Report from Dagstuhl Seminar 14281

Feature Interactions: The Next Generation

Edited by

Sven Apel[1], Joanne M. Atlee[2], Luciano Baresi[3], and Pamela Zave[4]

1 **University of Passau, DE,** apel@uni-passau.de
2 **University of Waterloo, CA,** jmatlee@uwaterloo.ca
3 **Politecnico di Milano, IT,** luciano.baresi@polimi.it
4 **AT&T Labs Research, US,** pamela@research.att.com

Abstract

The feature-interaction problem is a major threat to modularity and impairs compositional development and reasoning. A feature interaction occurs when the behavior of one feature is affected by the presence of another feature; often it cannot be deduced easily from the behaviors of the individual features involved. The feature-interaction problem became a crisis in the telecommunications industry in the late 1980s, and researchers responded with formalisms that enable automatic detection of feature interactions, architectures that avoid classes of interactions, and techniques for resolving interactions at run-time. While this pioneering work was foundational and very successful, it is limited in the sense that it is based on assumptions that hold only for telecommunication systems. In the meantime, different notions of feature interactions have emerged in different communities, including Internet applications, service systems, adaptive systems, automotive systems, software product lines, requirements engineering, and computational biology. So, feature interactions are a much more general concept than investigated in the past in the context of telecommunication systems, but a classification, comparison, and generalization of the multitude of different views is missing. The feature-interaction problem is still of pivotal importance in various industrial applications, and the Dagstuhl seminar "Feature Interactions: The Next Generation" gathered researchers and practitioners from different areas of computer science and other disciplines with the goal to compare, discuss, and consolidate their views, experience, and domain-specific solutions to the feature-interaction problem.

Seminar July 6–11, 2014 – http://www.dagstuhl.de/14281
1998 ACM Subject Classification D.2.1 Requirements/Specifications, D.2.4 Software/Program Verification, D.2.10 Design, D.2.11 Software Architectures, D.2.13 Reusable Software
Keywords and phrases Feature interactions, feature-interaction problem, feature orientation, product lines, modularity, composition
Digital Object Identifier 10.4230/DagRep.4.7.1
Edited in cooperation with Sergiy Kolesnikov

Feature Interactions: The Next Generation, *Dagstuhl Reports*, Vol. 4, Issue 7, pp. 1–24
Editors: Sven Apel, Joanne M. Atlee, Luciano Baresi, and Pamela Zave

Dagstuhl Reports
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Executive Summary

Sven Apel
Joanne M. Atlee
Luciano Baresi
Pamela Zave

Overview and Motivation

A major goal of software and systems engineering is to construct systems from reusable parts, which we call *features* (end-user–visible units of behavior or increments in system functionality). Such a compositional approach can decrease time to market, improve product quality, and diversify the product portfolio. However, the success of a compositional approach depends on the modularity of the reusable parts. The quest for modularity has a long tradition in software and systems engineering, programming languages research, and even in newer fields such as synthetic biology.

In the early days of software and systems engineering, the feature-interaction problem was identified (and coined) as a major threat to modularity [8, 31, 25]. A feature interaction occurs when the behavior of one feature is affected by the presence of another feature. Often the interaction cannot be deduced easily from the intended behaviors of the individual features involved. A canonical example is the inadvertent interaction between the call-forwarding and call-waiting features of a telephony system [8]: If both features are active, the system can reach an undefined possibly unsafe state when it receives a call on a busy line, because it is not specified whether the call should be suspended or forwarded. Alternatively, a feature interaction can be *planned*: for example, advanced cruise-control features are designed to interact with and extend basic cruise control.

To be safe, software developers must analyze the consequences of all possible feature interactions, in order to find and fix the undesired interactions. The *feature-interaction problem* is that the number of potential interactions to consider is exponential in the number of features. As a result, software developers find that their work in developing new features is dominated by the tasks to detect, analyze, and verify interactions.

The feature-interaction problem is deeply rooted in the fact that the world is often not compositional [25, 20]. That is, a feature is not an island. It communicates and cooperates with other features and the environment, so it cannot be completely isolated. Insights from complex-systems research suggest that feature interactions are a form of emergent behavior that is inherent to any system that consists of many, mutually interacting parts. So, emergent system behavior – which is not deducible from the individual parts of a system – can be observed in many situations including in quantum systems (e. g., superconductivity), biological systems (e. g., swarm intelligence), and economical systems (e. g., trading market crashes). The challenge is to foster and manage *desired* interactions and to detect, resolve, and even avoid *undesired* feature interactions – in a scalable manner.

The feature-interaction problem became a crisis in the telecommunications industry in the late 1980s [5]. To handle complexity, there was the strong desire to *compose* systems from independently developed features, but there was no means to detect, express, and reason about feature interactions. Researchers responded with formalisms that enable automatic detection of feature interactions [4, 7, 15, 14, 21, 26], architectures that avoid classes of interactions [17, 29, 18, 28, 31], and techniques for resolving interactions at run-

time [16, 27]. Architectural solutions have been the most successful because they impose general coordination strategies (i. e., serial execution) that apply to all features that are 'plugged' into the architecture, thereby, addressing the scalability issue at the heart of the feature-interaction problem. In coordination-based approaches, such as BIP [2, 3] or Composition Patterns [10], the interactions among a set of features are specified explicitly and can be specialized for subsets of features.

While the pioneering work on the feature-interaction problem in telecommunication systems was foundational and very successful [8], it is limited in the sense that it is based on assumptions that hold for telecommunication systems, but that do not hold in other domains. For example, architecture-based approaches take advantage of the fact that communication takes place over a mostly serial connection between communicating parties – which is not the case in systems made up of parallel components (e. g., service systems, automotive software) or software product lines (e. g., features implemented via conditional compilation such as the Linux kernel). Specifying interactions explicitly is not a general solution either. When facing systems composed of thousands of features, attempting to identify and model a possibly exponential number of feature interactions is elusive. Furthermore, the highly dynamic nature of feature (or service) composition in self-adaptive systems, dynamic product lines, cloud computing, and systems of systems imposes a new class of challenges to solving the feature-interaction problem [24, 9, 1].

So, it is not surprising that different notions of feature interactions have emerged in different communities [6]. Instances of the feature-interaction problem have been observed and addressed in Internet applications [11], service systems [30], automotive systems [12], software product lines [19], requirements engineering [23], computational biology [13], and in many other fields outside of computer science. While all instances of the problem are rooted in the nature of modularity and compositionality [25, 20], the individual views, interpretations, and possible solutions differ considerably. For example, the view on feature interactions taken in program synthesis [22] differs significantly from the view in automotive systems engineering [12]: there are structural vs. behaviour views, static vs. dynamic views, sequential vs. parallel views, functional vs. non-functional, coordinated vs. emergent-behaviour views, and so on. It turns out that feature interactions are a much more general concept than investigated in the past in the context of telecommunication systems, but a classification, comparison, and generalization of the multitude of different views is missing.

The feature-interaction problem is still of pivotal importance in various industrial applications, but, despite significant efforts, it is far from being solved. The underlying hypothesis of organizing a Dagstuhl seminar on this topic was that the time is ripe to gather researchers and practitioners from different areas of computer science and other disciplines to compare, discuss, and consolidate their views, experience, and domain-specific solutions to the feature-interaction problem. To make progress, scientific discourse on the feature-interaction problem must be based on a broader foundation to be able to join forces of different communities. Can other domains learn from the success of domain-specific solutions for telecommunication systems? Are there key principles, patterns, and strategies to represent, identify, manage, and resolve feature interactions that are domain-independent, that are valid and useful across domains? Or, should we strive for domain-specific solutions that are only loosely related to solutions from other domains? Can we develop a unified terminological and conceptual framework for feature-interaction research? Is that even possible or meaningful, given that interactions in telecommunication systems and emergent behavior and phase transitions in swarm systems are, although related, quite different views?

Goals of the Seminar and Further Activities

It is our goal and firm belief that the feature-interaction problem needs to be viewed from a broader perspective. While feature interactions are still a major challenge in software and systems engineering, both in academia and industry, research on the feature-interaction problem has diversified and diverged in the last decade. Researchers working on similar problems, but in different contexts, are largely disconnected and unaware of related work. A major goal of the seminar was to (re)launch a sustained research community that embraces researchers and practitioners from different fields within and outside computer science. We firmly believe that we reached this goal with our seminar. In particular, a subset of the participants is going to organize a follow-up seminar that directly builds on this seminar's results. The next major milestone will be – now as we gained a better understanding of the similarities and differences between the different notions of feature interactions – to establish a catalog on feature-interaction patterns and solutions thereof. The idea for this pattern catalog arose from the final panel session of the seminar. It is inspired by work on patterns in architecture (of buildings). Such a catalog will be the necessary basis for further research on leveraging patterns for detecting, managing, and resolving feature interactions in different kinds of systems.

References

1 L. Baresi, S. Guinea, and L. Pasquale. Service-oriented dynamic software product lines. *IEEE Computer*, 45(10):42–48, 2012.
2 A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *Proc. of the Int'l Conf. on Software Engineering and Formal Methods (SEFM)*, pages 3–12. IEEE, 2006.
3 S. Bliudze and J. Sifakis. The algebra of connectors – Structuring interaction in BIP. *IEEE Transactions on Computers*, 57(10):1315–1330, 2008.
4 J. Blom, B. Jonsson, and L. Kempe. Using temporal logic for modular specification of telephone services. In *Feature Interactions in Telecommunications Systems*, pages 197–216. IOS Press, 1994.
5 T. Bowen, F. Dworack, C. Chow, N. Griffeth, G. Herman, and Y.-J. Lin. The feature interaction problem in telecommunications systems. In *Proc. of the Int'l Conf. on Software Engineering for Telecommunication Switching Systems (SETSS)*, pages 59–62. IEEE, 1989.
6 G. Bruns. Foundations for features. In *Feature Interactions in Telecommunications and Software Systems VIII*, pages 3–11. IOS Press, 2005.
7 G. Bruns, P. Mataga, and I. Sutherland. Features as service transformers. In *Feature Interactions in Telecommunications Systems V*, pages 85–97. IOS Press, 1998.
8 M. Calder, M. Kolberg, E. Magill, and S. Reiff-Marganiec. Feature interaction: A critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003.
9 B. Cheng, R de Lemos, H. Giese, P. Inverardi, J. Magee, et al. Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for Self-Adaptive Systems*, LNCS 5525, pages 1–26. Springer, 2009.
10 S. Clarke and R. Walker. Composition patterns: An approach to designing reusable aspects. In *Proc. of the Int'l Conf. on Software Engineering (ICSE)*, pages 5–14. IEEE, 2001.
11 R. Crespo, M. Carvalho, and L. Logrippo. Distributed resolution of feature interactions for Internet applications. *Computer Networks*, 51(2):382–397, 2007.
12 A. Dominguez. *Detection of Feature Interactions in Automotive Active Safety Features*. PhD thesis, School of Computer Science, University of Waterloo, 2012.
13 R. Donaldson and M. Calder. Modular modelling of signalling pathways and their cross-talk. *Theoretical Computer Science*, 456(0):30–50, 2012.

14 A. Felty and K. Namjoshi. Feature specification and automated conflict detection. *ACM Transactions on Software Engineering and Methodology*, 12(1):3–27, 2003.

15 M. Frappier, A. Mili, and J. Desharnais. Defining and detecting feature interactions. In *Proc. of the IFIP TC 2 WG 2.1 Int'l Workshop on Algorithmic Languages and Calculi*, pages 212–239. Chapman & Hall, Ltd., 1997.

16 N. Griffeth and H. Velthuijsen. The negotiating agents approach to runtime feature interaction resolution. In *Feature Interactions in Telecommunications Systems*, pages 217–235. IOS Press, 1994.

17 J. Hay and J. Atlee. Composing features and resolving interactions. In *Proc. of the ACM SIGSOFT Symp. on Foundations of Software Engineering (FSE)*, pages 110–119. ACM, 2000.

18 M. Jackson and P. Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering (TSE)*, 24(10):831–847, 1998.

19 P. Jayaraman, J. Whittle, A. Elkhodary, and H. Gomaa. Model composition in product lines and feature interaction detection using critical pair analysis. In *Proc. of the Int'l Conf. on Model Driven Engineering Languages and Systems (MoDELS)*, LNCS 4735, pages 151–165. Springer, 2007.

20 C. Kästner, S. Apel, and K. Ostermann. The road to feature modularity? In *Proc. of the Int'l Workshop on Feature-Oriented Software Development (FOSD)*, pages 5:1–5:8. ACM, 2011.

21 F. Lin and Y.-J. Lin. A building block approach to detecting and resolving feature interactions. In *Feature Interactions in Telecommunications Systems*, pages 86–119. IOS Press, 1994.

22 J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *Proc. of the Int'l Conf. on Software Engineering*, pages 112–121. ACM, 2006.

23 A. Nhlabatsi, R. Laney, and B. Nuseibeh. Feature interaction: The security threat from within software systems. *Progress in Informatics*, (5):75–89, 2008.

24 L. Northrop, P. Feiler, R. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, D. Schmidt, K. Sullivan, and K. Wallnau. Ultra-large-scale systems – The software challenge of the future. Technical report, Software Engineering Institute, Carnegie Mellon University, 2006.

25 K. Ostermann, P. Giarrusso, C. Kästner, and T. Rendel. Revisiting information hiding: Reflections on classical and nonclassical modularity. In *Proc. of the Europ. Conf. on Object-Oriented Programming (ECOOP)*, LNCS 6813, pages 155–178, 2011.

26 K. Pomakis and J. Atlee. Reachability analysis of feature interactions: A progress report. In *Proc. of the Int'l Symp. on Software Testing and Analysis (ISSTA)*, pages 216–223. ACM, 1996.

27 S. Tsang and E. Magill. Learning to detect and avoid run-time feature interactions in intelligent networks. *IEEE Transactions on Software Engineering (TSE)*, 24(10):818–830, 1998.

28 G. Utas. A pattern language of feature interaction. In *Feature Interactions in Telecommunications Systems V*, pages 98–114. IOS Press, 1998.

29 R. van der Linden. Using an architecture to help beat feature interaction. In *Feature Interactions in Telecommunications Systems*, pages 24–35. IOS Press, 1994.

30 M. Weiss, B. Esfandiari, and Y. Luo. Towards a classification of web service feature interactions. *Computer Networks*, 51(2):359–381, 2007.

31 Pamela Zave. Modularity in Distributed Feature Composition. In *Software Requirements and Design: The Work of Michael Jackson*, pages 267–290. Good Friends Publishing, 2010.

2 Table of Contents

3 Perspective Talks

3.1 Toward User-Centric Feature Composition for the Internet of Things

Pamela Zave (AT&T Labs Research, US)

Main reference P. Zave, E. Cheung, S. Yarosh, "Toward User-Centric Feature Composition for the Internet of Things," unpublished manuscript.
URL http://www2.research.att.com/~pamela/userFtrComp.pdf

Many user studies of home automation, as the most familiar representative of the Internet of Things, have shown the difficulty of developing technology that users understand and like. It helps to state requirements as largely-independent features, but features are not truly independent, so this incurs the cost of managing and explaining feature interactions. We propose to compose features at runtime, resolving their interactions by means of priority. Although the basic idea is simple, its details must be designed to make users comfortable by balancing manual and automatic control. On the technical side, its details must be designed to allow meaningful separation of features and maximum generality. As evidence that our composition mechanism achieves its goals, we present three substantive examples of home automation, and the results of a user study to investigate comprehension of feature interactions. A survey of related work shows that this proposal occupies a sensible place in a design space whose dimensions include actuator type, detection versus resolution strategies, and modularity.

3.2 The Feature Interaction Problem in a Federated Communications-Enabled Collaboration Platform

Mario Kolberg (University of Stirling, Stirling, Scotland, GB)

Joint work of Kolberg, M.; Buford, J. F.; Dhara, K.; Wu, X.; Krishnaswamy, V.
Main reference M. Kolberg, J. F. Buford, K. Dhara, X. Wu, V. Krishnaswamy, "Feature Interaction in a Federated Communications-Enabled Collaboration Platform," Computer Networks Journal, 57(12):2410–2428, 2013.
URL http://dx.doi.org/10.1016/j.comnet.2013.02.023

For enterprise use there is a need to integrate various collaboration tools such as email, instant messages, wikis, blogs, web conferences, and shared documents, as well as link with existing intelligent communication systems to support long-term collaborations in a variety of ways. By the very nature of such systems they include a large number of independently developed features and services and thus provide a strong potential for feature interactions. This paper presents novel work on feature interaction analysis in collaboration environments and presents new types of interactions found in this space.

In this talk ConnectedSpaces is used as a basis to carry out a detailed analysis of feature interaction problems in collaboration environments. ConnectedSpaces is a new model for federated collaboration environments. Like a number of existing systems, ConnectedSpaces uses a collaboration space as the basic construct. ConnectedSpaces enables the user to work directly in the client application of their choice, including MS Outlook, Internet Explorer and Skype.

This talk presents distinctive characteristics of ConnectedSpaces, including views, spaces as communication endpoints, space persistence and structuring, and embedded objects. Using these features, new types of feature interactions for collaboration platforms are categorized and analyzed. This work is novel as it is the first investigation into feature interactions with collaboration platforms. The talk will also outline potential approaches to handle such interactions. We advocate a runtime feature interaction technique which can cope with features being provided by different organizations.

3.3 Feature Interactions in Software Systems: An Implementation Perspective

Christian Kästner (Carnegie Mellon University, US), Sven Apel (University of Passau, DE)

Joint work of Apel, Sven; Kästner, Christian; Nguyen, Hung Viet; Nguyen, Tien N.; Kolesnikov, Sergiy; Siegmund, Norbert
Main reference S. Apel, S. Kolesnikov, N. Siegmund, C. Kästner, B. Garvin, "Exploring Feature Interactions in the Wild: The New Feature-Interaction Challenge," in Proc. of the 5th Int'l Workshop on Feature-Oriented Software Development (FOSD'13), pp. 1–8, ACM, 2013.
URL http://dx.doi.org/10.1145/2528265.2528267

In this talk, we have discussed feature interactions in software systems from an implementation perspective. Feature interactions are a common problem in configurable systems, among implementations of configuration options. Key differences to classic research on feature interactions [3] are: We operate under a closed-world assumption where the implementation of all features are known; we typically focus on a single, non-distributed process; variability is induced by configuration options with non- trivial dependencies, implemented through various implementation mechanisms, from modules to conditional compilation. Feature interactions manifest in different forms, four of which we discussed.

First, interaction bugs occur when a system exhibits a bug if and only if multiple options are selected in specific combinations [5]. Typically options work well in isolation, but expose a bug if combined. The community has developed close-world, whole-product-line techniques, which we call variability-aware or family-based analyses [1, 7], that can identify certain classes of bugs. Compared to standard analysis techniques, variability-aware analyses cover the whole configuration space and allow statements about the configurable system in its entirety (and not only about individual configurations). This way, several bugs have been found that are only triggered by specific configurations settings. Empirical studies on the reported bugs have revealed common interaction patterns.

Second, performance interactions have been defined as unexpected performance behaviors when combining multiple configuration options [6]. Assuming that the influence of each option on the system's performance (for a given benchmark) can be isolated, a performance interaction occurs (according to our definition) when the performance of multiple options cannot be explained by their individual performances. Automated interaction detection based on sampling techniques has been successful in improving performance prediction in configurable systems; it has found that performance interactions occur mostly among pairs of options, but also beyond.

Third, at the level of source code, interactions are manifested as glue code that combines or coordinates the implementation of multiple options; the additional code is only included in the program if all options are selected. This pattern is common in component connectors,

lifters, derivatives, connector plugins, as well as code in nested ifdef directives and if statements. Code-level interactions are relatively easy to identify (especially, for compile-time configuration mechanisms), but they are not necessarily useful predictors of other kinds of interactions.

Finally, in a running system, we consider it as an interaction when the value of a program variable depends on multiple configuration options [4]. For example, when executing a test case in a configurable system, we expect most variables to have the same value in all executed configurations, and a few variables to have two (or more) alternative values depending on a single option. However, variables can potentially depend on many options. In a study of Wordpress, we found dependencies among up to 16 of 50 optional plugins.

Overall, the picture of feature interactions in configurable systems is diverse, and there is no single, feasible classification or terminology. Working with real systems provides lots of data and much opportunity for studying interactions. While some of these interactions are easy and reliable to detect, others require intensive testing and sophisticated sampling. We hope that, in the long run, we are able to identify correlations between different kinds of interactions, found with different techniques, and to combine them effectively [2].

References

1 S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, October 2013.

2 S. Apel, S. Kolesnikov, N. Siegmund, C. Kästner, and B. Garvin. Exploring Feature Interactions in the Wild: The New Feature-Interaction Challenge. In *Proc. of the Int'l Workshop on Feature-Oriented Software Development (FOSD)*, pages 1–8. ACM, 2013.

3 M. Calder, M. Kolberg, E. Magill, and S. Reiff-Marganiec. Feature Interaction: A Critical Review and Considered Forecast. *Computer Networks*, 41(1):115–141, 2003.

4 H. Nguyen, C. Kästner, and T. Nguyen. Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications. In *Proc. of the Int'l Conf. on Software Engineering (ICSE)*, pages 907–918. ACM, 2014.

5 C. Nie and H. Leung. A Survey of Combinatorial Testing. *ACM Computing Surveys*, 43(2):1–29, 2011.

6 N. Siegmund, S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake. Predicting Performance via Automated Feature-Interaction Detection. In *Proc. of the Int'l Conf. on Software Engineering (ICSE)*, pages 167–177. IEEE, 2012.

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

3.4 Feature Interactions in Smartphones

Christian Prehofer (fortiss GmbH – München, DE)

This talk reviews feature interactions in telecom, mobile phones and smartphones. We first discuss the history of feature interactions in telecommunications and focus on typical reasons for feature interactions. In particular, Software and standards evolution over long period of time as well as legacy devices are a frequent reason behind feature interactions. Then, we discuss technology developments in telephony systems, standards as well as system platforms which relate to these causes of feature interactions.

3.5 Behaviours and Feature Interactions

Michael Jackson (The Open University – Milton Keynes, GB)

- The feature interaction problem arises in the physical problem world of computer-based systems rather than in the machine – the software – per se. The difficulty, in essence, is that each feature places its own demands on the physical behaviour of the problem world, and that the demands of different features may be in some way incompatible. In automotive software, the speed limiting and cruise control features may conflict: the car can have only one speed at any one time. In controlling an elevator, normal use conflicts with use by firefighters: when the lift stops at a floor, normal use demands that the doors close automatically after a specified delay, but firefighter use demands that they close only in response to the Door_Close button in the lift car.
- Direct conflict is far from the only type of feature interaction. Additional interaction types include: mutual exclusion, interference, resource sharing, and – very commonly – switching, in which control of some part of the problem world is passed from one regime that has terminated to another that has been newly activated. Further, the physical nature of the problem world can vitiate apparently sound reasoning: the fact that each of two physical demands can be satisfied in isolation gives no guarantee that they can be satisfied in combination. Even when the alphabets of the two demands appear to be disjoint they may interact through the medium of other phenomena that have been neglected in the analysis.
- In computer-based systems, which interact with and control the physical world, our primary concerns are with system dynamics. The behaviour of the system – of the interacting computing machine and problem world – is the essential product of software development, and the salient aspect of this behaviour is in the problem world, not in the software. In developing the software we are developing this behaviour, and we may usefully regard feature interaction as the interaction among the constituent behaviours that together make up the whole behaviour of the system. Because the behaviour of a realistic – and especially a critical – system is very complex, it is necessary to adopt a disciplined approach to its design.
- The design approach suggested here structures the system behaviour into its constituent behaviours by a combination of top-down and bottom-up decomposition. Each constituent behaviour is considered in its totality: that is, the desired problem world behaviour is considered together with the software behaviour that will evoke it and also the behaviours of all parts of the problem world that lie implicitly on a causal path between the machine and the problem world behaviour explicitly desired.
- Further elements of the suggested approach are important. First, each proposed constituent behaviour is initially considered in a loose decomposition: it is considered in isolation, as if it were a complete stand-alone system in itself, ignoring its eventual interactions with other constituent behaviours. Second, the complexity of each constituent behaviour is developed in stages: the main-line isolated behaviour; the main-line behaviour elaborated as necessary to handle exceptional conditions; and the elaborated behaviour further complicated by its interactions with other constituent behaviours. Third, the control of behaviour is rigorously separated from behaviour content.
- This approach offers advantages in the identification and treatment of feature interactions. Considering each behaviour in its totality maintains an awareness of the physical implica-

tions of the design both of the desired behaviour and of the software behaviour that will evoke it. This awareness is strengthened by the emphasis on simplicity, separating out the sources of behavioural complexity. The insistence on loose decomposition ensures that combination of the constituent behaviours to give the complete system behaviour is a distinct and explicitly recognised design task in which the constituents to be combined are already well understood. In this combination task, feature interactions are the heart of the design problem to be addressed, and much – perhaps, everything possible – has been done to make the task and its design problem as perspicuous as it can be.

4 Lightning Talks

4.1 Extracting Feature Model Changes from the Linux Kernel using FMDiff

Nicolas Dintzner (TU Delft, NL)

Joint work of Dintzner, Nicolas; Van Deursen, Arie; Pinzger, Martin
Main reference N. Dintzner, A. Van Deursen, M. Pinzger, "Extracting feature model changes from the Linux kernel using FMDiff," in Proc. of the 8th Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'14), Article No. 22, ACM, 2014.
URL http://dx.doi.org/10.1145/2556624.2556631

The Linux kernel feature model has been studied as an example of large scale evolving feature model and yet details of its evolution are not known. We present here a classification of feature changes occurring on the Linux kernel feature model, as well as a tool, FMDiff, designed to automatically extract those changes. With this tool, we obtained the history of more than twenty architecture specific feature models, over ten releases and compared the recovered information with Kconfig file changes. We establish that FMDiff provides a comprehensive view of feature changes and show that the collected data contains promising information regarding the Linux feature model evolution.

4.2 Feature Interactions Taxonomy and Case Studies

Sergiy Kolesnikov (University of Passau, DE)

Joint work of Apel, Sven; Kolesnikov, Sergiy; Siegmund, Norbert; Kästner, Christian; Garvin, Brady
Main reference S. Apel, S. Kolesnikov, N. Siegmund, C. Kästner, B. Garvin, "Exploring Feature Interactions in the Wild: The New Feature-Interaction Challenge," in Proc. of the 5th Int'l Workshop on Feature-Oriented Software Development (FOSD'13), pp. 1–8, ACM, 2013.
URL http://dx.doi.org/10.1145/2528265.2528267

The *feature-interaction problem* has been keeping researchers and practitioners in suspense for years. Although there has been substantial progress in developing approaches for modeling, detecting, managing, and resolving feature interactions, we lack sufficient knowledge on the kind of feature interactions that occur in real-world systems. In this talk, we set out the goal to explore the nature of feature interactions systematically and comprehensively, classified in terms of order and visibility. Understanding this nature will have significant implications on research in this area, for example, on the efficiency of interaction-detection

or performance-prediction techniques. A set of preliminary results as well as a discussion of possible experimental setups and corresponding challenges give us confidence that this endeavor is within reach but requires a collaborative effort of the community.

4.3 (Structural) Feature Interactions for Variability-Intensive Systems Testing

Gilles Perrouin (University of Namur, BE)

Joint work of Perrouin, Gilles; Henard, Christopher; Papadakis, Mike; Klein, Jacques; Heymans, Patrick; Le Traon, Yves
Main reference C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, Y. L. Traon, "Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines," IEEE Transactions on Software Engineering, 40(7):650–670, 2014.
URL http://dx.doi.org/10.1109/TSE.2014.2327020

Adopting a middle-ground view between Michael Jackson's ("features are behaviours") and Christian Kaestner's ("features are configuration options") definitions, we consider features as units of variability specified within a feature model. Feature models can be used to document valid choices (called configurations) formed by combinations of features. Such configurations can either relate to desired elevator behaviours interactions (normal use, emergency use, etc.) or to viable Linux kernels. The number of configurations derivable from a given feature model grows exponentially with the number of features, making the testing process inherently difficult. To harness combinatorial explosion of the number of configurations to be considered, we propose to sample them by computing t-way interactions from the feature model. We present initial experiments and a search-based approach maximising dissimilarity between configurations. This approach mimics combinatorial interaction testing techniques in a flexible and scalable manner.

References

1 Gilles Perrouin, Sebastian Oster, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves le Traon. Pairwise testing for software product lines: comparison of two approaches. *Software Quality Journal*, 20(3-4):605–643, 2012.
2 Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Pledge: A product line editor and test generation tool. In *Proceedings of the 17th International Software Product Line Conference Co-located Workshops*, SPLC'13 Workshops, pages 126–129, New York, NY, USA, 2013. ACM.

4.4 Performance Prediction in the Presence of Feature Interactions

Norbert Siegmund (University of Passau, DE)

Joint work of Siegmund, Norbert; Kolesnikov, Sergiy; Christian, Kästner; Apel, Sven; Batory, Don; Rosenmüller, Marko; Saake, Gunter

Customizable programs and program families provide user-selectable features allowing users to tailor the programs to the application scenario. Beside functional requirements, users

are often interested in non-functional requirements, such as a binary-size limit, a minimized energy consumption, and a maximum response time.

In our work, we aim at predicting a configuration's non-functional properties for a specific workload based on the user-selected features [2, 3, 4]. To this end, we quantify the influence of each selected feature on a non-functional property to compute the properties of a specific configuration. Here, we concentrate on performance only.

Unfortunately, the accuracy of performance predictions may be low when considering features only in isolation, because many factors influence performance. Usually, a property is program-wide: it emerges from the presence and interplay of multiple features. For example, database performance depends on whether a search index or encryption is used and how both features interplay. If we knew how the combined presence of two features influences performance, we could predict a configuration's performance more accurately. Two features interact (i. e., cause a performance interaction) if their simultaneous presence in a configuration leads to an unexpected performance, whereas their individual presences do not. We improve the accuracy of predictions in two steps: (i) We detect which features interact and (ii) we measure to what extent they interact. In our approach, we aim at finding the sweet spot between prediction accuracy, measurement effort, and generality in terms of being independent of the application domain and the implementation technique. The distinguishing property of our approach is that we neither require domain knowledge, source code, nor complex program-analysis methods, and our approach is not limited to special implementation techniques, programming languages, or domains.

Our key idea to determine which features interact is the following: We measure each feature twice. In the first run, we try to measure the performance influence of the feature in isolation by measuring the variant that has the smallest number of additionally selected features. The second run, aims at maximizing the number of features such that all possible interactions that may influence on performance materialize in the measurement. If the influence of the feature in isolation differs with the influence when combined with other features, we know that this feature interacts. In the second step, we perform several sampling heuristics, such as pair-wise sampling, to determine the actual combinations of interacting features that cause interactions.

Our evaluation is based on six real-world case studies from varying domains (e. g., databases, encoding libraries, and web servers) using different configuration techniques. Our experiments show an average prediction accuracy of 95 percent, which is a 15 percent improvement over an approach that takes no interactions into account [1].

References

1 N. Siegmund, S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake. Predicting Performance via Automated Feature-Interaction Detection. In *Proc. ICSE*, pages 167–177. IEEE, 2012.

2 N. Siegmund, M. Rosenmüller, C. Kästner, P. Giarrusso, S. Apel, and S. Kolesnikov. Scalable Prediction of Non-functional Properties in Software Product Lines. In *Proc. SPLC*, pages 160–169. IEEE, 2011.

3 N. Siegmund, M. Rosenmüller, C. Kästner, P. Giarrusso, S. Apel, and S. Kolesnikov. Scalable Prediction of Non-functional Properties in Software Product Lines: Footprint and Memory Consumption. *Information and Software Technology*, 55(3):491–507, 2013.

4 Norbert Siegmund, Alexander von Rhein, and Sven Apel. Family-Based Performance Measurement. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 95–104. ACM, 2013.

4.5 Feature Interaction in the Browser and the Software-Defined Network

Shriram Krishnamurthi (Brown University, US)

In this presentation, I give a brief overview of feature interaction problems as they occur in current real-world systems and are likely to occur in future ones. I illustrate the present using Web browser extensions and the problem of finding violations (especially of security-sensitive properties such as Private Browsing Mode). For the future, I speculate that software-defined networking will result in "app stores" of networking behavior, which will inevitably interact in unsavory ways and will need to be kept distinct.

4.6 Feature Interaction and Emergent Properties

Gerhard Chroust (Johannes Kepler Universität Linz, AT)

Main reference G. Chroust, "System properties under composition," in Proc. of the European Meeting on Cybernetics and Systems Research (EMCSR'02), pp. 203–208, Austrian Society for Cybernetic Studies, 2002.

The interaction of features is a mixed blessing in engineering. It enables useful functions e. g. radio reception by utilizing resonance, but it can also have disastrous and destructive effects like the collapse of a bridge due to undesirable resonance. In systems theory the so-called emergent properties of systems with individual components show similar effects. In this paper I explore the similarities between Feature Interaction in Systems Engineering and Emergent Properties in Systems Theory and show the analogy between these two concepts.

According to my understanding (following K.C. Kang, 1990) *"a feature of a . . . product can be described as a prominent or distinctive user-visible aspect, quality, or characteristic . . . ".*

When composing a system from interconnected subsystems (components) the properties of the resulting system (and their predictability!) are one of the key issues of engineering. Difficulties stem from two observations:

- The structure of the system plays a key role.
- A composed system often exhibits 'unexpected' behavior due to the occurrence of so-called emergent properties.

In general a system's properties depend on the structure of the system and on all properties of all components in the system AND are usually different from the properties of the individual components.

We define an *"emergent property of a system is a property which cannot be determined solely from the properties of the system's components, but which is additionally determined by the system's structure (i. e. by the way the parts are connected to form the system)".*

Emergent properties depend inherently and essentially on the structure of the system i. e. on the way the system is composed, and change with a change of the structure! It is often not easy to determine them from the system, they often appear as a surprise ("emergence"!).

It is obvious that emergent properties complicate the prediction of system properties since their value can only be determined by considering also the system structure. As

long as we allow any imaginable structure for a system there is little chance to make any reasonable statements about the emergent properties. A restricted set of admissible composition structures allows to make some statements about the behavior of emergent properties under composition. Software patterns are good candidates for such "admissible composition patterns."

All Features are system properties, but not all system properties are to be considered Features. Feature Interactions can be interpreted as emergent properties since they usually depend on the (static and/or dynamic) structure of the system.

One of the standard examples in the seminar was an automatic door locking/unlocking mechanism which was dependent on the time-of-the-day, the setting of several switches with different purposes and a timeout facility. The resulting feature could be called "safety of the house" and is a typical "emergent property," where both the physical structure (whether some components are arranged in parallel or serial order) and also the chronological order (in which certain settings are activated) have a decisive influence on the outcome.

My contribution established the strong analogy between Feature Interaction in systems engineering and the system theoretic concept of emergent properties in multi-component systems. This analogy can be used to consolidate the terminology and exchange insights between these two domains. I hope that this will be a source for fruitful discussion and more clarification in both areas.

4.7 Extending Ruby into a DSL Good and Bad Feature Interactions

Thomas Gschwind (IBM Research GmbH – Zürich, CH)

Joint work of Michael H. Kalantar, James Doran, Tamar Eilam, Michael D. Elder, Fabio Oliveira, Edward C. Snible, Tova Roth
Main reference M. H. Kalantar, F. Rosenberg, J. Doran, T. Eilam, M. D. Elder, F. Oliveira, E. C. Snible, T. Roth, "Weaver: Language and runtime for software defined environments," IBM Journal of Research and Development, 58(2/3):10:1–10:12, 2014.
URL http://dx.doi.org/10.1147/JRD.2014.2304865

We present our experiences of using Ruby as the basis for Weaver, a Domain Specific Language (DSL) to describe applications to be deployed and run in a Cloud or Cloud-like environment. Weaver describes the requirements and constituents of such an application. Based on this description, Weaver automates the deployment of such applications and facilitates the cooperation between the development and operations teams.

Weaver is an Internal DSL which means it extends the Ruby language rather than being implemented as a DSL from scratch. Extending Ruby has several advantages. For instance, we do not have to implement our own parser and additionally because our language. Additionally, our DSL shall interoperate with Chef, another Ruby-based DSL. Hence, by using Ruby, users do not have to learn yet another totally different language.

By extending an existing language we have to deal with features provided by the base language interferring with features in our DSL. One such problem is error handling, as any stack traces produced by the Ruby language will intermingle with functions provided by users as part of the DSL with functions provided by the framework used to implement the DSL.

Another advantage of Ruby is its ability to sandbox code and override Ruby's approach for looking up symbols which allows to intercept access to variables etc. This is typically achieved with the `method_missing` method which Ruby typically invokes when a name cannot be

resolved. However, care has to be taken to distinguish legitimate calls to this method from those generated by spelling mistakes and again to factor this into errors generated by the DSL.

4.8 Probabilistic Model Checking of DTMC Models of User Activity Patterns

Oana M. Andrei (University of Glasgow, GB)

Joint work of Andrei, Oana; Calder, Muffy; Higgs, Matthew; Girolami, Mark
Main reference O. Andrei, M. Calder, M. Higgs, M. Girolami, "Probabilistic Model Checking of DTMC Models of User Activity Patterns," in Proc. of the 11th Int'l Conf. on Quantitative Evaluation of Systems (QEST'14), LNCS, Vol. 8657, pp. 138–153, Springer, 2014; pre-print available as arXiv:1403.6678v1 [cs.SE].
URL http://dx.doi.org/10.1007/978-3-319-10696-0_11
URL http://arxiv.org/abs/1403.6678v1

Software developers cannot always anticipate how users will actually use their software as it may vary from user to user, and even from use to use for an individual user. In order to address questions raised by system developers and evaluators about software usage, we define new probabilistic models that characterise user behaviour, based on activity patterns inferred from actual logged user traces. We encode these new models in a probabilistic model checker and use probabilistic temporal logics to gain insight into software usage. We motivate and illustrate our approach by application to the logged user traces of an iOS app. Next we will consider how to represent the orthogonal concerns of two classes of features – activity patterns and structural variability (e. g. configurability) of software systems, and their combined impact on user experience and user engagement.

4.9 Presence-Condition Simplification

Alexander von Rhein (University of Passau, DE)

Joint work of Apel, Sven; Berger, Thorsten; Beyer, Dirk; Grebhahn, Alexander; Siegmund, Norbert

Analysis approaches for configurable systems take system variability explicitly into account. The notion of presence conditions is central to such approaches. A presence condition specifies a subset of system configurations in which a certain artifact is present (e. g., the presence of a certain piece of code) or any other concern of interest that is associated with this subset (e. g., the presence of a defect). Our informal goal is to raise awareness of the problem of presence-condition simplification; we will demonstrate that presence conditions often contain redundant information, which can be safely removed in the interest of simplicity. As contributions, we present a formalization of the problem of presence-condition simplification, discuss various application scenarios, compare different algorithms for solving the problem, and report on an empirical evaluation comparing the algorithms by means of a set of substantial case studies.

4.10 On the Relation between Feature Dependencies and Change Propagation

Bruno Cafeo (PUC – Rio de Janeiro, BR)

As the SPL evolves, dealing with feature dependencies in the source code in a cost- and effort-effective way is challenging. It is expected that changes affect a minimum of existing features as possible. However, changes in one feature usually require changes in the code of other dependent features. In this context, it is important to have an understanding on the relation between feature dependency and change propagation. To this end, we present an exploratory study analysing this relation in five evolving SPLs. The results revealed that the extent of change propagation in SPL features might be higher than it was found in studies of change propagation in modules of stand-alone programs (i. e., non-SPL). We also found a high concentration of change propagation in a few feature dependencies. This result shows that feature dependencies are not alike regarding change propagation.

5 Breakout Groups: Domain-independence of Feature Interactions

5.1 Summary of Group 1

Krzysztof Czarnecki (University of Waterloo, CA)

What We've Done

- Revisited
  - Each plenary talk
  - Gerhard's presentation on systems theory
- Analyzed
  - The notion of feature used
  - The notion of feature interactions used
  - The handling of features and feature interactions in the lifecycle
- Recorded
  - Similarities and differences
  - Questions

Features

- Feature notion
  - Requirement or Behavior decomposition to understand a problem
  - Implementation unit to achieve reuse
- Purpose of features
  - Incremental development (additive) vs. independent design ("care about interactions later")
  - Single system vs. product line (variability)
- Other characteristics
  - Automatic (prepared) composition vs. manual integration (combination)
  - Open vs. close: Are features independently developed
  - Functional vs. non-functional properties

Feature Interactions

- Vaguely: "feature behave differently together than in isolation"
  - "surprising or unexpected", different, missing, extra behavior or properties
- Coordination code
  - Manifestation of interactions in the implementation
  - Potential exponential explosion in configurable systems
- Systems theory
  - Studying composition of components in systems and the resulting properties
  - Structure-independent properties (like mass) vs. emergent properties (like usability)

Handling Feature Interactions

- Upfront composition mechanism (architecture) vs. manual integration (combination)
  - Isolating features (e. g., Android)
- For some properties there is hope of compositionality, for some there isn't
- Partial specifications, feature-based specifications
- Address by updates

Discussion

- Are there any conceptual problems?
- Hierarchy considered important?
- Can we learn from systems theory?
- Michael: Problem-oriented decomposition vs. component-oriented decomposition?
- Definition of feature interaction?

5.2 Summary of Group 2

Sandro Schulze (TU Braunschweig, DE), Sebastian Erdweg (TU Darmstadt, DE)

Note: This abstract is the result of a breakout group a the Dagstuhl Seminar 14281 on Feature Interactions.

When talking about feature interactions (FIs) it becomes clear quickly that, although people work in related domains, they may have totally different views on feature interactions, which needs to be taken into account in discussions. Basically, we identified three major views on feature interactions within our breakout group. First, feature interactions may be documented in a specification such as through requirements or contracts (for example, the latter is used for verification). In any case, this provides a rather formal (and sometimes theoretical) way of defining expected FIs. Second, the source code itself may manifest a variety of feature interactions such as method calls or method extensions between two or more features. Finally, we argue that even the user may have expectations about how certain features interact, even if this expectation is not spelled out explicitly. Just think about modern cars and their capabilities to support the driver in driving the car. For instance, having a cruise control and a speed limit assistant, a driver has certain expectation how both work together, that is, how they interact.

Detecting and validating existing feature interactions is difficult without a specification that describes the expected behavior. However, during our discussion we came to the conclusion that the implementation of a feature can induce a specification beyond a formal

system or requirements specification. For instance, the requirement that a certain program (or product of a product line) does compile can be considered as a (somewhat implicit) specification. Hence, features that prevent a program from compiling in fact do constitute unwanted interactions. Other examples for such specifications are generic properties (such as absence of deadlocks or conflicts), system invariants, a poor user experience (i. e., the experienced interactions are criticized), or concrete feature specifications such as test cases. All of these specifications may support developers in detecting and validating wanted and unwanted feature interactions.

Even when we can detect feature interactions, it is even more challenging to guarantee a certain behavior of features in concert. We especially identified the fact that many (eco) systems are open world and require specific platform support for dealing with feature interactions. For example, one can deal with FIs by coordinating features on the architectural level. In other eco systems, such as the Android platform, it is common to adhere to certain conventions and to involve the user in resolving feature interactions. Finally, a practical solution to deal with feature interactions is to a) provide a default behavior in case of alternatives and b) to let the user decide which behavior she wants in case of alternatives (e. g., think of the case of selecting an App for displaying a PDF file). In any case, there is no silver bullet for how to deal with feature interactions, especially in an open-world scenario, but it is necessary to choose at least one way to resolve possible conflicts. Sometimes, this may even be a manual and time-consuming task such as informing developers about unintended FIs and (optionally) providing a corresponding patch to fix it.

To summarize, we think that feature interactions can not be treated generically, because different views and other non-functional factors have to be considered. Nevertheless, it is clear that a) there always has to be some kind of specification and b) at least an idea of how to deal with interactions when they arise. Of course, it depends on the criticality of such interactions how and when to resolve them.

5.3 Summary of Group 3

Oscar M. Nierstrasz (Universität Bern, CH)

This breakout group discussed open issues and challenges for feature interaction.

We identified two essentially different perspectives on feature interaction:

1. **Design:** Here the focus is on understanding requirements with a view towards building a system in which features either do not interact, or do so in a positive way.
2. **Development:** The focus here is on code, with a view towards analysing and understanding an existing system. *Tools* are used to detect FI bugs or control quality.

We discussed three questions related to a taxonomy of FI:

What's a feature? We reviewed the results of the feature interaction survey conducted by the organizers and concluded that the two perspectives (i. e., Design vs Development) dominated: Features are either considered to be requirements (design) or units of functionality, behaviour etc. (development).

What's feature interaction? Generally, FI means that *features behave differently in isolation than in combination.* FI can take different forms: (i) features are independent and compose additively; (ii) features may depend on or require other features; (iii) combinations of

features may exhibit "emergent behaviour"; (iv) features may conflict, yielding logical inconsistencies or unacceptable behaviour.

How to deal with FIs? We identified three general approaches. (i) Modeling FI: specify desired properties (e. g., safety/liveness, non- functional properties etc.). (ii) Detecting FIs: use approaches like testing, model checking or principle component analysis to expose unknown FIs. (iii) Resolve FIs: use static or dynamic techniques, such as coordination patterns with priorities to resolve FIs.

Finally we discussed a number of open challenges and tasks for the FI community.

- Bridging the gap between the Design and Development perspectives of FI.
- Producing a map/taxonomy of FI systems and views.
- Eliciting FI patterns.
- Designing a "feature-aware computational model", i. e., that expresses when features interact or interfere

6 Breakout Groups: Framework for Modeling Feature Interactions

6.1 Summary of Group 1

Michael Jackson (The Open University – Milton Keynes, GB)

In this breakout session we found no reason to disgree specifically with the reference model proposed by Pamela Zave for discussion and criticism. However, one member of the group was sceptical about its possible value, on the grounds that wherever there is interaction it must have some locus: calling that locus a 'problem domain' added little or nothing to our understanding.

We agreed that feature interactions revealed in apparent anomalies of performance were likely to be located in a problem domain from which the analysis in hand had abstracted. So for example, the interaction might be due to conflicting demands for positioning the arm of a disk drive, where the disk drive itself was not explicitly mentioned in the immediate analysis.

It was suggested that the reference model was primarily intended as an aid to development, structuring the problem world and hence contributing to structuring the problem itself. Features could in particular cases be regarded as delimited by development or requirement modularity, or by software modules.

The correspondence between 'subproblem machines' and 'behaviours' (or perhaps 'features') could not be expected to carry across into software structure, for which structural transformation would surely be necessary.

Feature interaction detection by analysis of program texts seemed to some participants no different from any other formal analysis of program texts; a 'feature' construct in programming languages seemed highly desirable.

6.2 Summary of Group 2

Kathi Fisler (Worcester Polytechnic Institute, US)

One of the breakout groups discussed the idea that every feature interaction in every domain could be described as a conflict within a design domain pertinent to the system. We noted early that the hypothesis was almost certainly true, but the real question was whether this approach made practical sense: are there domains or kinds of interactions for which the design domain that witnesses the interaction is simply too fine-grained to be modeled or analyzed?

Each person in our breakout group contributed a feature interaction of interest (from a domain that that participant studies). We then selected two to discuss in more detail.

The contributed interactions included: different markup features in a text editor; synchronization policies that break upon inheritence in OO code; performance anomalies in course-management software; semantic features of an IDE that must change after the supported languages change; bugs arising from the introduction of interaction code to mediate between features; performance impacts when combining compression and encryption; call forwarding and voicemail; spam picking up email from colleagues in the contact list; and displaying filtered papers despite conflict-of-interest settings in a conference manager.

We briefly discussed the telephony example, questioning what resource was in question between call forwarding and voicemail. These simply seemed like different user-level preferences. Later discussion clarified that the single voice channel carrying the data between these two options constituted the resource in contention.

We spent much of our time debating the "change in IDE" example: we generally converged on the opinion that this was not really a feature interaction. One participant raised the idea that something counts as an interaction if some component of the system could be "blamed" for causing the problem (thus distinguishing cases of misuse from situations where interactions simply emerge through no error on the part of existing components). The group disbanded before getting to explore the blame idea in sufficient detail.

7 Panel Discussions

7.1 Reflections and Perspectives

Sven Apel (University of Passau, DE)

The panel discussion on the final day wrapped up the seminar. The format was a concentric-circles format. In the inner circle, five researchers discussed the results of the seminar, their insights, and avenues of further research on feature interactions. The panelists were: Marsha Chechik, Krzysztof Czarnecki, Michael Jackson, Christian Kästner, Pamela Zave. Joanne Atlee moderated the panel.

The panel session summarized that, during the seminar, we have seen that, in many different domains, the feature-interaction problem exists: In the telecommunication domain, the problem of feature interactions has been addressed for years, but now it appears in more

and more domains, which is owed to the fact of ever increasing complexity. An important question is whether research results can be transferred from one domain to another, and to what extent results can be domain dependent. To answer this question, the panelists suggested establishing a catalog of exemplary, domain-specific feature interactions. Examples often need to be specific to a particular domains because the domains and solutions are very different with respect to abstraction level, methodology, and requirements. The catalog is supposed to reveal – if possible – more general feature-interaction patterns, arising from concrete instances. A follow-up seminar or workshop on modeling and feature interactions shall be established to collect the data for such a catalog.

An important insight mentioned during the panelists' discussion was that we require a clearer definition of what a feature is (or should be) and how we handle features at an architectural level. To achieve better architectures, the community should investigate how to resolve and combine features independently of the domain. A catalog of feature interactions that occur in practice would be useful for this investigation.

According to the panelists, the seminar showed that there are many different aspects of feature interactions, such as the effect (good or bad), the developers' or designers' intention (intended or unintended), and the context of the interaction (design or implementation level). Several participants noted that this observation broadened their understanding of the relationship between behaviors of features in the real world and how program features work. During the discussion, it became clear that feature-oriented systems (1) are often engineered by people who are not aware of the feature-interaction problem, and (2) that are used in safety-critical domains. Therefore (intended) feature interactions must be very well explained, and we should not strive for solutions (for unintended interactions) that work only half the time. That said, features have the great potential to help people understanding complex systems in terms of the features, incl. their interactions, they provide.

Finally, the panelists stated that, while several domains experience problems similar to feature interactions, they do not call them feature interactions. An example are cars where single features are updated in a garage or even "over the air", which give rise to feature interactions. These domains are a rich field for feature-interaction research. How can we put "the feature-interaction stamp" on them? One participant suggested having a keynote on feature interactions at a major software-engineering conference.

Participants

- Oana M. Andrei
University of Glasgow, GB
- Sven Apel
University of Passau, DE
- Joanne M. Atlee
University of Waterloo, CA
- Luciano Baresi
Politecnico di Milano Univ., IT
- Sandy Beidu
University of Waterloo, CA
- Bruno Cafeo
PUC – Rio de Janeiro, BR
- Marsha Chechik
University of Toronto, CA
- Gerhard Chroust
Universität Linz, AT
- Krzysztof Czarnecki
University of Waterloo, CA
- Nicolas Dintzner
TU Delft, NL
- Sebastian Erdweg
TU Darmstadt, DE
- Kathi Fisler
Worcester Polytechnic Inst., US
- Stefania Gnesi
CNR – Pisa, IT
- Thomas Gschwind
IBM Research GmbH – Zürich, CH
- Reiner Hähnle
TU Darmstadt, DE
- Michael Jackson
The Open University – Milton Keynes, GB
- Cliff B. Jones
Newcastle University, GB
- Christian Kästner
Carnegie Mellon University, US
- Mario Kolberg
University of Stirling, GB
- Sergiy Kolesnikov
University of Passau, DE
- Shriram Krishnamurthi
Brown University, US
- Malte Lochau
TU Darmstadt, DE
- Oscar M. Nierstrasz
Universität Bern, CH
- Gilles Perrouin
University of Namur, BE
- Christian Prehofer
fortiss GmbH – München, DE
- Gunter Saake
Universität Magdeburg, DE
- Sandro Schulze
TU Braunschweig, DE
- Norbert Siegmund
University of Passau, DE
- Stefan Sobernig
Universität Wien, AT
- Mirco Tribastone
University of Southampton, GB
- Alexander von Rhein
University of Passau, DE
- Andrzej Wasowski
IT Univ. of Copenhagen, DK
- Pamela Zave
AT&T Labs Research – Bedminster, US