Feature Interactions in Highly Configurable Systems: A Dynamic Analysis Approach with Varxplorer

Larissa Rocha[1,2], **Eduardo Almeida**[1], **Ivan Machado**[1], **Christian Kästner**[2]

[1]Computer Science Department – Federal University of Bahia (UFBA)
Caixa Postal 15.064 – 91.501-970 – Salvador – BA – Brazil

[2]State University of Feira de Santana (UEFS) – Feira de Santana, BA, Brazil

[3]Institute for Software Research – Carnegie Mellon University
Pittsburgh, US

larissa@ecomp.uefs.br, esa@rise.br, ivan.machado@ufba.br,

kaestner@cs.cmu.edu

Abstract. *Highly-configurable systems provide significant reuse opportunities by tailoring system variants based on a set of features. Those features can interact in undesired ways which may result in faults. Thus, we propose VarXplorer, a dynamic and iterative approach to detect suspicious interactions. To evaluate whether VarXplorer helps improving the performance of identifying suspicious interactions, we performed two empirical studies. Our results shows that from the VarXplorer graphs, participants are able to identify suspicious interactions more than 3 times faster compared to the state-of-the-art tool. Additionally, the iterative detection process provides a more efficient feature interaction analysis, reducing the data developers needs to check to find problematic interactions.*

Resumo. *Os sistemas altamente configuráveis oferecem oportunidades significativas de reuso, adaptando variantes do sistema com base em um conjunto de features. Essas features podem interagir de formas indesejadas resultando em falhas. Assim, propomos o VarXplorer, uma abordagem dinâmica e iterativa para detectar interações suspeitas. Para avaliar se o VarXplorer ajuda a melhorar o desempenho da identificação dessas interações, realizamos dois estudos empíricos. Nossos resultados mostram que, a partir dos grafos do VarXplorer, os participantes identificaram interações suspeitas mais de três vezes mais rápido em comparação com a ferramenta do estado da arte. Além disso, o processo de detecção iterativa fornece uma análise mais eficiente das interações, reduzindo a quantidade de informação que um desenvolvedor precisa verificar para encontrar interações problemáticas.*

1. Introduction

Highly-Configurable Systems (HCS), such as Software Product Lines (SPL), may be composed of thousands of features (configuration options). The Linux kernel[1], for example, has more than 15,000 configuration options [Lotufo et al. 2010]. This large set of options may be combined in different ways, generating millions of possible configurations and developers should ensure that all valid combinations work correctly. Additionally,

[1]`https://www.kernel.org/`

interactions between two or more features may result in a surprising behavior that is not easily deduced from the analysis of each feature separately.

Feature interactions can be classified as either expected or unexpected. Expected (desired) interactions are related to features that are frequently combined to cooperate and contribute to an intended behavior. However, most interactions cannot be predicted upfront. These unexpected interactions can be further classified as either benign or problematic for a system. Most unexpected interactions may result in a benign behavior that does not cause any problem and might be crucial to integrate the functionalities of multiple features. Problematic are undesired feature interactions that may cause faulty or damaging system behavior. Identifying and classifying feature interactions is challenging as they only appear in certain test cases and configurations.

While it might be relatively simple to specify the behavior of a feature in isolation, anticipating and specifying all likely consequences of each feature interaction might not be possible. Some of the reasons are related to the number of configurations and feature interactions grow exponentially to the number of features; and the behavior of some interactions may be unpredictable in advance. Moreover, human effort is required, but people usually do not like writing specifications.

To address those challenges, recent analyses focus on detecting feature interaction problems from *global specifications*, i.e., specifications that all configurations of a system need to fulfill. Usually, these approaches check global specifications based on systematic sampling [Kim et al. 2013], combinatorial interaction testing [Cohen et al. 2007], model checking [Li et al. 2005], and variational execution [Meinicke et al. 2016]. However, since specifications at the feature level are usually missing, the mentioned approaches may not detect all incorrect system behavior, especially bugs not covered by global specifications and bugs that do not result in either a crash or other easily observable behavior.

In this study, we investigate the state-of-the-art on feature interactions and support developers on the detection of problematic feature interactions with VarXplorer. First, we carried out a systematic mapping study to understand *how do the existing approaches deal with feature interactions in the community*, which is the general question that drives the research [Soares et al. 2018c]. Additionally, five more specific questions were derived, regarding proposed solutions (RQ1), feature interactions types (RQ2), software lifecycle (RQ3), software domains (RQ4), and empirical assessment methods (RQ5).

From the 40 studies identified, more than 60% of them either provide an initial discussion on feature interactions or discuss how to identify interactions at early phases of the SPL development. Often, detection strategies are partial and only address specific points of a feature interaction investigation. On the one hand, when interactions are detected, it is not possible to identify which interactions cause the problems. On the other hand, when interactions are resolved with a specialized module or implementation changes, the previous step on how they came up with those interactions is not explained. In addition, the set of existing approaches is strictly dependent on software specifications [Soares et al. 2018c].

Based on the mapping results, we propose VarXplorer, a tool-supported approach that provides a dynamic feature interaction inspection process [Soares 2018, Soares et al. 2018b]. Instead of upfront specifications as most of the approaches do, we propose to inspect feature interactions as they are detected and incrementally classify them as either benign or problematic. VarXplorer is a dynamic approach automated by an Eclipse plug-in to identify feature interactions without any previous system or feature specification. From a configurable system and its test suite, VarXplorer proposes an iterative (runtime) analysis of interactions. For each test case, it creates a *feature interaction graph* (FIG) as a representation of *all* feature interactions, and shows interaction-dependent variables and feature relationships, such as the suppression of a feature by another.

The proposal was empirically evaluated through two studies. We run a controlled experiment to investigate whether FIG could improve the efficiency of identifying feature interactions compared to the state-of-the-art tool [Soares et al. 2018a]. Furthermore, we performed a second study to understand how the VarXplorer *iterative process* may improve and facilitate the identification of suspicious interactions [Soares 2019]. Our results show that, when using VarXplorer, developers are three times faster to detect interactions. Also, the VarXplorer iterative process reduces up to 50% the data produced by interactions, which simplifies the detection process and makes it easier to find problematic interactions.

The remainder of this paper is organized as follows. In Section 2, we introduce the proposed VarXplorer approach. Section 3 presents a controlled experiment that investigates how the FIG can assist developers during interaction analysis. Section 4 presents a second study to discuss the VarXplorer iterative process. Finally, section 5 summarizes the research presented in this paper.

2. VarXplorer: Feature Interaction Inspection Process

In a pair of features that interact, a feature might enable, require, overwrite variables, or even block the effect of another feature. Figure 1a shows an example illustrating the behavior of unexpected interactions. It represents a code excerpt modeled after WordPress (wp), an extendable blogging and content management system.[2] The example involves five features, `Weather`, `Smiley`, `Statistics`, `Fahrenheit`, and `Login`. Figure 1b shows two possible configurations among them. For *Configuration 1*, the features `Weather` and `Fahrenheit` interact intentionally to display the weather information in a desired format. However, for *Configuration 2*, the feature `Smiley` interacts with the `Weather` in an unintended way, although they do not crash the system. When they are together in the same system, the temperature is not showed and the system presents an unexpected output: ("[:weather☺").

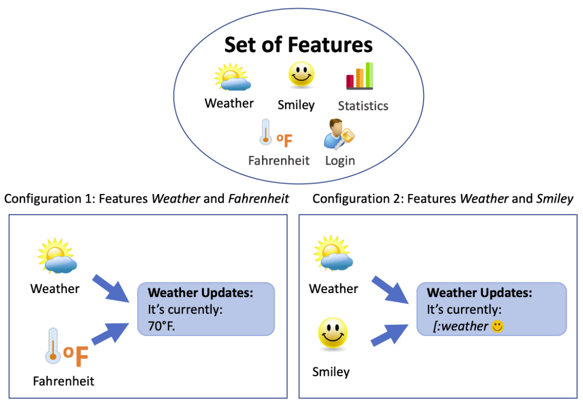
Feature interactions can be detected by comparing the executions of *all* system configurations. We use *variational execution* as an efficient method to compare executions [Meinicke et al. 2016]. In general, variational execution runs all program configurations, often efficiently, by sharing redundancies of the executions and values among these configurations. Additionally, our current approach focuses on pairwise feature interactions. They have been proved to be an effective and practical method to test software [Maity and Nayak 2005], especially with variational execution, where higher-order interactions are less common in practice [Meinicke et al. 2016].

Figure 2 shows how this process works for the WordPress example. In this example, we focus on just two options, `SMILEY` and `WEATHER` (lines 1-13 of the code

[2]`https://developer.wordpress.org/`

```
 1
 2 boolean STATISTICS, SMILEY, WEATHER, FAHRENHEIT, SECURE_LOGIN;
 3
 4     void createHtml(String c) {
 5         c = wpGetContent();
 6         if (SMILEY)
 7             c = c.replace(":]", getSmiley(":]"));
 8         if (WEATHER) {
 9             String weather = getWeather();
10             c = c.replace("[:weather:]", weather);
11         }
12         String head = initHeader();
13         print("<html><head>" + head + "</head><body>")
14         if (STATISTICS) {
15             int time = getCurrentTime();
16             printStatistics(time);
17         }
18         print("<div>" + c + "</div>");
19         String foot = wpGenFooter();
20         print("<hr/>" + foot + "</body></html>");
21     }
22
23     String getWeather() {
24         float temparature = 30;
25         if (FAHRENHEIT)
26             return (temperature * 1.8 + 32) + "°F";
27         return temperature + "°C";
28      }
```

(a) Feature interactions modeled after WorldPress [Meinicke et al. 2016]



(b) Example of two Wordpress configurations

Figura 1. Code excerpt modeled after WordPress.

in Figure 1a. Depending on the conditions `SMILEY` and `WEATHER`, there exists four execution paths, i.e., two options generate four possible combinations, as Figure 2a shows. The number in the nodes indicates the number of the lines according to Figure 1a. We observe that for all executions, lines 5 and 6 are always executed. In turn, lines 7 and 9 are only executed in certain configurations. Line 7 is executed when `SMILEY` is selected; and line 9 is executed when `WEATHER` is selected. Figure 2b shows how the variational execution works: a single run, instead of four, sharing all common parts among the executions and showing the differences. The main idea is to split the execution when the code presents differences and join again when it has to execute the same code. As data and control flows are shared, we are able to observe feature interactions in the differences of the execution and assignments of data [Meinicke et al. 2016].

Based on variational execution, we introduce VarXplorer, an iterative analysis and tool to inspect feature interactions [Soares 2018, Soares et al. 2018b]. Figure 3 shows an overview of our approach that proposes to incrementally analyze interactions. Given a configurable system and its test suite, we execute test cases (system inputs) looking for feature interactions. The developer then explores which interactions are problematic. Each test execution generates a FIG, a concise representation of all pairwise interactions among features. Based on the variational execution of a system, the FIG provides a visualization of which features interact, in addition to present their relationships and data context.

Only revealing which features interact (raw interactions) may not provide the developer with enough insights to identify whether a certain interaction is benign or problematic. 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 unintended way. To understand the relationship between features, we propose to investigate the relation that a feature may have over others, such as suppressing or requiring another feature. Interaction relationships may additionally be associated with the data context of the interaction, as the variables involved in the relation,

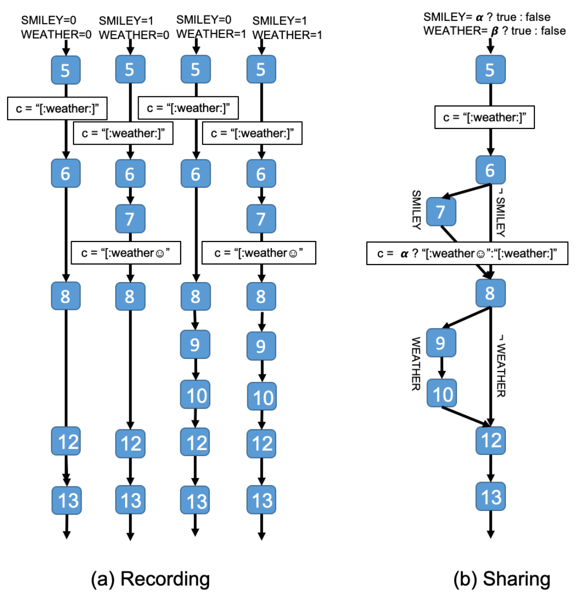


Figura 2. WordPress example executed with a traditional approach versus variational execution.

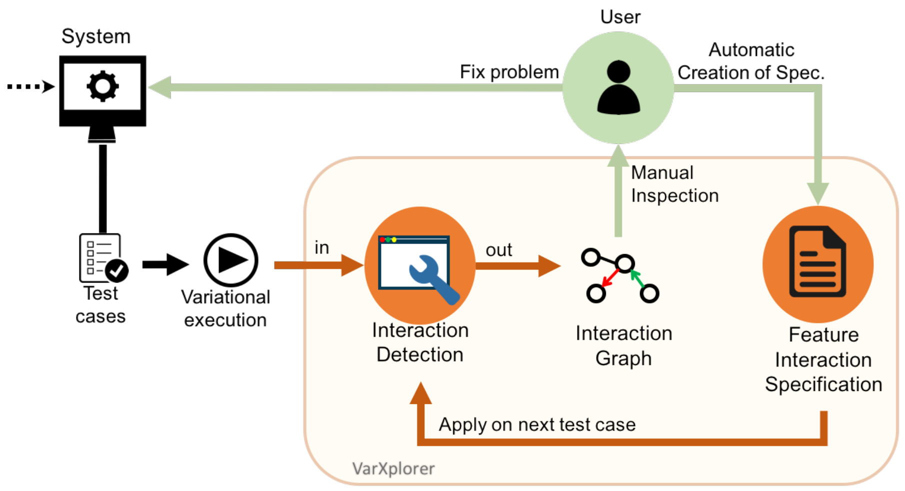


Figura 3. Overview of our approach.

for instance. The different values that a given variable may assume can be a signal that something wrong occurred.

Figure 3 shows the VarXplorer iterative process. VarXplorer is incremental in the sense that all tests of a test suite should be executed sequentially. We propose to start from smallest tests to the most complex ones. Each individual test creates a FIG, which based on developer inspection, it is automatically refined by removing benign interactions. This refinement is supported through a *feature interaction specification language* (FILang) to ensure that: (i) the developer does not see benign interactions again in future iterations (i.e., when executing other test cases); and (ii) any newly detected unintended interactions will be flagged in the future.

To analyze all the likely feature interactions on a system, the feature interaction detection should be applied over different inputs to achieve a high system coverage. However, when applied over real systems, the FIG may present a large amount of interactions and conditional variables. In addition, different FIG 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, such as, overwriting the same variables with the same values. In this way, the FILang represents a lightweight strategy to indicate that there is an interaction among features.

Based on the FILang, developers automatically create feature interaction specifications when marking interactions as either allowed or forbidden (right click on the line that connects two features) in the FIG. When allowing, they may remove interactions from features that are intended to interact, which "cleans" the graph and can facilitate finding interactions that represent a bug. Conversely, 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. The FILang does not require a formal description of the behavior of systems or features, as global and feature-based specifications do. Furthermore, those behavior specifications are usually missing. In particular, feature interaction specifications can be created according to three parameters: type (*Allow, Forbid*), relationship (*Require, Suppress, Any*), and target (*Variable, Method, Class, Any*).

2.1. Feature 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 a variational trace created from executing a test case, and the output is the FIG presenting all the interactions. The creation process of the FIG has two major steps: *pairwise detection* and *relationship analysis* [Soares et al. 2018b]. From the variational execution, we identify the pairs of features that interact and create a basic feature interaction graph (bFIG). Then, we perform the relationship analysis and refine the bFIG to include the underlying variables affected by the features to produce the *complete* FIG.

Pairwise detection. For pairwise detection, we collect a set $\mathbb{PC}$ with all the presence conditions in data and control flow present in the variational trace. $\mathbb{PC}$ contains all the conditions that shows how the features interact in the system. Control flow conditions are path conditions of the trace, and data flow conditions are formed by the conditions on each system variable. From $\mathbb{PC}$, we identify all pairs of features that interact together by finding feature pairs 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 as literals in p:

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

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

From Equation 1 and 2, we are able to collect all pairwise interactions. We use them to create the bFIG, a simple visualization of all interactions identified in the trace. For example, Figure 4 shows the execution of the WordPress (*wp*) example, corresponding to the code in Figure 1a. The figure 4 shows the presence conditions identified during the

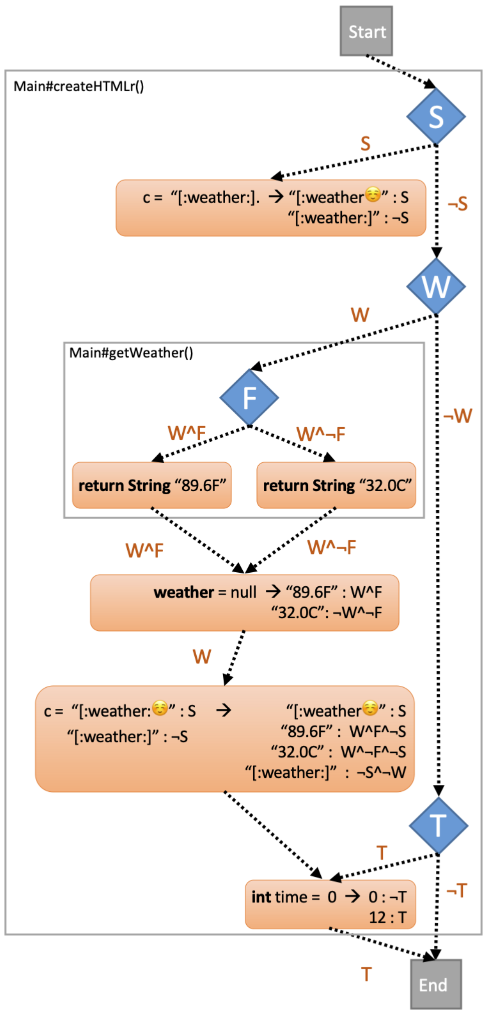
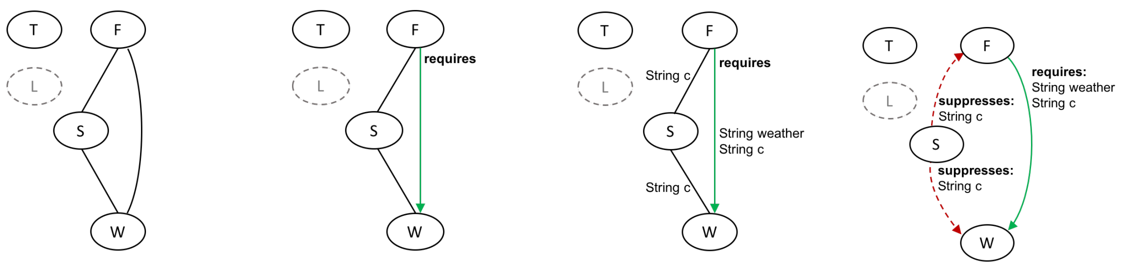


Figura 4. Variational trace of the WordPress example.

variational execution that add or change any functionality during the execution. Control-flow conditions are showed as arrows and data-flow conditions are showed in the rounded rectangles.

The WordPress example has five features: `SMILEY` (S), `WEATHER` (W), `FAHRENHEIT` (F), `STATISTICS` (T), and `SECURE LOGIN` (L). $\mathbb{PC}_{wp}$ represents the set of presence conditions of the Wordpress example. The $\mathbb{PC}_{wp}$ collected from Figure 4 has eleven unique presence conditions, as follows: $S, \neg S, W, \neg W, T, \neg T, W \wedge F, W \wedge \neg F, W \wedge F \wedge \neg S, W \wedge \neg F \wedge \neg S, \neg S \wedge \neg W$.

Based on the above equations, we identified three interactions ($\mathbb{I}_{wp}$) in the entire set of presence conditions $\mathbb{PC}_{wp}$, i.e., $\mathbb{I}_{wp}$ = {(F, W), (S, F), (S, W)}. Figure 5a shows the bFIG for our running example, illustrating the interactions in $\mathbb{I}_{wp}$. Although the program contains 5 features, only 3 out of them interact with each other. The remainder 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 bFIG shows which features interact with each other, it does not provide enough insight on *how* features interact. Thus, we also analyze each pair of features to



(a) bFIG (b) Flow-based analysis (c) Variables detection (d) Variables analysis

Solid black line: interaction. **Dashed line**: data flow interaction. **Dashed line around the feature**: features that has no effect in the execution. **Red arrow**: suppress relationship. **Green arrow**: require relationship.

Figura 5. Creation process of the WordPress FIG.

determine its relationship and the variables involved.

Relationship analysis. In this step, we investigate each pair to determine the effect one feature has on the other. We provide two complementary analysis: $\mathbb{PC}$-*based analysis* and *data-based analysis* [Soares 2018, Soares et al. 2018b]. 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 condition a given feature has an effect on the trace. If a feature *f1* has no effect on the trace, then the selection of *f1* never adds nor changes any functionality that was not present before [Nadi et al. 2014]. In the bFIG of Figure 5a, the dashed feature *L* is not active and, therefore, *L* has no effect in the WordPress trace. To the other features of the graph, *T*, *F*, *S*, and *W* (circles with solid lines), they have been executed and add functionalities to the execution. For example, to feature *S* have an effect on the execution of Figure 1a, the variable *time* must contain the current time and it has to be printed, i.e., lines 15-16 need to be executed.

In addition, we can analyze the effect of features on each other based on their relationships. We define two effects, also called relationships, *suppress* and *require*, 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.*

Relationship based on PC. Formally, the effect of a feature f on a condition p is given as the function $\mathbb{U}(f, p)$, as follows:

$$\mathbb{U}(f, p) = p[f/True] \oplus p[f/False] \qquad (3)$$

The function $\mathbb{U}(f, p)$ give us the condition in which f has effect in p, using xor ($\oplus$). A feature f has no effect on p if enabling it (f as $True$) or disabling (f as $False$) it does not affect the value of p; therefore f does not have an effect on selecting the corresponding code fragment under the condition p. In other words, we say that feature f has no effect in condition p if $(f \leftarrow True)$ is equivalent to $(f \leftarrow False)$, where $(f \leftarrow y)^3$ means substituting every occurrence of f in p by y. When changing the feature to false or true, and the execution did not present any difference, it is because the feature has no effect on that program.

Otherwise, a feature f has an effect on p when enabling and disabling the feature in p, it presents a different result at least for one configuration, which means that different code fragments are executed. This method of verifying whether a feature is enabled or not is known as unique existential quantification [Nadi et al. 2014]. For example, to determine the effect of feature FAHRENHEIT (F) on the presence condition $W \wedge F$, we would substitute F with *True* and *False*, as follows:

$$\begin{aligned} \mathbb{U}(F, W \wedge F) = &= p[f/True] \oplus p[f/False] \\ &= (W \wedge F)[F/True] \oplus (W \wedge F)[F/False] \\ &= (W \wedge True) \oplus (W \wedge False) \\ &= W \end{aligned} \tag{4}$$

Thus, on the code blocks that the condition $(W \wedge F)$ holds, F has an effect if and only if (iff) W is selected. Similarly, we can determine the overall effect of a feature g taking in account all conditions in $\mathbb{PC}$. In this way, we need to consider the disjunction of all feature effects of g on each presence condition $p \in \mathbb{PC}$:

$$\mathbb{U}(g, \mathbb{PC}) = \bigvee_{p \in \mathbb{PC}} \mathbb{U}(g, p) \tag{5}$$

The result of Equation 5 corresponds to the condition under which a feature g has an effect on the whole system's presence conditions. Hence, we can now determine the effect of F on the whole WordPress execution, which is given by the disjunction of all feature effects considering all the 11 presence conditions of the Wordpress ($\mathbb{PC}_{wp}$). Thus, the effect of F on the whole program is calculated as:

$$\begin{aligned} \mathbb{U}(F, \mathbb{PC}_{wp}) =& \{\mathbb{U}(F, S) \vee \mathbb{U}(F, \neg S) \vee \mathbb{U}(F, W) \vee \mathbb{U}(F, \neg W) \vee \\ & \mathbb{U}(F, T) \vee \mathbb{U}(F, \neg T) \vee \mathbb{U}(F, W \wedge F) \vee \\ & \mathbb{U}(F, W \wedge \neg F) \vee \mathbb{U}(F, W \wedge F \wedge \neg S) \vee \\ & \mathbb{U}(F, W \wedge \neg F \wedge \neg S) \vee \mathbb{U}(F, \neg S \wedge \neg W)\} \\ \mathbb{U}(F, \mathbb{PC}_{wp}) =& W \end{aligned}$$

In this case, it confirms that F only has an effect iff W is selected, considering the whole program, and not only one single condition. The Equation 5 identifies explicit relationships between features (suppress and require). We say f_2 *suppresses* f_1 in an execution with presence conditions $\mathbb{PC}$ iff the result of the following equation is a tautology:

$$f_2 \implies \neg\mathbb{U}(f_1, \mathbb{PC}) \tag{6}$$

Otherwise, we say f_1 *requires* f_2 in a trace iff the result of Equation 7 is a tautology:

$$\neg f_2 \implies \neg\mathbb{U}(f_1, \mathbb{PC}) \tag{7}$$

[3] p is implicit in this notation.

For example, the effect of the feature `FAHRENHEIT` (F) on the WordPress execution results in $\mathbb{U}(F, \mathbb{PC}_{wp}) = W$. 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 Figure 1a: line 25 is only executed when the decision in line 8 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 example of 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). To conclude our analysis, we need to perform the same analysis for each pair of features to determine the effects of each feature in an interaction. Figure 5b shows the result of the relationship analysis based on $\mathbb{PC}$ for our running example. It presents the feature effect analysis for all pairs in $\mathbb{PC}_{wp}$. In this case, we only found an explicit feature effect in the interaction (F, W), which is a *require* relationship. The other two interactions, (S, F) and (S, W), did not expose any explicit flow relation. To further explore additional relationships between features, we complement the flow analysis with a data analysis.

Relationship based on data. In HCS, the same variable can assume different values under different configurations. Features that do not directly interact on the control flow may still interact by controlling the same variables. *Conditional variables* are variables in which the values depend on more than one feature. Unexpected data values may reveal bugs from unintended interactions on variables. In this data analysis step, we further perform two main tasks: (i) we collect the data context of interactions, based on the variables they interact on; and (ii) we analyze feature effects on data to find feature relationships related to variables (e.g., a feature may suppress another related to a given variable) [Soares 2018, Soares et al. 2018b].

Thus, we investigate each conditional variable to analyze the context of data interactions. A *variable context* is the set of conditions that affect the value of one variable. From the variable context analysis, we can identify all pairs of features that interact on the variable's value. To identify feature interactions in variables (data interaction), we consider the same Equation 2, but replace the set of presence conditions $\mathbb{PC}$ with the context of a given variable. For instance, the WordPress example has three variables (`c`, `weather`, and `time`), but just two (`c` and `weather`) are conditional variables. Since the variable `time` only depends on feature *T* (Figure 1a), it is not part of any data interaction. The graph in Figure 5c shows all variables involved in WordPress' interactions.

To help developers understand what is happening in each variable, we detect relationships on variables and present them in the graph. Thus, we again investigate the feature effect of each feature pair, but now only related to the presence conditions of the variable being analyzed. The analysis of feature effect per variable is analogous to the analysis of the entire set of presence conditions $\mathbb{PC}$, describe in Equation 5. The only difference is: in place of $\mathbb{PC}$, we use the context of a variable [Soares et al. 2018b].

As a result, we found that `SMILEY` (`S`) *suppresses* `WEATHER` (`W`) in relation to variable `c`. When `S` is present in the configuration, `W` effect is blocked in the program and, thus, `W` cannot override the value of variable `c` as it should. Furthermore, `S` also *suppresses* `FAHRENHEIT` (`F`) in relation to *c*. Therefore, `S` *suppresses* both `WEATHER` and `FAHRENHEIT`. Figure 5d shows the complete FIG for our WordPress example. Since those new relationships do not cover all the information of a feature, but just the variable analyzed, we call them *partial relationships* and they are represented as dashed directed arrows.

3. Controlled experiment

We performed a controlled experiment to understand how FIGs can help users identify problematic interactions [Soares et al. 2018a]. The study investigates and compares the ability of users to identify problematic interactions with and without VarXplorer, in a setting with different tasks and systems. Therefore, we compare VarXplorer to the current state-of-the-art tool in the area, Varviz [Meinicke et al. 2016]. Varviz is an Eclipse plug-in that enables programmers to use variational traces for debugging interaction faults. We aimed to answer the following main question: *Does VarXplorer help developers 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?

Before the main experiment, we conducted two pilot studies with 8 graduate students from 2 universities in Brazil and in the US. After that, the real experiment was executed with 24 participants from different universities and companies in Brazil. We measured the effort to identify a buggy interaction based on the information provided by the FIG. We used two systems very used in the literature once they contain many interactions: Elevator [Plath and Ryan 2001] and Telephone [Griffeth et al. 2000]. Then, we compared VarXplorer with Varviz [Meinicke et al. 2016].

We designed our experiment as a within-subjects study. For this design, the same group of participants receives more than one treatment. In this way, all participants performed tasks using both tools, VarXplorer and Varviz. We designed 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 present just one suspicious interaction for each system and a couple of benign interactions. We provided the participants with the description of each feature in the target system, test case scenario documentation, and the system's source code.

For the statistic analysis of our data, we conducted an analysis of variance using a within-subjects ANOVA. It is a parametric test for determining whether significant differences occur in an experiment containing two or more conditions. We used the Shapiro-Wilk normality test, the Bartlett test of homogeneity of variances, and the Tukey HSD test to the multiple comparisons of means. As a result, we found that participants using VarXplorer outperformed participants using the Varviz tool. The former group took on average 3 min to perform each task (finding the problematic interaction), while the latter had on average 9 min. All the participants could identify the suspicious interaction in both tasks, which is why we compare time and not also correctness. Thus, all of them finished successfully their tasks.

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 they used the tools and systems had any influence on the results. For the systems group, the data from the order of the systems were not different, i.e., the order of the systems does not statistically influence the results (p-value = 0.803). For the tools groups, it presented a large effect size between the groups that used VarXplorer against the Varviz groups. According to the ANOVA test, we got 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 [Heiman 2013]. 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, i.e., 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 did 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.

We also performed a qualitative analysis based on experiment video and audio recordings, and interviews. From the qualitative analysis, we leveraged the following observations:

- The types of feature's relationships guide the analysis and decrease the analysis time;
- To use VarXplorer, the developer may not need to know details of the implementation and programming language;
- VarXplorer shows non-interacting features and no-effect features, which also might be indicatives of bugs;
- VarXplorer and Varviz may complement each other.

The results confirmed that the VarXplorer relationships graphically represented as arrows and colors in the FIG can make the developer's work easier and faster. Also, VarXplorer only shows conditional variables, which reduces the amount of information shown to developers [Soares et al. 2018a].

4. Exploratory study

We performed an exploratory study complementary to the controlled experiment [Soares 2019]. Instead of just evaluating the FIG, the second study explores the entire approach to investigate how the iterative and interactive approach may support the discovery of suspicious interactions. We aimed to answer the following question: *How does the iterative process on individual test cases reduce the complexity of identifying interactions?*

Iterativeness stands for the potential to optimize the 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 were also interested in understanding how much effort we could save using interaction specifications when executing test cases. Whenever a developer *allows* or *forbid* an interaction in the graph, 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.

We conducted this study on the basis of the RiSE Event SPL, which supports organizers of a conference [Neto et al. 2016]. The test suite of the SPL contains 15 test cases (T1-T15). We executed the tests in order of complexity, from the smallest to the most complex test (increasing number of features and test activities). The analysis procedure consisted of running each test separately until we have tested all system functionalities. Thus, every test case is run after the analysis of its preceding case. When executing a given test case, the variational execution tests all combinations among the features of that test in one single execution. Hence, the test suite does not need to contain one test to each configuration. Consequently, the variational execution reduces the need for additional tests.

To analyze interactions, developers have to check the FIG and judge them as either benign or suspicious. If developers identify suspicious interactions, we suggest them to look for the causes of the problem and fix them in the source code before running the next test case. Then, they may have to run the test again and check the graph after the fix to guarantee the problem has been fixed. At this point, the FIG 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 not appear anymore. Hence, this graph need to be checked again, and the process should restarts. The interactions of this new FIG should be checked until all interactions have been understood.

During interaction analysis of the RiSE Event SPL, we observed that the use of reduced graphs (when known interactions are removed) decreased by about 50% the amount of information that the user needs to analyze to identify suspicious interactions. Specifications clean the graphs to help developers focus on new interactions that are likely to present problems. For example, the test of number #10 (T10) had 19 interactions, but 11 interactions and 58 variables were removed from specifications [Soares 2019]. The user only had to judge the 8 remaining interactions showed in the reduced graph. The others were present in the previous tests (T1 - T9), being previously marked as benign. For T10, the user had about 58% less interactions and almost 45% less variables to analyze when using the specifications provided by VarXplorer. As the test cases grow and features are repeated over tests, many interactions are repeated among them.

The test suite presented a total of 431 conditional variables and 143 different interactions. 118 were *require* interactions and 25 were *suppress* interactions, scattered over 15 tests. 11 *suppress* interactions and 6 *require* interactions presented problems. Thus, 6 out of 15 tests presented feature interaction problems, which shows that 40% of the total of tests had problems. 17 interactions out of 143 were problematic, which represents less than 12% of the total interactions. As expected, most of the interactions that appeared in the FIGs were benign. Our results show that the feature relationships (require and suppress) may indicate the presence of undesired interactions, which assist developers during feature interaction analysis.

Additionally, we did not find any problem that led the program to a crash. Most of the problems were related to lack of source code modularity and incorrect implementation, such as, wrong 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, since it is necessary to test *all* interactions.

5. Concluding Remarks

Our work pursued a twofold goal, mapping the state-of-the-art on feature interactions and identifying problematic interactions without upfront specifications. We developed an inspection process supported by VarXplorer that provides an automatic way to identify feature interactions based on the software execution. From the execution of a test case, we analyzed interactions based on its control and data flow. Moreover, we presented additional indicators that can help developers to identify which interactions may represent a bug, such as the suppression of one feature by another and the variables involved in the interaction. To validate our approach, we performed two empirical studies that confirmed VarXplorer is three times faster to detect buggy interactions than the state-of-the-art tool. Further studies could investigate a test case generation process to use in combination with our approach to cover the most representative inputs of a given system. In addition, scalability analysis studies could also be carried out.

Referências

Cohen, M. B., Dwyer, M. B., and Shi, J. (2007). Interaction testing of highly-configurable systems in the presence of constraints. In *Int. Symp. on Software Testing and Analysis*, ISSTA, pages 129–139, NY, USA. ACM.

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

Heiman, G. W. (2013). *Basic statistics for the behavioral sciences*. Cengage Learning.

Kim, C. H. P., Marinov, D., Khurshid, S., Batory, D., Souto, S., Barros, P., and D'Amorim, M. (2013). SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems. In *9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, pages 257–267, NY, USA. ACM.

Li, H. C., Krishnamurthi, S., and Fisler, K. (2005). Modular verification of open features using three-valued model checking. *Automated Software Engg.*, 12(3):349–382.

Lotufo, R., She, S., Berger, T., Czarnecki, K., and Wasowski, A. (2010). Evolution of the linux kernel variability model. Int. Software Product Lines Conference, pages 136–150, Berlin. Springer-Verlag.

Maity, S. and Nayak, A. (2005). Improved test generation algorithms for pair-wise testing. In *16th IEEE Int. Symposium on Software Reliability Engineering (ISSRE)*.

Meinicke, J., Wong, C. P., Kästner, C., Thüm, T., and Saake, G. (2016). On essential configuration complexity: Measuring interactions in highly-configurable systems. In *31st Int. Conf. on Automated Software Engineering*, pages 483–494.

Nadi, S., Berger, T., Kästner, C., and Czarnecki, K. (2014). Mining configuration constraints: Static analyses and empirical results. ICSE, pages 140–151, NY, USA. ACM.

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

Plath, M. and Ryan, M. (2001). Feature integration using a feature construct. *Science of Computer Programming*, 41(1):53 – 84.

Soares, L. R. (2018). Varxplorer: Reasoning about feature interactions. In *Proceedings of the 40th Int. Conf. on Software Engineering - Research Competition*, pages 500–502.

Soares, L. R. (2019). *Feature Interactions In Highly Configurable Systems: A Dynamic Analysis Approach With Varxplorer*. PhD thesis, Federal University of Bahia (UFBA).

Soares, L. R., Meinicke, J., Nadi, S., Kästner, C., and de Almeida, E. S. (2018a). Exploring feature interactions without specifications: A controlled experiment. In *17th Int. Conf. on Generative Programming*, pages 40–52.

Soares, L. R., Meinicke, J., Nadi, S., Kästner, C., and de Almeida, E. S. (2018b). Varxplorer: Lightweight process for dynamic analysis of feature interactions. In *12th Int. Workshop on Variability Modelling of Software-Intensive Systems*.

Soares, L. R., Schobbens, P.-Y., Machado, I. C., and Almeida, E. S. (2018c). Feature interaction in software product line engineering: A systematic mapping study. *Information and Software Technology*, 98:44–58.