first tool (depending on the participant group), the mock WordPress source code, and a list of features. At this point, they have to identify the suspicious interaction in WordPress using the tool given to them. During this warm-up, we answer their questions about the tool. After that, we give them the *first experiment task* corresponding to that tool, which includes again the Eclipse with the tool, the experiment system source code (either Elevator or Telephone) and the list of features. Those steps correspond to the first part of the experiment.

The second part starts when the participants finish the first task. Hence, we perform a second warm-up with the second tool and WordPress, where they again have to identify a suspicious interaction. After they finish the warm-up with the second tool, they start to perform the *second experiment task*, which consists of identifying the suspicious interaction in the second tool and the second system (either Elevator or Telephone). The experiment finishes when they have performed both tasks. As final step, we conduct the post-experiment interview. Each participant took on average 1 hour to complete the experiment. Thus, we approximately had 24 hours of audio and screen recording to analyze.

To provide the same environment to each participant and avoid having to install and configure Java and Eclipse parameters, we used the same laptop for all participants in the experiment. Each group received two versions of Eclipse, one version with the Varviz installed and the other one with VarXplorer, with two programs in the workspace, the warm-up program (WordPress) and the experiment program (either Elevator or Telephone). Thus, each Eclipse installation corresponds to a cell of our Latin Square design, and we could make sure they would use the right tool in the right system.

3.8 Data Analysis

For the statistic analysis of our data, we conducted an analysis of variance using a between-subjects ANOVA. It is a parametric test for determining whether significant differences occur in an experiment containing two or more conditions. ANOVA has three assumptions: the dependent variable measures normally distributed interval, the population has homogeneous variance, and each cell (Latin Square cell, in

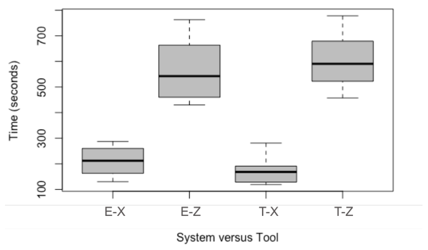


Figure 6. Time results grouping tools and systems. E: Elevator; X: VarXplorer; Z: Varviz; T: Telephone.

our case) contains an independent sample [23]. We used the Shapiro-Wilk normality test, the Bartlett test of homogeneity of variances, and the Tukey HSD test to test the multiple comparisons of means. We conventionally reject our hypothesis when *p-value* < 0.05.

For the qualitative analysis, we watched the videos and listened to the audios (including experiment and post-interview) looking for commonalities and differences in the way participants execute the tasks and their perception about the tools. We transcribed participants answers and informally generated codes to passages of the data which are relevant to understand participants difficulties and meaningful differences about the two tools used.

4 Results and Discussion

This section presents the results of our experiment and discusses the implications. We next present both statistical and qualitative analysis to answer our research questions.

4.1 RQ1: Does VarXplorer Improve Performance of Identifying Suspicious Interactions Compared to Varviz?

Participants using VarXplorer outperformed participants using Varviz with respect to the average task time.[8] On average, participants accomplished their tasks 3.06 times faster using VarXplorer. The participants that used VarXplorer took an average of 3 min to perform the task, while the others that used Varviz had an average of 9 minutes. All the participants were able to identify the suspicious interaction in both tasks, which is why we compare time and not also correctness. Figure 5 graphically shows the time results.

Statistic Analysis of Performance. We used ANOVA to statistically evaluate the tools. The difference between the average times to perform the study tasks with each tool proved to be statistically significant. Based on the ANOVA test, we rejected the null hypothesis (p-value $<$ 2e-16) that the distribution of the population is homogeneous. Thus, VarXplorer reduces developer effort to identify suspicious interactions in both tasks, elevator and telephone system.

Figure 6 shows the results for our 4 groups. For both systems, there is a significant effect size between Varviz and VarXplorer tasks. The subject systems had similar performance times, and regardless of system, VarXplorer was faster.

Test of assumptions. The ANOVA test requires two assumptions of the underlying data: normal distribution and homogeneous data. Our statistical tests show that our data is normally distributed (p-value = 0.4447), but is not homogenous (p-value = 0.0015). However, the heterogeneity of the data does not affect the results of ANOVA, since the groups have the same size [23], 24 measures each.[9] Figures 5 and 6 also show that the data presents a large effect size, such that violating the assumption is unlikely going to change the decision of rejecting the null hypothesis.

Analysis of Order Influence. Even though we designed our experiment to avoid learning effects and tool/system order influence, we still performed the ANOVA test on the groups to check whether the order presented an influence.

For the systems group, the data from the order of the systems are not different, i.e., the order of the systems does not statistically influence the results (p-value = 0.803). For the tools groups, it presents a large effect size between the groups that used VarXplorer against the Varviz groups. According to the ANOVA test, we get statistically significant evidence that our groups have different averages (p-value $<$ 2e-16). Thus, the order of the systems does not matter to the evaluation.

In order to analyze the interactions in the tools order group, we performed a Tukey HSD test [23]. We saw a small learning effect when Varviz is used after Varxplorer (p-value = 0.0378). This situation occurs because the participants learn from VarXplorer graphs: they learn about relationships between features and start to explicitly look for them in the Varviz trace. Although the systems presented a small difference, this situation does not significantly affect the analysis of variance: this effect is tiny compared to the overall effect size. The fastest Varviz time is still significantly slower than the slowest VarXplorer time.

RQ1: The results confirmed that participants using VarXplorer identify feature interactions at least 3 times faster compared to participants using Varviz.

4.2 RQ2: How does the Interaction Graph Presented by VarXplorer Help Understand the Suspicious Interactions in a Program?

We analyzed the videos (audio and the screen recordings) of all participants to know how feature-interaction graphs help understand feature interactions. We watched the videos to find common activities that the participants performed during the tasks, besides comparing the findings with the interviews answers. Thus, we could compare the participants

[8]The time measured for the participants is available at https://goo.gl/JhtDEj and the R script is at: https://goo.gl/rmWv7L

[9]To confirm our analysis, we also performed non-parametric tests with Kruskal and Wilcox. For both tests, we rejected the null hypothesis, affirming that our results are robust.

perception with the activities performed in the tasks.

Observation 1: *The explicit type of relationship for a pair of features guides the analysis and decreases the analysis time.*

VarXplorer represents an alternative to detect interactions with no need to debug the code. The *require* and *suppress* relationships graphically represented as colored arrows in the graph caught the attention of the participants to what is happening with a given pair of features. In a debugging tool, such as Varviz, subjects need to follow the execution flow step by step to interpret what is going on in the system based on methods and variables calls, for example. During the survey performed after the experiment, P20 (see Table 1) affirmed: *"VarXplorer is simpler and easier because it shows, in addition to the interactions, the relationships"*. Along the same lines, P13 said: *"The colors of the arrows in VarXplorer serve as an alert to me to investigate whether the interactions are correct or not"*.

Participants also talked about partial relationships. For example, when a feature suppresses the other of changing a specific variable. Partial relationships affect just one or some variables (also called *conditional variables*, but not all the behavior of a given feature. A program usually has many variables, which may assume many values to different interactions. Looking for conditional variables is a hard task. VarXplorer makes this process faster to show the influence of a relationship on variables, interaction-dependent variables. For instance, P14 stated: *"I don't need to look for the variables which may be problematic, the VarXplorer graph already brings this information to me."*

Observation 2: *To use VarXplorer, you might not need to know details of the implementation, or even the programming language used.*

The VarXplorer graph only presents interaction information, without showing other unrelated details, such as control flow paths, methods and classes names, non-related variables, and variables values. Any person that has knowledge about the system requirements and the feature specifications may be able to judge whether the features behave as expected, based on the relationships presented in the graph. For example, P21 said: *"VarXplorer is more objective in showing the interactions. I do not need to worry about low level of the system, such as methods, classes, components, and all the possible ways the program can go to realize that an interaction is suspicious."*

We observed that participants become convinced an interaction is suspicious based on the perception they have of the features description of the system. Figure 7 shows that just two of the participants looked at the source code. They looked at it during the Varviz tasks to see how some features were implemented. In this way, VarXplorer can be used by different profiles in a development team (e.g., engineers, managers, testers, and developers), even those that do not know details about the system. P8, who is a developer affirmed: *"I even do not need to know the programming language or how the code is implemented, any person from our team can use the graph to understand what is going on in the system"*.

Observation 3: *VarXplorer also shows non-interacting features and no-effect features, which might be indicatives of bugs.*

Besides interactions, VarXplorer also shows features that do not interact with any other feature (non-interacting features). In cases where developers know that a given feature should interact, this information can alert them that something may have happened, leading the feature to not interact with anyone else.

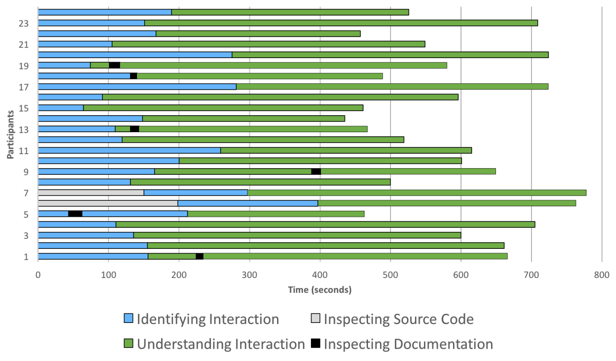
The same occurs with features that are not called in the execution (no-effect features): if we run a test case that a given feature should be executed, and it is not, it is a case to be investigated. This information about non-interacting features and no-effect features is not explicit in Varviz, while VarXplorer graphically represents them. Non-interacting features are shown in the trace because they have effect and are part of the execution, but no-effect features are not represented. This information was perceived as useful by some participants, as P16 stated: *"VarXplorer shows me features that have no effect or do not interact, while this information is hidden in Varviz"*.

Observation 4: *Understanding interactions in Varviz takes longer than finding out where they are located in the trace.*

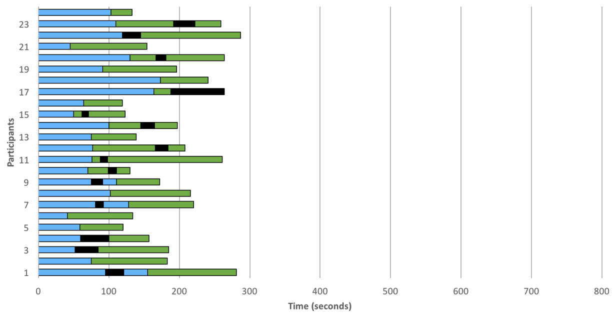
Varviz tests all the combinations of features and captures control and data flow of a given test case. Since the trace shows more information than the graph, we suspected that the time to *identifying* a suspicious interaction (find where in the trace the features are interacting together) would be longer than to properly *understand* the interaction (tell us if the interaction is either OK or suspicious).

Furthermore, participants complained about the time spent to find an interaction in the Varviz trace. For example, P5 said: *"I see how the trace works, but it is not clear to me where I should take a look at to find interactions"*. Another one, P7, also stated: *"Varviz presents everything together, a mix of information. It shows more things than I need to understand the interaction."*

However, we observed that most of the time was spent on properly understanding the interactions, which in practice consists in comprehending Boolean logic expressions related to the suspicious interactions present in the trace. Figure 7 shows the execution of the tasks for the 24 participants. For most of them, the time related to *understand* an interaction with Varviz took twice as long than *identifying* it in the trace. Still, we believe we cannot generalize this particular result



(a) Varviz Tasks



(b) VarXplorer Tasks

Figure 7. Time spent on performing the tasks to Varviz and VarXplorer.

to other interactions or even other systems, because the suspicious interactions of our tasks was placed in the first third part of the trace, coincidentally. Thus, since the participants started to analyze the trace from the beginning, we believe that the time to identify an interaction is more related to the moment it is called in the execution.

Observation 5: *VarXplorer and Varviz complement each other.*

VarXplorer has been discussed as faster and easier to identify and understand an interaction than Varviz. Although Varviz also shows interactions, it has a different purpose. It was designed for understanding faults and program comprehension tasks that involve understanding differences among similar executions [32]. For instance, P3 reported: *"I can use the graph to get an overview on the features that interact, and then I can use the trace to understand the details, see the value of the variables and the flow of execution."*

We observed that Varviz is a valuable strategy to be used after detecting the interactions with VarXplorer. Varviz can be used instead of a standard debugger to look for the cause of an interaction bug. Both tools are Eclipse plug-ins and we believe they may can complement each other in practice.

RQ2: The results confirm that the relationships graphically represented as arrows and colors in VarXplorer make the developer work easier and faster. Also, VarXplorer only shows conditional variables, which reduces the amount of information shown to developers.

4.3 Threats to Validity

We applied our approach to small programs due to the boundaries of an in-lab study; our results may not generalize to larger programs in the wild. However, given that our approach was clearly helpful even in small programs, we argue that is likely helpful for larger systems as it is nearly impossible to detect behavioral interactions without specifications or without specialized tool support [34].

We did not compare our tool with a standard debugger as baseline, as we believe that the task without specialized tool support (e.g., Varviz) would be too difficult and slow for an in-lab study. Thus, a direct comparison with Varviz, which is specialized to graphically show the execution, is more practical than to compare with a standard debugger. Varviz, at least, shows what happens in the execution when features interact (the trace shows all the possible paths), while using the debugger the developer has no clue where to start looking for interactions. Given that VarXplorer was shown to be significantly faster than Varviz with a large effect size, and that Varviz was shown to outperform the standard debuggger [32], we speculate that comparing VarXplorer to the standard debugger would have produced an even larger effect size.

We used 24 participants in our study of which several where students without former experience on interactions (i.e., beginners for this kind of analysis). Experienced programmers for such kind of analysis will perform better for the tasks proposed. However, also experienced programmers will benefit from our tool support as VarXplorer provides them essential information that helps to understand and detect feature interactions. In addition, we used a think-aloud protocol to gain qualitative insights, which may influence the performance of the participants. However, we argue that the influences are similar across the groups and that the differences among the tools are large enough that this influence can be neglected.

5 Related Work

Instead of variability-aware execution, some approaches have performed static analysis to detect interactions [1, 12, 29]. However, despite recent advances, static analysis of systems with high accuracy remains challenging [7, 29]. In contrast, we use a dynamic analysis, variational execution, which is able to analyze large software [33, 36, 52]. Others aim to execute configurations separately, and use symbolic execution to identify interaction problems [17, 24]. Reisner et al. measured the effect of interactions only on control flow using symbolic execution [41], whereas we analyze both control and data flow.

Delta debugging approaches systematically narrow the state difference between a passing run and a failing run [47, 48, 54]. For example, Zeller [54] isolated cause-effect chains for failures. Sumner et al. [47, 48] improved Zeller's work and provided an automatic debugger to precisely align two executions. Conversely, our approach explains differences among many executions. Unlike Delta debuggers, Varviz dynamically tests different executions [32]. However, as far as we know, no work provides explicit information about the relation between features, as we do with suppress and require relationships.

Several approaches work with feature-based specifications to detect interactions. Li et. al [28] present a model checking approach to detect interactions automatically given a group of feature specifications. The approach tests CTL (computation tree logic) properties of features to identify cases in which the specification is violated. Apel et. al [5] also propose a technique to verify whether specifications hold across system configurations. To perform this verification, specifications for intended interactions may be needed, and each feature requires a formal specification of its behavior.

With feature-based specifications, interaction faults can be detected when a feature specification is violated in a configuration. In practice, nevertheless, it is uncommon to create specifications for all features. In general, approaches based on feature specifications present two main drawbacks: (1) from the whole set of features, it is not clear which combinations of features need to be verified and (2) verification tools need precise specifications to check against, information that developers are often reluctant to prepare.

Global specifications only describe properties for all configuration systems, and can thus not describe nuances of intended and unintended interactions to recognize if they affect feature behavior. Generally, it is difficult to find bugs caused by unintended interactions without any specification. Thus, despite their disadvantages, global specifications provide a convenient way of detecting interactions. For that reason, many studies base their approaches on that kind of specifications and focus on exploring the configuration space, such as systematic sampling [25, 27, 46], combinatorial interaction testing [18, 31, 38], model checking [5, 15, 17, 28, 50], and variational execution [9, 26, 33, 36, 52].

6 Concluding Remarks

We propose VarXplorer, a dynamic approach to identify feature interactions without any previous system or feature specification. From a configurable system, it creates feature-interaction graphs as a representation of all pairwise interactions between features, besides showing relationships and the interaction-dependent variables. We conducted a controlled experiment to evaluate how interaction graphs help identify suspicious feature interactions in highly configurable systems. We used two of the most common systems in the literature designed to contain many interactions, Elevator and Telephone. Then, we compared VarXplorer with another tool, Varviz, and we found that VarXplorer is on average 3 times faster than Varviz. In future work, we aim to study how to scale VarXplorer to large systems through slicing, where we can control the software and stop the execution on predetermined stop points.

Acknowledgments

We would like to thank the participants in our experiment and pilot study. We also thank professors Ivan Machado and Gecynalda Gomes for the valuable feedback. This work has been supported in part by the NSF (awards 1318808, 1552944, and 1717022), AFRL, DARPA (FA8750-16-2-0042), and NSERC.

References

[1] F. Angerer, A. Grimmer, H. Prähofer, and P. Grünbacher. Configuration-Aware Change Impact Analysis. In *ASE*, pages 385–395. IEEE, 2015.

[2] S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *JOT*, 8(5):49–84, 2009.

[3] S. Apel, S. Kolesnikov, N. Siegmund, C. Kästner, and B. Garvin. Exploring Feature Interactions in the Wild: The New Feature-interaction Challenge. In *FOSD*, pages 1–8. ACM, 2013.

[4] S. Apel, A. v. Rhein, P. Wendler, A. Größ linger, and D. Beyer. Strategies for product-line verification: Case studies and experiments. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE, pages 482–491. IEEE Press, 2013.

[5] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Detection of Feature Interactions Using Feature-Aware Verification. In *ASE*, pages 372–375. IEEE, 2011.

[6] S. Apel, A. Von Rhein, T. Thüm, and C. Kästner. Feature-interaction detection based on feature-based specifications. *Comput. Netw.*, 57(12):2399–2409, 2013.

[7] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. *PLDI*, 49(6):259–269, 2014.

[8] J. M. Atlee, U. Fahrenberg, and A. Legay. Measuring behaviour interactions between product-line features. In *Formalise*, pages 20–25. IEEE Press, 2015.

[9] T. H. Austin and C. Flanagan. Multiple Facets for Dynamic Information Flow. *SIGPLAN Not.*, 47(1):165–178, 2012.

[10] S. Ben-David, B. Sterin, J. M. Atlee, and S. Beidu. Symbolic model checking of product-line requirements using sat-based methods. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE, pages 189–199. IEEE Press, 2015.

[11] H. Beyer and K. Holtzblatt. *Contextual Design: Defining Customer-Centered Systems*. Elsevier, 1997.

[12] M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury. Regression Tests to Expose Change Interaction Errors. In *ESEC/FSE*, pages 334–344. ACM, 2013.

[13] T. F. Bowen, F. S. Dworack, C. H. Chow, N. Griffeth, G. E. Herman, and Y. J. Lin. The feature interaction problem in telecommunications systems. In *SETSS*, pages 59–62, 1989.

[14] G. E. Box, J. S. Hunter, and W. G. Hunter. *Statistics for experimenters: design, innovation, and discovery*, volume 2. Wiley-Interscience New York, 2005.

[15] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L.-J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *LICS*, pages 428–439. IEEE, 1990.

[16] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115 – 141, 2003.

[17] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic Model Checking of Software Product Lines. In *ICSE*, pages 321–330. ACM, 2011.

[18] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction Testing of Highly-Configurable Systems in the Presence of Constraints. In *ISSTA*, pages 129–139. ACM, 2007.

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

[20] A. Dean. Experimental design: Overview. In N. J. Smelser and P. B. Baltes, editors, *International Encyclopedia of the Social Behavioral Sciences*, pages 5090 – 5096. Pergamon, 2001.

[21] N. Griffeth, R. Blumenthal, J.-C. Gregoire, and T. Ohta. Feature interaction detection contest of the fifth international workshop on feature interactions. *Computer Networks*, 32(4):487 – 510, 2000.

[22] R. J. Hall. Feature combination and interaction detection via foreground/background models. *Computer Networks*, 32(4):449 – 469, 2000.

[23] G. W. Heiman. *Basic statistics for the behavioral sciences.* Cengage Learning, 2013.

[24] M. Hentschel, R. Hähnle, and R. Bubel. The Interactive Verification Debugger: Effective Understanding of Interactive Proof Attempts. In *ASE*, pages 846–851, 2016.

[25] C. H. P. Kim, D. Batory, and S. Khurshid. Eliminating Products to Test in a Software Product Line. In *ASE*, pages 139–142. ACM, 2010.

[26] C. H. P. Kim, S. Khurshid, and D. Batory. Shared Execution for Efficiently Testing Product Lines. In *ISSRE*, pages 221–230. IEEE, 2012.

[27] C. H. P. Kim, D. Marinov, S. Khurshid, D. Batory, S. Souto, P. Barros, and M. d'Amorim. SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems. In *ESEC/FSE*, pages 257–267. ACM, 2013.

[28] H. Li, S. Krishnamurthi, and K. Fisler. Modular Verification of Open Features Using Three-Valued Model Checking. *ASE*, 12(3):349–382, 2005.

[29] M. Lillack, C. Kästner, and E. Bodden. Tracking Load-Time Configuration Options. *TSE*, 2017.

[30] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wąsowski. Evolution of the linux kernel variability model. SPLC, pages 136–150. Springer-Verlag, 2010.

[31] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *ICSE*, pages 664–675. ACM, 2016.

[32] J. Meinicke, C.-P. Wong, C. Kästner, and G. Saake. Understanding Differences among Executions with Variational Traces. *ArXiv e-prints*, 2018.

[33] J. Meinicke, C. P. Wong, C. Kästner, T. Thüm, and G. Saake. On Essential Configuration Complexity : Measuring Interactions in Highly-Configurable Systems. In *ASE*, number 2, pages 483–494, 2016.

[34] J. Melo, C. Brabrand, and A. Wąsowski. How Does the Degree of Variability Affect Bug Finding? In *ICSE*, pages 679–690. ACM, 2016.

[35] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. Mining configuration constraints: Static analyses and empirical results. ICSE, pages 140–151. ACM, 2014.

[36] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications. In *ICSE*, pages 907–918. ACM, 2014.

[37] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Exploring variability-aware execution for testing plugin-based web applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE, pages 907–918. ACM, 2014.

[38] C. Nie and H. Leung. A Survey of Combinatorial Testing. *CSUR*, 43(2):11:1–11:29, 2011.

[39] J. A. Parejo, A. B. Sánchez, S. Segura, A. Ruiz-Cortés, R. E. Lopez-Herrejon, and A. Egyed. Multi-objective test case prioritization in highly configurable systems: A case study. *Journal of Systems and Software*, 122:287 – 310, 2016.

[40] M. Plath and M. Ryan. Feature Integration Using a Feature Construct. *SCP*, 41(1):53–84, 2001.

[41] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter. Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems. In *ICSE*, pages 445–454. ACM, 2010.

[42] V. Rothberg, N. Dintzner, A. Ziegler, and D. Lohmann. Feature models in linux: From symbols to semantics. VaMoS, pages 65–72. ACM, 2016.

[43] L. R. Soares. Varxplorer: Reasoning about feature interactions. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE, pages 500–502. ACM, 2018.

[44] L. R. Soares, J. Meinicke, S. Nadi, C. Kästner, and E. S. de Almeida. VarXplorer: Lightweight Process for Dynamic Analysis of Feature Interactions. In *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 59–66. ACM, 2018.

[45] L. R. Soares, P.-Y. Schobbens, I. do Carmo Machado, and E. S. de Almeida. Feature interaction in software product line engineering: A systematic mapping study. *Information and Software Technology*, pages –, 2018.

[46] S. Souto, M. d'Amorim, and R. Gheyi. Balancing Soundness and Efficiency for Practical Testing of Configurable Systems. In *ICSE*, pages 632–642. IEEE Press, 2017.

[47] W. N. Sumner and X. Zhang. Algorithms for automatically computing the causal paths of failures. In *FSE*, pages 355–369, 2009.

[48] W. N. Sumner and X. Zhang. Comparative causality: Explaining the differences between executions. In *ICSE*, pages 272–281. IEEE Press, 2013.

[49] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *CSUR*, 47(1):6:1–6:45, 2014.

[50] A. von Rhein, S. Apel, and F. Raimondi. Introducing Binary Decision Diagrams in the Explicit-State Verification of Java Code. In *JPF Workshop*, 2011.

[51] A. von Rhein, A. Grebhahn, S. Apel, N. Siegmund, D. Beyer, and T. Berger. Presence-condition simplification in highly configurable systems. ICSE, pages 178–188. IEEE Press, 2015.

[52] C.-P. Wong, J. Meinicke, L. Lazarek, and C. Kästner. Faster Variational Execution with Transparent Bytecode Transformation. In *OOPSLA*. ACM, 2018.

[53] P. Zave. *Software Requirements and Design: The Work of Michael Jackson*, chapter Modularity in Distributed Feature Composition, pages 267–290. Good Friends Publishing Company, 2009.

[54] A. Zeller. Isolating Cause-Effect Chains From Computer Programs. In *FSE*, pages 1–10. ACM, 2002.