

Problem-Oriented Feature Interaction Detection in Software Product Lines

Andreas CLASSEN^a

^a *acs@info.fundp.ac.be*

*Computer Science Department, University of Namur
5000 Namur, Belgium*

Abstract.

Feature interaction detection in the context of systems that are highly integrated into their environment, such as embedded or software-intensive systems, is different from classical feature interaction detection. The physical environment may be “*the source of additional interactions*” as interactions may be caused by, and occur in, the system’s physical environment.

We thus propose an approach for automated detection of feature interactions in the environment which is based on feature diagrams capturing variability, problem diagrams describing the system in its context and event calculus formulae allowing for automated reasoning. Feasibility of the approach is demonstrated through a proof-of-concept tool implementation and an in-depth illustration.

Keywords. Formal Verification, Feature Interactions, Software Product Lines, Requirements Engineering

1. Introduction

Systems that are highly integrated into their environment are a focus of feature interaction detection research [5]. As Metzger points out, feature interaction detection in this context is different from classical feature interaction detection, because the physical environment may be “*the source of additional interactions*” [5]. This was already illustrated by Calder *et al.* [1] on a control system for automobiles. Another interesting example is the case of smart home control systems [4,5]. Interactions can actually pass through shared environment variables. The heating service of such a system, for instance, can start a fan, causing false alarms in the security service [4]. These features (or services) are often developed independently, either by different service providers or by different teams in a company offering a large product line [7] to their customers.

Eventually, this means that feature interaction detection in this context has to take into account two different perspectives. On the one hand, the *software product lines* perspective is needed in order to determine the actual systems that have to be verified. The *system description* perspective, on the other hand, has to be considered because it provides the descriptions against which interaction-related properties will eventually be checked. In the context of our approach, we further structure this last perspective by adopting the Zave and Jackson requirements engineering reference model [9] which

divides system descriptions into three categories: requirements (R), domain assumptions (W) and specifications (S).

2. Feature Interaction Detection

Feature interactions in embedded and environment-integrated systems can generally be seen as causal chains initiated by one service that interfere with other services, causing undesirable behaviour of the system. Depending on the system's purpose and on its environment, these causal chains can be very complex, and it would probably be impossible to verify them by hand. We thus suggest an automatable approach based on formal verification. Basically, we mostly follow current off-line approaches by doing model-checking on feature descriptions in order to detect interactions.

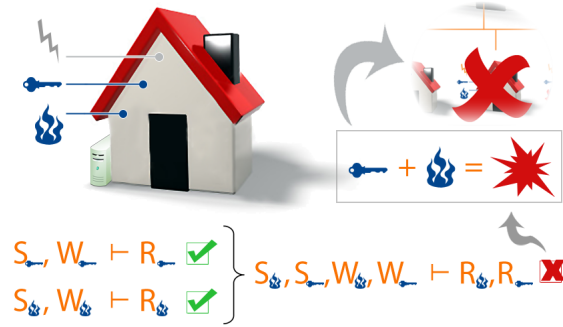


Figure 1. An illustration of the feature interaction detection procedure, scetching the detection of the interaction between security and heating service in a smart home.

The basic idea of our approach is to verify each product of a product line, based on the first proof obligation of the Zave and Jackson framework [9], equation 1, which serves as correctness proof for a system (including systems consisting of a single feature):

$$S, W \vdash R \quad (1)$$

This proof obligation expresses the fact that the requirements have to be satisfied if both, the specification and the assumptions about the world, are satisfied. Now given a set of features $p = f_1..f_n$, expressed as S_i, W_i, R_i for $i = 1..n$ and $n \geq 2$, we say that features $f_1..f_n$ interact if the following holds:

- they satisfy their individual requirements in isolation,

$$\forall f_i \in p . S_i, W_i \vdash R_i \quad (2)$$

- they do not satisfy the conjunction of these requirements when put together,

$$\bigwedge_{i=1}^n S_i, \bigwedge_{i=1}^n W_i \not\vdash \bigwedge_{i=1}^n R_i \quad (3)$$

- and removing any feature from p results in a set of features that do not interact.

$$\forall f_k \in p. \bigwedge_{i \in \{1..k-1, k+1..n\}} S_i, \bigwedge_{i \in \{1..k-1, k+1..n\}} W_i \vdash \bigwedge_{i \in \{1..k-1, k+1..n\}}^n R_i \quad (4)$$

A feature interaction in a system $s = \{f_1..f_q\}$ is then any set $p \subseteq s$ such that its features interact.¹ In addition to equation 1, there are other proof obligations that need to be verified, namely to make sure that equation 1 is not trivially verified. The approach can then be described by four different algorithms covering the various verifications, the detail of which is presented in [2]. Fig. 1 illustrates this by showing how the previously mentioned interaction between security and heating service in a smart home is detected. Both features are first verified in isolation, and then in combination. The first verification is assumed to pass, the second assumed to fail. This indicates that an interaction is present, which will then probably lead to changes in the feature diagram or in the system descriptions in order to avoid or correct the interaction.

The algorithms defining our approach build on the two perspectives introduced in Section 1. Feature diagrams are used to model and describe the variability, and eventually to determine the different systems belonging to the product line, i.e. the valid configurations of the feature diagram [8]. In turn, each feature of this feature diagram is mapped to a problem diagram [3] providing its three constituent descriptions: R , W and S .

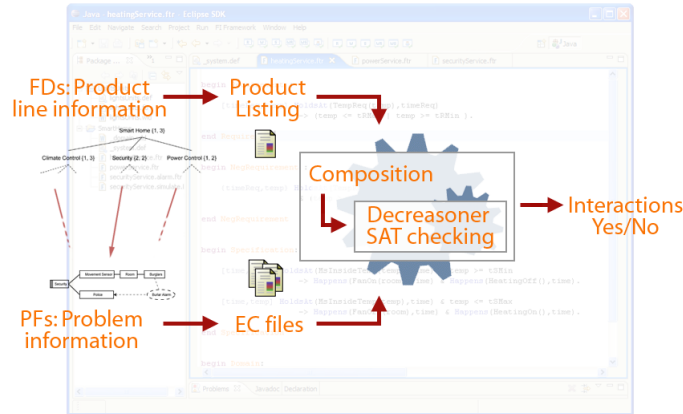


Figure 2. FIFramework screenshot and workflow.

3. Automation

The procedure sketched above can be largely automated, provided an automatable formalism for expressing the R , W and S descriptions is chosen. In our approach we chose the event calculus, which is based on first-order predicate logic, and allows intuitive expression of causal relations in the real world [6]. Furthermore, the event calculus comes with several implementations of which we chose Mueller’s *discrete event calculus rea-*

¹Note that in classical logic, the satisfaction relation would be monotonic and systems satisfying this definition impossible. Depending on the formalism used, however, the relation may be not monotonic, hence its interest.

soner (Decreasoner) [6], which basically transforms a set of event calculus formulae into a SAT problem, passes it to a SAT-solver and interprets the results.

On top of Decreasoner we built an Eclipse-plugin, *FIFramework*,² which is effectively a proof-of-concept implementation of the suggested approach. Fig. 2 gives a high level overview of the workflow and of what the tool does. Essentially, the user specifies his product line in terms of features, each feature being represented by a file containing event calculus formulae. These files as well as the list of products serve as an input for *FIFramework*, which composes the descriptions relevant to the proof, sends them to Decreasoner and interprets the result. The tool thus automates all verification steps of the approach.

4. Conclusion

The high degree of integration in the environment, as in the case of embedded control devices or software-intensive systems, leads to feature interactions that are not restricted to the software, but may be caused by, and occur in, the physical environment of the system. We thus propose an approach for automated detection of feature interactions in the environment which is based on feature diagrams capturing variability, problem diagrams describing the system in its context and event calculus formulae allowing for automated reasoning. A proof-of-concept tool implementation for the Eclipse platform demonstrates its feasibility. Furthermore, an in-depth illustration, based on the smart home example case, is provided in [2].

Benefits of this approach to feature interaction detection are (i) its foundations in a well accepted requirements engineering reference model, which allows the approach to be very general; (ii) the ability to detect interactions exterior to the machine and (iii) an approach that not only focuses on single-system development, but also covers the case of product line engineering.

References

- [1] Calder, M., Kolberg, M., Magill, E.H., Reiff-Marganiec, S.: Feature interaction: a critical review and considered forecast. *Computer Networks* **41**(1) (2003) 115–141
- [2] Classen, A.: Problem-oriented modelling and verification of software product lines. Master's thesis, University of Namur, Belgium (June 2007)
- [3] Jackson, M.A.: Problem frames: analyzing and structuring software development problems. Addison-Wesley Longman Publishing, Boston, MA, USA (2001)
- [4] Kolberg, M., Magill, E.H., Wilson, M.: Compatibility issues between services supporting networked appliances. *IEEE Comm. Magazine* **41**(11) (2003) 136–147
- [5] Metzger, A.: Feature interactions in embedded control systems. *Computer Networks* **45** (2004) 625–644
- [6] Mueller, E.T.: Commonsense Reasoning. Morgan Kaufmann (2006)
- [7] Pohl, K., Bockle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer (July 2005)
- [8] Schobbens, P.Y., Heymans, P., Trigaux, J.C., Bontemps, Y.: Feature Diagrams: A Survey and A Formal Semantics. In: Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06), Minneapolis, Minnesota, USA (September 2006) 139–148
- [9] Zave, P., Jackson, M.A.: Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology* **6**(1) (1997) 1–30

²Available online at www.classen.be/references/mscthesis.