VarXplorer: Reasoning About Feature Interactions

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

ABSTRACT

Feature interactions occur when a feature behavior is influenced by the presence of another feature(s). Typically, interactions may lead to faults that are not easily identified from the analysis of each feature separately, especially when feature specifications are missing. In this paper, we propose VarXplorer, an iterative approach that supports developers to detect internal interactions on control and data flow of configurable systems, by means of feature-interaction graphs and an interaction specification language.

CCS CONCEPTS

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

KEYWORDS

Highly Configurable Systems, Feature Interaction, Variability-Aware Execution

1 INTRODUCTION

Highly configurable systems allow to customize software to individual needs by composition of features (aka. configuration options) [6]. A *feature interaction* is a common problem in highly configurable systems. Usually, an interaction among two or more features may result in an unexpected behavior that cannot be deduced from the analysis of each feature in isolation [1, 3]. In addtion, it is hard to early specify all possible interactions in configurable systems, mainly due to the unpredictable nature of interactions [1]. Writing specifications requires human effort, and testing each feature combination may be unfeasible as the number of configurations and feature interactions grows exponentially to the number of features [4].

We propose an iterative and interactive process to support developers to distinguish intended interactions from interactions that may lead to bugs, without the need for upfront specifications. Our tool *VarXplorer* [10] inspects interactions from the variational traces generated by VarexJ [7], a variational interpreter to simultaneously execute all system configurations. A variational trace contains the differences among all configurations on control and data flow. From a configurable system, we analyze interactions as they are detected and incrementally classify them as benign (they do not cause any problem to the system) or problematic (undesired interactions that may cause faulty or damaging system behavior, such as crashes).

```
1  boolean STATISTICS, SMILEY, WEATHER, FAHRENHEIT, SECURE_LOGIN;
2    void createHtml() {
3        String c = wpGetContent();
4        if (SMILEY)
5            c = c.replace(":]", getSmiley(":]"));
6        if (WEATHER) {
7            String weather = getWeather();
8            c = c.replace("[:weather:]", weather);  }
9        if (STATISTICS) {
10           int time = getCurrentTime();
11           printStatistics(time);  }  }
12   String getWeather() {
13       float temparature = 30;
14       if (FAHRENHEIT)
15           return (temperature * 1.8 + 32) + "°F";
16       return temperature + "°C";   }
```

Figure 1: Feat. interactions modeled after WorldPress [7].

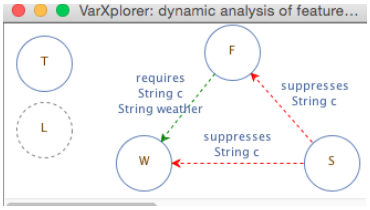

Figure 2: Interaction graph for the WordPress by VarXplorer. Red arrow: suppress relationship. Green arrow: require.

In Figure 1, we illustrate both benign and problematic interactions. In the code excerpt modeled after WordPress, the features *weather* and *fahrenheit* interact intentionally to display the weather information. However, the feature *smiley* may interact with *weather* in an undesired way. When they are together, instead of replacing the "[:weather:]" tag by the current temperature (e.g., 70°F), it is rewritten to "[:weather☺".

In this paper, we present our ongoing research to support developers in understanding interactions, firstly introduced on [10]. *VarXplorer* investigates pairwise interactions and presents the relationships among features through *feature-interaction graphs*. From the graph, developers may indicate intended behavior and also select interactions as forbidden through the feature-interaction specification language. This process enables developers to refine the graph and remove interactions that do not represent a bug.

2 STATE OF THE ART

Specifying all feature interactions may not scale considering the large number of possible configurations. An alternative is to describe the behavior of a feature in isolation without any explicit reference to other features, known as a *feature-based specification* [11]. Conversely, a strategy to reduce the effort of creating specifications for each feature is to define *global specifications*. Typically, they are

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05…$15.00
https://doi.org/10.1145/3183440.3190329

common properties to all configurations based on general requirements (e.g. tests in which the systems should not crash) [11]. Many studies are based on global specifications to deal with interactions, such as combinatorial interaction testing [9], model checking [2], and variational execution [7].

Although there is a lot of work to detect faults caused by feature interactions, it remains challenging to identify accidental and unexpected interactions. Since many of them cause faulty behavior without necessarily lead the system to a crash, it is hard to identify incorrect outputs. In our work, we approach the challenge of identifying interactions through the analysis of relationships between features without upfront existing specifications.

3 ITERATIVE ANALYSIS OF INTERACTIONS

In our approach, we detect feature interactions by comparing the executions of all system configurations, in which the interactions are manifested in the differences in data and in the control flow that depend on more than one feature. We use the variational interpreter VarexJ [7] to efficiently compare the executions of *all* configurations and to create the variational trace. It contains the presence conditions that add or change any functionality during the execution. *Presence conditions* ($\mathbb{PC}$) are propositional expressions over options that determine when a specific code artifact is executed [8].

In this way, we execute test cases to identify which features interact in a configurable system. From the variational trace of a system, we are able to detect interactions, their relationships and data context (variables), besides presenting them as a *feature interaction graph*: a representation of all (pairwise) interactions among features. The graph provides a higher level of abstraction to allow developers to understand and detect problematic interactions.

Feature-interaction relationships represent the relation that a feature may have over others. We identify two main types of relationship: when a given feature *suppresses* or *requires* another feature. Relationships are often accompanied by variables that depend on the features involved in the interaction. Problematic interactions may be associated with variables that have different values to each feature combination.

Interaction graphs created from real systems may present a large amount of interactions. To facilitate finding bugs and to "clean" the graph, we propose an incremental process of interaction detection supported through a *feature interaction specification language*. On the one hand, it automatically refines the graph based on the user manual inspection to remove interactions that present a benign behavior. On the other hand, it allows developers to flag interactions as forbidden, which will be tracked throughout all test cases executions to point out the cases when it may occur.

Interaction specifications describe that there exists an interaction between two features. For instance, developers can right click on the graph to specify that an interaction is intended, which is then automatically added to the specification. Unlike global and feature-based specifications, it does not require a formal description of the behavior of either systems or features.

Interaction detection [10]. In this process, we analyze all pairs of features that interact in a system. The detection receives as input a variational trace created from executing a test case, and delivers the interaction graph presenting all pairwise interactions. Basically, the interaction detection consists of: (i) *pairwise detection* and (ii) *relationship analysis*.

First, we create a basic graph composed only of the pairs of features that interact. Then, we perform the relationship analysis and investigate each pair to determine the effect one feature has on the other (i.e., either suppress or require other features) by means of the unique existential quantification [8]. Our approach also identifies variables and relationships exclusive to variables. For example, a feature f may not present an overall suppression on the feature g, but f may suppress g in relation to a given variable. This second step refines the graph with the relationships between features, including the underlying variables they affect, to produce the complete interaction graph.

Figure 2 shows the complete graph for our WordPress example. From the relationship analysis based on $\mathbb{PC}$ and variables, we found that `FAHRENHEIT` (F) *requires* `WEATHER` (W), which means that F is only executed when W is also selected. Based on the domain knowledge, this is a benign interaction between F and W. Besides, `SMILEY` (S) *suppresses* F in data (variable c): when both F and S are selected, the variable c is not overwritten by F. This case may be a bug because instead of seeing the current temperature, users see the tag "[:weather☺". Finally, we found that S also *suppresses* W in variable c. In the presence of S, W does not overwrite c, which presents the same wrong tag to the user.

From the graph provided by VarXplorer, developers can view how features interact, besides specify interactions. In the WordPress example, the developer can use the specification language to specify benign behaviors. For example, they can right click on the graph to allow the benign data interaction between F and W, for the variables `c` and `weather`. However, the other two interactions of the WordPress example (S-W and S-F) may indicate bugs, once one of the features in each interaction is being suppressed by the other. The user may want to fix the problem directly in the code and can also mark those interactions as suspicious in the graph.

4 CONCLUSIONS AND FUTURE WORK

VarXplorer is a lightweight strategy to identify feature interactions without any previous system or feature specification. Based on a configurable system, our approach investigates all pairwise interactions between features to identify bugs from features relationships and interaction-dependent variables. Additionally, although higher-order interactions are less common in practice [7], they may result in an unintended behavior too. To proper present those interactions in the future, we aim to improve the graph visualization to consider scalability issues. Since we use test cases, we may miss interactions present in uncovered inputs. Then, we can use test case generation to cover the most representative inputs of a system. We have been working on an evaluation design to apply VarXplorer on small configurable systems (e.g., elevator and email [5]) and also on real-world systems in the future.

REFERENCES

[1] S. Apel, S. Kolesnikov, N. Siegmund, C. Kästner, and B. Garvin. Exploring Feature Interactions in the Wild: The New Feature-interaction Challenge. In *FOSD*, 2013.
[2] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Detection of Feature Interactions Using Feature-Aware Verification. In *ASE*. IEEE, 2011.
[3] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1), 2003.

[4] 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.
[5] R. J. Hall. Fundamental Nonmodularity in Electronic Mail. *ASE*, 12(1), 2005.
[6] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990.
[7] 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.
[8] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. Mining configuration constraints: Static analyses and empirical results. ICSE, pages 140–151. ACM, 2014.
[9] C. Nie and H. Leung. A Survey of Combinatorial Testing. *CSUR*, 43(2), 2011.
[10] L. R Soares, J. Meinicke, S. Nadi, C. Kästner, and E. S. de Almeida. Varxplorer: Lightweight process for dynamic analysis of feature interactions. In *VaMoS*, 2018.
[11] 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), 2014.