VarXplorer: Lightweight Process for Dynamic Analysis of Feature Interactions

Larissa Rocha Soares
Carnegie Mellon University, USA
Federal University of Bahia, Brazil

Jens Meinicke
Carnegie Mellon University, USA
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

Features in highly configurable systems can interact in undesired ways which may result in faults. However, most interactions are not easily detectable as specifications of feature interactions are usually missing. In this paper, we aim to detect interactions and to help create feature-interaction specifications. We use variational execution to observe internal interactions on control and data flow of highly configurable systems. The number of potential interactions can be large and hard to understand, especially as many interactions are benign. To help developers understand these interactions, we propose feature-interaction graphs as a concise representation of all pairwise interactions. We provide two analyses that provide additional details about interactions, namely suppress and require interactions. Finally, we propose a specification language that enables developers to define different kinds of allowed and forbidden interactions, which help to detect interaction faults. Our tool, VarXplorer, provides a visualization of feature-interaction graphs and supports the creation of feature interaction specifications. VarXplorer also provides an iterative analysis of feature interactions allowing developers to focus on suspicious cases.

CCS CONCEPTS

• **Software and its engineering** → **Feature interaction**; **Reusability**; *Specification languages*;

KEYWORDS

Feature Interaction, Configurable Software, Feature Interaction Specification, Variability-Aware Execution

1 INTRODUCTION

Highly configurable systems provide significant reuse opportunities by tailoring system variants based on a set of *features* (aka. configuration options) [25, 32]. Such systems may be composed of thousands of features. For example, Eclipse has more than 1,600 plugins [25] and the Linux kernel has more than 15,000 configurable options [19, 27]. This large set of options may be combined in different ways, and developers must guarantee that all valid combinations work properly. A common problem in highly configurable systems is that a *feature interaction* between two or more features may result in a surprising behavior that is not easily deduced from the analysis of each feature separately [2]. Even if a system behaves as expected most of the time, it may exhibit unexpected and unwanted interactions under specific feature combinations.

Determining the influence of feature interactions on a system's behavior is challenging. 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 [5]. Anticipating and specifying all likely consequences of each possible feature interaction might be not possible, mainly due to the fact that (i) the number of configurations and feature interactions grows exponentially in relation to the number of features [11]; (ii) the behavior of some interactions may be unknown and unpredictable in advance [2]; and (iii) human effort is required, but people usually do not like writing specifications.

Instead, recent analyses focus on detecting feature interaction bugs from *global specifications*, i.e., specifications that all configurations of a configurable system need to fulfill, such as requiring that each configuration does not crash [30]. Usually, these approaches are based on systematic sampling [14, 16, 28], combinatorial interaction testing [10, 20, 24], model checking [3, 7, 9, 17, 31], or variational execution [6, 15, 21, 23] to cover large spaces, but they check global specifications for all configurations.

Since specifications at the feature level are usually missing, the above approaches may not detect all incorrect system behavior,

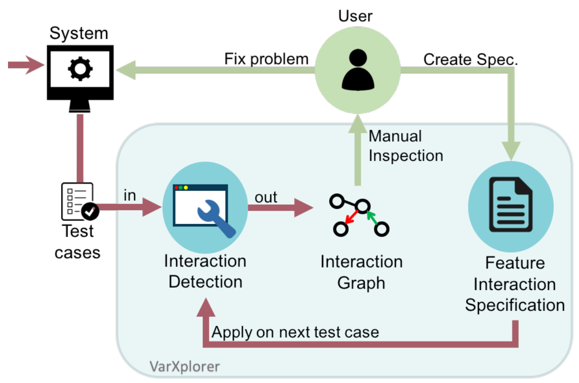

Figure 1: Overview of our approach to iteratively and automatically inspect feature interactions with VarXplorer

VAMOS 2018, February 7–9, 2018, Madrid, Spain
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5398-4/18/02…$15.00
https://doi.org/10.1145/3168365.3168376

specially bugs not covered by global specifications and bugs that do not result in a crash or other easily observable behavior. For instance, in the simplified WordPress example of Figure 2, the options *weather* and *smiley* interact 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: instead of replacing the "[:weather:]" tag to the current temperature (e.g., 70°F), it is rewritten to "[:weather☺" and presented to the user in place of the temperature.

It is hard to reason about interactions without feature specifications. When statically detected in the source code, (i) predicted interaction may never appear during system execution; (ii) many feature interactions can be observed only at runtime; and (iii) it is difficult to automatically determine if an unexpected interaction is benign or represent a real problem. Although dynamic approaches overcome drawbacks of static analysis by analyzing systems at runtime, identifying problematic interactions still remains challenging, especially when no feature specification is provided.

Instead of upfront specifications, we propose to inspect feature interactions as they are detected and incrementally classify them as benign 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 [21, 23]. It performs variational execution to simultaneously execute all system configurations. VarexJ reveals the differences among configurations on both control and data flow [21], and represents them as a variational trace. An interaction is represented as a control-flow or state difference in the system that depends on two or more options.

In this work-in-progress paper, we propose VarXplorer, an iterative and interactive analysis that inspects feature interactions from the variational traces generated by VarexJ. Figure 1 shows an overview of our approach: given a configurable system and a set of test cases, we detect interactions and provide an incremental analysis of the relationship between features, illustrated through a feature-interaction graph.

To further support developers in understanding the detected interactions, we analyze control and data flow interactions to present additional indicators, such as the suppression of one feature by another. We also mark each interaction with additional helpful information, as for example the affected program variables. We present this feature-interaction graph to developers for *manual inspection*. Based on this inspection, they may indicate intended behavior and also select interactions as forbidden through the feature interaction specification language (*create spec.*). For unintended interactions, developers may go back and *fix the problem* in the code. The graph is then refined as more test cases are run, while also taking into account the documented interaction specifications (*apply on next test case*). 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

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

  void createHtml() {
      String c = wpGetContent();
      if (SMILEY)
          c = c.replace(":]", getSmiley(":]"));
      if (WEATHER) {
          String weather = getWeather();
          c = c.replace("[:weather:]", weather);
      }
      String head = initHeader();
      print("<html><head>" + head + "</head><body>");
      if (STATISTICS) {
          int time = getCurrentTime();
          printStatistics(time);
      }
      print("<div>" + c + "</div>");
      String foot = wpGenFooter();
      print("<hr/>" + foot + "</body></html>");
  }

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

Figure 2: Feature interactions modeled after WorldPress [21].

potential bugs by automatically removing benign interactions from the graph. In summary, we make the following contributions:

- We detect interactions based on both control and data flow;
- We determine the relationships between features and present two classes of interactions, namely suppress and require interactions. Those classes provide details how features interact to support users in identify unintended interactions;
- We implemented feature-interaction graphs, a concise visual representation of feature interactions identified at runtime;
- We propose an interaction specification language to allow and forbid interactions on data and control flow;
- We present an iterative and interactive approach to refine the feature-interaction graph and remove interactions that do not represent a bug, allowing the developer to focus only on suspicious cases.

2 ON DETECTING FEATURE INTERACTIONS: STATE OF THE ART

A *feature* describes a unit of functionality of a software system that satisfies a requirement [1]. Products of highly configurable systems (resp. software product lines) can be composed by selecting a set of features. Henceforth, we use the term feature to refer to any configuration option, module, or component in a configurable system.

Feature interactions. Often, the development team needs to deal with unexpected system behavior due to interactions between features. Features developed and tested separately may present a different behavior when combined in a system. A *feature interaction* is observed when the combined behavior of two or more features differs from the individual behaviors of both features [8, 12, 33]. For example, one feature can interfere with, enable, or overwrite the effects of another feature.

Features are frequently combined to cooperate to an intended behavior (expected interactions). However, most interactions cannot be predicted upfront. Unexpected interactions can be classified as either benign or problematic to a system. Problematic are undesired interactions that may cause faulty or damaging system behavior, such as crashes. However, most interactions, although unexpected, may result in a benign behavior that does not cause any problem to the system or might even be essential to coordinatre the functionalities of multiple features. Detecting and classifying feature interactions is challenging as they only appear in certain test cases and configurations (variants of a system composed of different feature combinations).

In Figure 2, we show an example illustrating both benign and problematic behavior. In the code excerpt modeled after WordPress, an extendable content management system, the features *weather* and *fahrenheit* interact intentionally to display the weather information in desired format. However, the feature *smiley* interacts with *weather* in an undesired way in some executions: As *smiley* replaces a part of the weather tag, the option *weather* has no effect if *smiley* is selected.

Global and feature-based specifications. Detecting all feature interactions requires having specifications for all system configurations. However, this usually does not scale to the large number of possible configurations. Another strategy is to specify features in isolation; a *feature-based specification* describes the behavior of a feature in isolation without any explicit reference to other features [30]. Such behavior is supposed to be preserved independent of other features in the system. For example, in a product line of electronic messages, a feature-based specification for the feature *Encrypt* is that the encryption key must be valid independent of what other features might do [4, 13]:

$$\mathbf{AG}\ outgoing(email\ \mathrm{e}) \Rightarrow (\mathrm{e}.isEncrypted \Rightarrow valid_{addr}(\mathrm{e}.encryptionKey))$$

Several approaches work with feature-based specifications to detect interactions. Li et. al [17] 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 [3] 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.

Conversely, *global specifications* represent a widespread strategy to reduce the effort of creating specifications for individual features, since they cover all configurations using general requirements [30]. Typical global specifications are requirements that the system should not crash and that it fulfills certain functional requirements in all configurations (e.g., passes all test cases). In the previously mentioned product line example, a global specification is that an outgoing e-mail message must have valid sender and

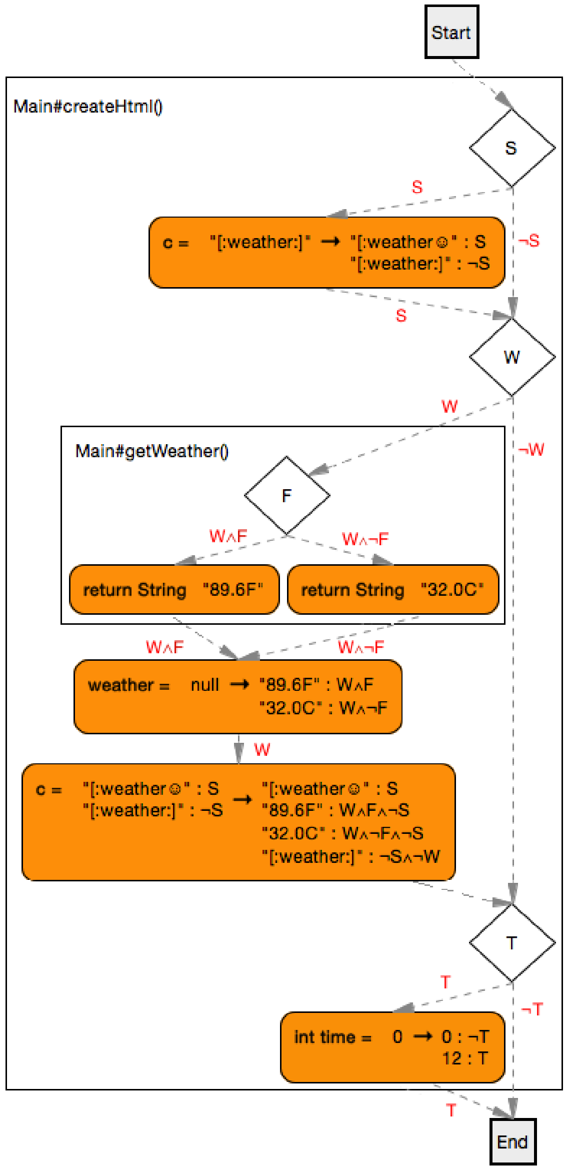

Figure 3: Variational trace of the WordPress example showing interactions among features. S: Smiley, W: Weather, F: Fahrenheit, T: Statistics.

receiver addresses independent of what any features might do [4]:

$$\mathbf{AG}\ outgoing(email\ \mathrm{e}) \Rightarrow valid_{addr}(\mathrm{e}.from) \wedge valid_{addr}(\mathrm{e}.to)$$

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 [14, 16, 28], combinatorial interaction testing [10, 20, 24], model checking [3, 7, 9, 17, 31], and variational execution [6, 15, 21, 23].

There exists a lot of work to detect faults caused by feature interactions, as well as techniques to resolve them. However, detecting unexpected feature interactions that do not lead to a crash (at least not for the given test cases), but that cause faulty behavior, remains an open challenge. In our work, we approach the challenge of identifying feature interactions without upfront existing specifications. We aim to help developers to dynamically identify potentially faulty feature interactions and to automatically create feature interaction specifications based on an iterative and interactive approach.

Dynamic Detection of Feature Interactions. In our work, we consider any state difference in a system that depends on more than one feature as an interaction, even if it does not cause a crash or any observable behavior differences. Such interactions describe fine grained internals of the system which may often be benign. Still, some of them can indicate faulty or unexpected behavior, such as an interaction among features that overwrite the same variables.

Feature interactions can be detected by comparing the executions of all system configurations. Feature interactions are then manifested in the differences in data and in the control flow that depend on multiple features. We use variational execution [21] to efficiently compare the executions of *all* configurations. Variational execution is able to scale to large configuration spaces due to its aggressive sharing abilities of redundancies among the executions of all configurations. As data and control flow is shared, we are able to observe feature interactions in the differences of the execution and assignments of data [21].

A *variational trace* is a concise representation of the differences, and the conditions to trigger those differences, among the executions of all configurations. It shows all interactions on data and control flow for a single test case in all configurations. Specifically, we use the variational interpreter VarexJ [21] and its extension (VarViz[1]) to create the variational trace. In Figure 3, we show the variational trace for our WordPress example, corresponding to the code in Figure 2. The trace shows the presence conditions identified by VarexJ that add or change any functionality during the execution.

Presence conditions, presented by VarViz, are propositional expressions over options that determine when a specific code artifact is executed [22]. We identify two main types of presence conditions: (i) *control-flow conditions*, expressions that define each path condition in a trace (arrows); and (ii) *data-flow conditions*, expressions responsible for changing the value of a given variable (rounded rectangles). In addition, diamonds in the trace represents decisions that affect the control flow. For example, the first decision (Line 5) separates the control flow to execute line 6 depending on the selection of option `SMILEY` (indicated with S and $\neg S$ on the arrows). The trace shows further the causes of differences in data. For example, in Line 6 the value of `c` is replaced if `SMILEY` is selected.

3 ITERATIVE ANALYSIS OF FEATURE INTERACTIONS

In Figure 1, we present an overview of our process to incrementally analyze feature interactions. 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 among features. Based on the variational trace of a system, the graph provides a visualization of which features interact, in addition to presenting their relationships and data context.

Only 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. 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 for example 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 in identify problematic interactions.

Our interaction detection process is incremental in the sense that, based on user inspection, the graph is automatically refined by removing benign interactions. 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., when executing other test cases); and (ii) that any newly detected unintended interactions are flagged in the future. The goal is to incrementally remove intended interactions in order to focus on unintended interactions.

Unlike global and feature-based specifications, that (respectively) represent the behavior of configurations and features, interaction specifications aim to point out that there exists an interaction between two features, without the need to formally specify its behavior. To make specifications easy to create, developers can mark interactions as either allowed or forbidden through a right click on the line that connects two features in the graph.

3.1 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 a variational trace created from executing a test case, 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. This complete graph provides more details about how the features interact, which goal is to support developers to understand problematic interactions.

3.1.1 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, which represents all the features that interact in the system. Control

[1]https://github.com/meinicke/varviz

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 all conditions in $\mathbb{PC}$, the set of features of a system ($\mathbb{F}$), and the set of all possible pairs of features in a system, relation $\mathbb{I}$ represents only the pairs that interact:

$$\mathbb{I} \subseteq \mathbb{F} \times \mathbb{F} \tag{1}$$

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:

$$f \blacktriangleright p \ := \ f \text{ occurs as literal in } p \tag{2}$$

$$\mathbb{I} = \{(f_1, f_2) \mid p \in \mathbb{PC} \wedge (f_1 \blacktriangleright p) \wedge (f_2 \blacktriangleright p)\} \tag{3}$$

From Equation 2 and 3, we are able to collect all pairwise interactions. We use them to create the *basic feature interaction graph*, a simple visualization of all interactions identified in the trace. Based on Equation 2, we can also determine the set of *active features* ($\mathbb{A}$) in a system, all the features that appear in presence conditions and are responsible for the functionalities executed in the system, by:

$$\mathbb{A} = \{f \mid p \in \mathbb{PC} \wedge (f \blacktriangleright p)\} \tag{4}$$

For example, our running example has five features (*S, T, W, F,* and *L*). Based on the above equations, we identified four active features $\mathbb{A}_{wp}$ and three pairs of interactions $\mathbb{I}_{wp}$ in the entire set of presence conditions $\mathbb{PC}_{wp}$, as follows:

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

Figure 4a shows the basic graph for our running example, illustrating the interactions in $\mathbb{I}_{wp}$. Although the program of Figure 2 contains five features, only three of them interact with each other, as shown by the edges in Figure 4a. The other two are non-interacting features; they either do not interact with any other feature during system execution (they are activated but do not interact) or are not executed in any configuration related to the current test case (they are not activated). In general, some features do not interact, because they cannot be simultaneously selected due to constraints in the variability model, or their implementations are orthogonal [21].

The basic graph (Figure 4a) may support developers in the detection of incorrect interactions. From the visualization, they can identify features that should not been interacting, or even missing interactions. 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 from the graph to determine relationships that further describe the interaction. To support users in identifying problematic interactions, we also analyze the variables involved.

3.1.2 Relationships Analysis

In the relationship analysis, we investigate each pair to determine the effect one feature has on the other. In this step, we provide two complementary analysis: $\mathbb{PC}$*-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 features 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 does a given feature have an effect on the trace. If a feature *f1* has no effect on the trace, then the selection of *f1* never adds or changes any functionality that was not present before [22]. In the basic graph of Figure 4a, the dashed feature L is not active and, therefore, L has no effect in the WordPress trace. Inactive features never have an effect, but beyond this we can analyze the effect of features on each other: *suppress* and *require* relationships. 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. In turn, 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. We investigate each presence condition (on control and data flow) to detect feature effects in interactions. The feature effect is given by analyzing the effect of a given feature on the set of presence conditions. Formally, the effect of f on a condition p is given as the function $\mathbb{U}(f, p)$, as follows:

$$\mathbb{U}(f, p) = (f \leftarrow True) \oplus (f \leftarrow False) \tag{5}$$

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, 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 [22].

Similarly, we can determine the overall effect of a feature f taking in account all conditions in $\mathbb{PC}$. In this way, we need to consider the disjunction of all feature effects of f on each presence condition $p \in \mathbb{PC}$:

$$\mathbb{U}(f, \mathbb{PC}) = \bigvee_{p \in \mathbb{PC}} \mathbb{U}(f, p) \tag{6}$$

The result of Equation 6 corresponds to the condition under which a feature f has an effect on the whole system's presence conditions. We can use the Equation 6 to identify the suppress and require relationships. We say f_1 *suppresses* f_2 in a trace with presence conditions $\mathbb{PC}$ iff:

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

Otherwise, we say f_1 *requires* f_2 in a trace iff:

$$\mathbb{U}(f_1, \mathbb{PC}) = f_2 \tag{8}$$

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. This behavior can be observed in Figure 2: Line 25 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

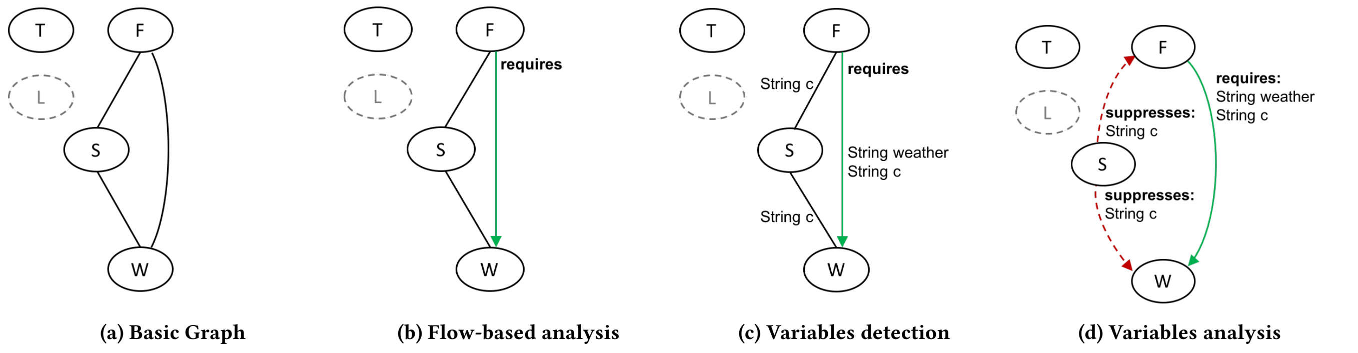

Figure 4: Creation process of the WordPress feature interaction graph, generated by VarXplorer. Solid black line: interaction. Dashed line: data flow interaction. Red arrow: suppress relationship. Green arrow: require relationship.

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). We perform the same analysis for each pair of interaction to determine the effects of features in a pair. This analysis identify all cases of suppress and require relationships between features, which may support the user to find faulty behaviors, relationships between features that should not be allowed.

Figure 4b 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), a *require* relationship. The other two interactions, (S, F) and (S, W), did not expose any explicit flow relation. Although S interact with F and W interact, they do not present any flow relationship in terms of *require* or *suppress* interactions. To further explore additional relationships between features, we complement the flow analysis with a data analysis.

Relationship based on data. In a highly configurable system, the same variable can assume different values under different configurations. Features that do not directly interact on the system 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. Conditional variables can help developers understand if a feature changes a variable value incorrectly under certain configurations, leading to a bug.

In the data analysis step, we perform two main tasks: (i) we present 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).

Context Collection. To analyze the context of data interactions, we investigate each conditional variable and its context. A *variable context* is the set of conditions that affect the value of one variable, as the rounded rectangles of Figure 3 shows. 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 3, but replace the set of presence conditions $\mathbb{PC}$ per the context of a given variable.

The WordPress example has three variables (`c, weather`, and `time`), but just two (`c` and `weather`) are considered as conditional variables. Since the variable `time` only depends on feature T (as Figure 3 shows), it is not part of any data interaction. In contrast, the context of variable `c`, for example, is composed of five different presence conditions, presenting combinations among F, S, and W. The graph in Figure 4c shows all variables involved in WordPress' interactions, (S, F), (S, W), and (F, W). Figure 4c is the same graph of Figure 4b, but now additionally shows the variables.

Analyzing Data Relations. From Figure 4c, the developer is able to inspect variables that should be overwritten in an interaction, for instance. However, that graph does not provide any information on how the features behave in relation to variables. For example, we may identify cases where a feature is suppressed by another related to a given variable. 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. Feature effect on data can be used to inspect each conditional variable and identify the effect it causes in the relationship between two features.

The analysis of feature effect per variable is analogous to the effect analysis for the entire set of presence conditions $\mathbb{PC}$ in Equation 6. The only difference is that in place of $\mathbb{PC}$, we use the context of a variable. For example, we can use the feature effect on data to investigate the variables of Figure 4c: interaction (F, W) related to variables c and *weather*; and interactions (S, F) and (S, W) related to *variable c*. To investigate (S, W) according to c, we need to analyze the effect of both S and W. First, we check the effect of feature W. Given ctx_c as the set of conditions of variable c, we analyze the effect of W according to c creating a disjunction among all W effects of each condition in ctx_c:

$$\begin{aligned}\mathbb{U}_c(W, ctx_c) &= \mathbb{U}_c(W, S) \vee \mathbb{U}_c(W, \neg S) \vee \mathbb{U}(W, W \wedge F \wedge \neg S) \vee \\ &\quad \mathbb{U}_c(W, W \wedge \neg F \wedge \neg S) \vee \mathbb{U}_c(W, \neg S \wedge \neg W) \\ &= \neg S\end{aligned}$$

As a result of the disjunction $\mathbb{U}_c(W, ctx_c)$, W has effect on variable c iff S is not selected. In other words, we may say that `SMILEY` (S) *suppresses* `WEATHER` (W) in relation to variable c. Second, we check the effect of the other feature S on each presence condition of c:

```
<system name="WordPress">
    <specification type="allow">
        <require from="Fahrenheit" to="Weather">
            <var name="weather"/>
            <var name="c"/>
        </require>
    </specification>
</system>
```

Figure 5: Example of interaction specification to WordPress.

$\mathbb{U}_c(S, ctx_c)$. The second analysis results in *true*, which means that S does not interact with another feature to affect the value of c.

Similarly, we can analyze the effect of feature FAHRENHEIT (F) on the interaction pair (S, F) related to the variable c. SMILEY also *suppresses* FAHRENHEIT (W) in relation to c. According to the trace in Figure 3, the variable c only gets the temperature (either 89.6°F or 32°C) when SMILEY is not selected. Therefore, SMILEY *suppresses* both WEATHER and FAHRENHEIT. Figure 4d shows the complete feature interaction graph for our WordPress example, for both relationship analyses provided by our approach. Figure 4d is an update of the graph in Figure 4c, now also presenting the relationships per variable (dashed directed arrows).

In summary, from the relationship analysis of WordPress based on both $\mathbb{PC}$ and data, we found that F *requires* W in $\mathbb{PC}$, which means that F is only executed when S is also selected. Based on the domain knowledge, that case represents a benign interaction between F and W. Besides, S *suppresses* F in data (variable c): when both F and S are selected, the variable c is not overwritten by F. This last 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☺". Finally, we found that S also *suppresses* W in variable c. Then, in the presence of S, W also does not overwrite c, which presents the same wrong tag to the user.

3.2 Interaction specification language

The feature-interaction graph shows all the data and control flow interactions based on a variational trace. The trace shows the differences among all configurations for a given test case (specific system input). To better inspect all the possible interactions in 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 graphs may present a large amount of interactions and conditional variables. In addition, different graphs from different test cases may share the same interactions. Although the input may be different, some pairs of feature may interact in the same way, as for example, overwriting the same variables with the same values.

Hence, we propose the *feature interaction specification language*. It helps developers to either allow or forbid interactions in a configurable system. When allowing, they may remove interactions from features that are intended to interact and present a benign behavior, which "clean" the graph and can facilitate finding interactions that represent a bug. Otherwise, an interaction flagged as forbidden in a graph can be tracked throughout all test cases executions to point out the cases when it may occur.

The interaction language is a lightweight strategy to indicate that there is an interaction among features. It does not require a formal description of the behavior of systems or features, as global and feature-based specifications do. Furthermore, those behavior specifications are usually missing. Our language is then an alternative to automatically support developers in detecting bugs. For example, they can right click on the graph to specify that an interaction is intended, which is then automatically added to the specification. In particular, specifications can be created according to three parameters: type, relationship, and target, as follows:

$$Type = \{Allow, Forbid\}$$
$$Relationship = \{Require, Suppress, Any\}$$
$$Target = \{Variable, Method, Class, Any\}$$

The *Type* defines whether the specification either allows the interaction to occur or forbids it. The former can be used to approve benign interactions that may be repeated in most test cases of a system; and the latter can be used to flag features that should not interact. Relationship and Target correspond to refinements of specifications. The *relationship* is used to refine the specification in terms of suppress and require, and combined with a *Target*, it is possible to allow or block interactions under the scope of a method, class, or variable. Allowing any interaction between two features may be dangerous. Then, the refinements are used to specify under which conditions two features present a benign or faulty interaction.

User Inspection. Our tool, VarXplorer, investigates interactions among features and helps users inspect unexpected interactions. From the feature interaction graph, a user can view how features interact and specify interactions. In the WordPress example, the developer can use the specification language to specify benign behaviors (*allow*). Guided by the visualization provided by the graph, the user can automatically allow the benign data interaction between FAHRENHEIT and WEATHER, for the variables c and weather. Figure 5 shows the *allow* interaction specification to this example. In this way, our interaction detection approach receives the specification and guarantees that the intended interaction will not be shown again in the analysis of future test cases. The interaction between FAHRENHEIT and WEATHER is only shown again in subsequent test cases if they interact in a different way, such as on different variables or through a different relationship.

Conversely, in the other two interactions of the WordPress example (SMILEY-WEATHER and SMILEY-FAHRENHEIT), one of the features in each interaction is being suppressed by the other, which may represent a bug. 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. Thus, if the same interactions reappear in other test cases, our tool will point them out as potential problematic interactions.

4 DISCUSSION AND DIRECTIONS FOR FUTURE WORK

Interaction detection. When using our approach, the user may spend less effort in finding problematic interactions. VarXplorer provides a visualization of all interactions in a configurable system and highlights feature relationships that may help users to find bugs. Although we cover all feature combinations in an execution, we use test cases to detect interactions, and, then, we may miss interactions present in uncovered inputs. We can use test-case generation in combination with our approach to cover the most representative inputs of a given system. We have applied VarXplorer on small

SPLs, such as elevator and email [13], where we were able to collect a set of interactions. In future work, we plan to investigate those systems and also real-world systems as case studies.

VarXplorer uses the variability-aware interpreter, VarexJ, to generate variational traces [21]. Thus, our approach inherits VarexJ's technical limitations. For example, it can only execute Java programs, and analyzing large systems may be computationally expensive. However, VarXplorer does not depend on VarexJ. The set of presence conditions seen during runtime can be obtained from other variability-aware execution approaches. Furthermore, we may also obtain information about feature interactions from symbolic execution [26], static analysis [18], or execution comparison [29].

Technical aspects. Our current approach focuses on pair-wise feature interactions. While higher-order interactions are less common in practice [21], they do still sometimes lead to unexpected behavior. In the future, we aim to consider such interactions, which however come with challenges for scalability and appropriate visualization that need to be solved. In general, we plan to improve our current preliminary feature interaction graph to enable easier visualization of systems with a large set of features.

Feature interaction specification language. The specification of interactions has two main benefits. Besides helping create the specifications of the system, it contributes to "clean" the graph by iteratively removing interactions that the user recognizes as desired or benign. Thus, in a graph with many interactions, the user can focus only on suspicious interactions that may represent a problem for the correct operation of the system. To make the first graph less cluttered and thus easier for the user to interpret, we could additionally consider already documented global specifications and feature specifications to filter the interactions accordingly.

5 CONCLUSIONS

In highly configurable systems, features may interact unexpectedly, producing faulty behavior. We propose VarXplorer, an incremental and interactive lightweight process to detect problematic interactions dynamically. From a variational execution, we gather all the variability context (control-flow paths and shared data) in which each instruction is executed to create a feature interaction graph. VarXplorer uses this information as input to identify how the features are related to each other and helps users to inspect unintended interactions. While analyzing the graph, users may indicate interactions that present a benign behavior and also mark others as forbidden, through the feature interaction specification language.

ACKNOWLEDGMENTS

Soares and Almeida have been supported by the CNPq and CAPES. Nadi's work has been supported by the NSERC. Kaestner's and Meinicke's work has been supported by NSF award 1552944.

REFERENCES

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

[2] 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.

[3] 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.

[4] S. Apel, A. von Rhein, T. Thüm, and C. Kästner. Feature-Interaction Detection Based on Feature-Based Specifications. *ComNet*, 57(12):2399–2409, 2013.

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

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

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

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

[13] R. J. Hall. Fundamental Nonmodularity in Electronic Mail. *ASE*, 12(1):41–79, 2005.

[14] 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.

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

[16] 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.

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

[18] M. Lillack, C. Kästner, and E. Bodden. Tracking Load-Time Configuration Options. In *ASE*, pages 445–456. ACM, 2014.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

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

[25] 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.

[26] 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.

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

[28] 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.

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

[30] 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.

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

[32] 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.

[33] 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.