Exploring Feature Interactions in the Wild

The New Feature-Interaction Challenge

Sven Apel,
Sergiy Kolesnikov,
Norbert Siegmund
University of Passau
Germany

Christian Kästner
Carnegie Mellon University
USA

Brady Garvin
University of
Nebraska—Lincoln
USA

ABSTRACT

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 position paper, 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.

Categories and Subject Descriptors: D.3.1 [**Software Engineering**]: Management—*Software Configuration Management*; D.3.3 [**Software Engineering**]: Reusable Software—*Domain Engineering*

General Terms: Management, Experimentation, Performance, Reliability

Keywords: Feature Interactions, Feature-Interaction Problem, Feature Modularity, Feature-Oriented Software Development

1. INTRODUCTION

> The feature interaction problem is fascinating because it is real and easy to explain, yet has been hard to pin down in a satisfactory way [12].

Feature modularity is the holy grail of feature-oriented software development. Ideally, one can deduce the behavior of a system composed from a set of features solely on the basis of the behavior of the features involved. But, features interact. Despite substantial progress in developing plugin architectures for feature composition [25, 26, 52, 53], formalisms for describing features as coherent units [10, 13, 20, 39], as well as modularization techniques for

crosscutting feature implementations, such as feature-oriented programming [1, 2, 9, 45] and aspect-oriented programming [3, 31, 34], feature interactions are still a major challenge and counteract feature modularity and compositional reasoning [14, 29, 43, 57].

A *feature interaction* occurs when the behavior of one feature is influenced by the presence of another feature (or a set of other features). Often, the interaction cannot be deduced easily from the behaviors of the individual features involved, which hinders compositional reasoning. A classic example is the inadvertent interaction between the call-forwarding and call-waiting features of a telephony system [14]: If both features are activated, the system reaches an undefined, possibly unsafe state when it receives a call on a busy line.

Not all feature interactions are undesired. Often a feature communicates and cooperates with other features to accomplish a task in concert. For example, transaction management and locking in a database system cooperate to ensure atomicity, consistency, isolation, and durability [24].

Nevertheless, it is imperative to analyze and understand the consequences of *all* feature interactions inside a system, and, possibly, to resolve undesired interactions. The root of the feature-interaction problem is that the number of *potential* interactions in a system is exponential in the number of features.

A question that has not been addressed so far in sufficient breadth and depth is: Which kinds of feature interactions exist in real-world systems and how do they manifest? Especially, we are interested in two dimensions that characterize feature interactions: (1) order and (2) visibility. The *order* of a feature interaction is defined as the minimal number of features (minus one) that need to be activated to trigger the interaction. For example, an interaction between two features is of order one. The *visibility* of a feature interaction denotes the context in which a feature interaction appears. Feature interactions may appear at the level of the externally observable behavior of a program, including functional behavior (e.g., segmentation faults and all kinds of other bugs) and non-functional behavior (e.g., performance anomalies and memory leaks). Feature interactions may also appear internally in a system, at the level of code that gives rise to an interaction or at the level of control and data flow of a system (e.g., data-flows that occur only when two or more features are present). We believe that there may be systematic correlations between externally-visible and internally-visible interactions, which is a major motivation for our endeavor to explore and understand the nature of feature interactions.

There is a large body of research on detecting, managing, and resolving different kinds of feature interactions, in various domains (e.g., Internet applications [16], service systems [55], automotive systems [18], software product lines [27], requirements engineering [42], and computational biology [19]), and using different ap-

proaches (e.g., formalisms describing features and their interactions [10, 13, 20, 39], architectures that avoid classes of interactions [25, 26, 52, 53], sampling, static-analysis, and model-checking approaches for interaction detection [4, 5, 15, 21, 27, 28, 44, 48], and techniques for resolving interactions at run-time [22, 51]). While this body is substantial and diverse, the individual approaches and studies concentrate on specific kinds of interactions, in specific settings, using specific solutions.

In this position paper, we set out the goal to explore the nature of feature interactions systematically and comprehensively. Which feature interactions occur in real-world systems? What order do they have? Are they internally or externally visible? Answering these questions will immediately lead to a series of follow-up questions: Which strategies for interaction detection, management, and resolution are superior in which circumstances (e.g., different sampling heuristics vs. variability-aware analysis vs. feature-based analysis [7, 36, 38, 50])? Do different kinds of (internally and externally-visible) feature interactions correlate (e.g., are memory leaks caused by interactions among multiple features correlate with data-flow interactions)? Can we predict feature interactions of one kind based on the knowledge about feature interactions of another kind (e.g., predicting performance anomalies based on control-flow and data-flow interactions)? Notably, some empirical studies have been published in the literature, mostly concentrating on specific kinds of interactions in specific settings [21, 28, 30, 33, 37, 48, 56], but a systematic and coordinated effort is necessary to answer these questions.

This paper is not meant to make a technical contribution to the area, but to propose and motivate a research endeavor based on a review and classification of feature interactions addressed in the literature. In particular, we make the following contributions:

- A discussion and classification of different kinds of feature interactions addressed in the literature.

- An agenda, setup, and a discussion of challenges for a research endeavor to explore the nature of feature interactions, as well as a report on preliminary results.

As a call for participation to the community, let us accept and address the *feature-interaction challenge*!

2. CHARACTERIZING FEATURE INTERACTIONS

Different kinds of feature interactions have been discussed in the literature, and there have been many attempts to characterize and classify feature interactions [12, 35, 54, 55]. In the quest of understanding feature interactions, we classify feature interactions along two dimensions: (1) order and (2) visibility. But, before we explain this classification, we stress the role of specifications for the feature-interaction problem.

2.1 Specifications

Talking about feature interactions without talking about specifications makes little sense. To identify a feature interaction, one needs to be able to identify deviating and inadvertent properties or behaviors. A *specification* defines the expected behavior when features are combined. If a feature combination $f_1 \bullet \ldots \bullet f_n$, with '$\bullet$' denoting composition, satisfies specification ϕ, we write:

$$f_1 \bullet \ldots \bullet f_n \models \phi \qquad (1)$$

Specifications may be concerned with individual features (stating their expectations and provisions; in this case, ϕ in Equation 1

would be combined of multiple smaller specifications, associated with individual features) or combinations of features (e.g., stating properties that all feature combinations must exhibit) [6]. In the remainder, we abstract over this difference, for simplicity.

Furthermore, depending on the kind of feature interaction, specifications are formulated more or less explicitly. The requirement that a system does not crash with a segmentation fault is very general and is often implicitly assumed; the same applies to other properties, such as the absence of null-pointer dereferences and data races. However, in other situations, specifications are much more explicitly formulated, using formalisms such as temporal logic or automata, for example, stating that a certain process terminates before another process or that adaptive cruise control does not disable the break system [18].

Specifications are essential for the endeavor to understand feature interactions, as we explain next when discussing two properties along which we classify feature interactions.

2.2 Model of Feature Interactions

Given a set of features $\mathcal{F}$, predicate $interact_i^{\phi}(f_1, \ldots, f_n)$ denotes a feature interaction i that occurs in the subset $\{f_1, \ldots, f_n\} \subseteq \mathcal{F}$ of features with respect to specification ϕ, and that the feature set is *minimal*—removing one feature from the set would deactivate interaction i. Accordingly, the only derivation rule for $interact$ is:

$$\frac{f_1 \bullet \ldots \bullet f_n \not\models \phi \qquad n > 1}{\forall \{g_1, \ldots, g_k\} \subset \{f_1, \ldots, f_n\} \ : \ \neg interact_i^{\phi}(g_1, \ldots, g_k)}{interact_i^{\phi}(f_1, \ldots, f_n)}$$

$$(2)$$

This inference rule applies only to feature sets with two or more features, and it does not rule out that a given feature combination may contain multiple feature interactions of the same or different orders. This is in line with Siegmund et al., who found that the presence of higher-order interactions often implies the presence of corresponding lower-order interactions [48], which we model as distinct interactions $i_1, \ldots, i_m$.

As a design decision, we model interactions as violations of specifications. If features interact and satisfy a given specification, predicate $interact$ does not hold. To model such desired interactions, one can adapt the specification such that it exposes the interaction—a case that we ignore, for simplicity.

Furthermore, our definition of feature interactions is centered around the presence of features, which is natural when reasoning about feature-oriented systems. Work on interaction testing also considers interactions that occur when one feature must be selected and another must be deselected [21, 56], which we do not consider for simplicity.

Finally, we do not consider constraints among features (e.g., that one feature requires or excludes another feature). Although constrains play an important role in modeling, managing, and analyzing variability [17], including them into our model would make it more complicated, but does not add anything substantial to our message. See Garvin et al. [21] and Siegmund et al. [48] for examples of how constraints are incorporated in modeling and detecting feature interactions.

Inspired by Batory et al. [8], we denote an interaction between the n features $f_1, \ldots, f_n$ with the shorthand $f_1 \# \ldots \# f_n$.

2.3 Order of Feature Interactions

The *order* of a feature interaction $f_1 \# \ldots \# f_n$ is the number n of participating features minus one:

$$order(f_1 \# \ldots \# f_n) = n - 1 \qquad (3)$$

An interaction between two features has order one (i.e., is of first-order); an interaction between three features has order two (i.e., is of second-order); and so on. [1]

The number of potential interactions in a system can be exponential in the number of its set $\mathcal{F}$ of features:

$$2^{|\mathcal{F}|} - |\mathcal{F}| - 1 \in \mathcal{O}(2^{|\mathcal{F}|}) \qquad (4)$$

Fortunately, the situation is not that bad in practice. The number of actual feature interactions that occur is likely to be much lower [15, 27, 49]—otherwise feature-based systems would not be practical. Motivated by this assumption, some researchers focus only of interactions between pairs of features, which is much more tractable:

$$\frac{|\mathcal{F}| \cdot (|\mathcal{F}| - 1)}{2} \in \mathcal{O}(|\mathcal{F}|^2) \qquad (5)$$

Note that Equation 4 and Equation 5 are approximations in the sense that they ignore that a given set of features may give rise to multiple different interactions of the same order. Still, they illustrate the nature of the feature-interaction problem very well.

A major problem in practice is that, for a given system and specification, it is not obvious which feature interactions *really* occur. Intuitively, the higher the order of a feature interaction, the harder the interaction is to detect. Feature interactions of order one (or two) can be simply detected by creating all pairs (or triples) of features [7, 15, 27] and applying a proper interaction-detection technique (e.g., testing [21, 27, 28] or model checking [5, 7]). But, this way, one may miss interactions of a higher order—these can be found *reliably* only by creating all possible feature combinations, which induces, again, an exponential effort.

The tradeoff between the ability to find feature interactions of higher orders and the computational effort to achieve this raises the question of how the order of feature interactions are actually distributed in practice, which we discuss in Section 3.

2.4 Visibility of Feature Interactions

Different levels of visibility of feature interactions have been discussed in the literature. Feature interactions may appear at the level of the externally-visible behavior, which we call henceforth *external feature interactions*, for short, and at the level of the internal properties of a system, which we call henceforth *internal feature interactions*, for short.

2.4.1 External Feature Interactions

Feature interactions, as such, have been described first in the domain of telecommunication systems [11]. There, feature interactions have been described as inadvertent deviations from the expected externally-visible behavior of a system. Basically, the behavior of a system composed of features is more (or less) than the sum of the (well-known and well-defined) behaviors of the individual features involved.

This behavior-centric view requires to specify which behaviors are expected and desired (and which are not) and how behaviors relate. Does a behavior subsume another behavior? Does a behavior violate a specification? Is the combination of two behaviors more than the sum of its constituents (however 'sum' is defined)? Answering these questions is notoriously difficult. As we mentioned previously, the feature-interaction community has been developing a whole tool set of formalisms, methods, and tools for answering these and other questions.

Functional Interactions. Interestingly, two lines of research on external feature interactions emerged in the recent years. One line of research is concerned mainly with interactions that violate the *functional specification* of a composed system, which includes all kinds of bugs, including segmentation faults, race conditions, and deadlocks. We call these interactions *functional* feature interactions. [2]

Consider Hall's e-mail system with features for message encryption and forwarding as an example [23]: While encryption and forwarding operate individually as expected, their combination gives rise to an undesired feature interaction. The interaction occurs if one host sends an encrypted message to a second host that forwards the message automatically to a third host. If the second host does not have the public key of the third host, it forwards the message in plain text. The reason is that the forwarding feature has been developed independently of the encryption feature, so it does not "know" whether an e-mail is encrypted. This interaction is clearly undesired: it contradicts what we expect from the encryption feature, and it violates the specification of the encryption feature (if there is one), which states that messages that have been encrypted initially must never be sent unencrypted over the network.

Finding feature interactions that violate the functional specification of a composed system boils down to combining analysis techniques, such as testing, static analysis, and model checking, with strategies to reduce the analysis effort in the face of feature combinatorics (e.g., sampling, feature-based and variability-aware analyses). In the e-mail example, one could create (a subset of) feature combinations and analyze whether messages are sent unintentionally in plain text over the network using the following temporal-logic specification [6]:

$$\mathbf{AG}\, \big(\mathsf{recv}(\mathsf{msg}\ m) \wedge m.\mathsf{isEncrypted} \big) \Rightarrow \\ \big((\mathsf{send}(\mathsf{msg}\ m) \Rightarrow m.\mathsf{isEncrypted})\ \mathbf{R}\ \mathsf{send}(\mathsf{msg}\ m) \big) \qquad (6)$$

This specification states essentially that all incoming messages (recv) that were encrypted (isEncrypted) must be encrypted when leaving the system (send).

Non-Functional Interactions. Another line of research is concerned with interactions that influence *non-functional properties* of a composed system, including performance, memory consumption, energy consumption, etc. We call these interactions *non-functional* feature interactions. Non-functional feature interactions have been discussed in the literature with regard to explicit and implicit specification [46, 48, 54]. If we have an explicit specification of the desired non-functional properties of a system at hand (e.g., the maximum latency), we can typically decide whether a given feature combination satisfies the specification (e.g., whether it is fast enough).

If we do not have a specification at hand, it is still useful to reason about non-functional feature interactions. An assumption that guides work on the prediction of non-functional properties [48, 49] is that each feature has an influence on the non-functional properties of a system and that this influence can be quantified. Features are considered not to interact, if their contributions to a given non-functional property can be simply aggregated (e.g., by adding their execution times or taking the maximum peak performance). This statement is actually an implicit specification that serves to detect feature interactions, and to make predictions more accurate [48]. In this sense, feature are considered to interact, if a non-functional property of the composed system diverges from the aggregation of

---

[1]Some researchers call interactions between n features also n-way feature interactions [32].

[2]The concept of *interaction faults* used in the interaction-testing community is very similar [21, 28, 33].

the individual contributions of the features involved, for example, in that the performance goes substantially down.

For example, many features in a database system can be freely combined to tailor the system to the specific needs of a customer or application scenario, including encryption, compression, and various kinds of index structures and locking strategies [47]. However, there are subtle feature interactions that lead to performance abnormalities, for example, when a coarse-grained locking strategy hinders query evaluation and optimization [41].

Detecting non-functional feature interactions is, at least, as challenging as detecting function feature interactions. Typically, various techniques for the measurement, prediction, and modeling of non-functional properties are combined with strategies to reduce the analysis effort in the face of feature combinatorics [48]. For example, if we measure the performance of a database system with and without encryption and with and without compression, we will notice that these two feature interact: encrypting compressed data is computationally less expense than encrypting uncompressed data.

2.4.2 Internal Feature Interactions

Beside the behavior-centric view, researchers have proposed to take an implementation-centric view, which aims at the internals of a system, to understand the feature-interaction problem [8, 30, 40]. Specifications are given usually implicitly, as we will discuss.

Structural Interactions. It is a matter of fact that a feature is typically not an island; it communicates and cooperates with other features and the environment. In the end, the communication and cooperation among features needs to be implemented somewhere. To let features interact, we need corresponding *coordination code*. Interestingly, Batory et al. uses the $\#$ operator also to denote such coordination code when describing feature compositions [8]. For example, if we attempt to coordinate the call-forwarding and call-waiting features of the telephony example, we have to add additional code for this task (e.g., to deactivate one of the two features in favor of the other). If we activate both features in a system, we need to include also the corresponding coordination code:

$$CallForw \wedge CallWait \Rightarrow CallForw \# CallWait \qquad (7)$$

Coordination code breaks feature modularity and hinders compositional reasoning [29]. But, there is more to this. Much like with external feature interactions at the behavioral level, in the worst case, the number of pieces of coordination code grows exponentially with the number of features. Although researchers have proposed and discussed a number of solutions, there is no "silver bullet" to this problem [30]. The problem becomes even more problematic when all interacting features are supposed to be independently selectable or activatable by the user [30].

The key observation that is important here is that coordination code gives rise to a *structural* feature interaction. Features are considered to interact structurally if some coordination code is necessary that is different from the combination of the code of the individual features involved [8, 30, 40].

In many cases, structural feature interactions can be easily identified statically (e.g., based on naming or coding conventions, code-nesting structure, feature-tracing approaches, or dedicated implementation techniques [30, 40, 45]). As an example, in practice, the presence of coordination code is often controlled by nested preprocessor directives [37] or dedicated glue-code modules [30], such as lifters in feature-oriented programming [45] and connector plugins in ECLIPSE.

Operational Interactions. Apart from just analyzing the code base and searching for coordination code that gives rise to structural interactions, one can collect more detailed information on internal feature interactions by analyzing the execution or operation of a system. Which features refer to which other features? Which features pass control to which other features? Which features pass data to which other features? This information on *operational* interactions cannot be easily extracted from just looking syntactically at the source code, but requires more sophisticated (static or dynamic) analyses of the control and data flow. These analyses may provide valuable insights. For example, if we find that a contact-management and a messaging feature in an office groupware interact at the level of the control flow, but not at the level of the data flow (i.e., they do not exchange or share any data, even not via other features), we can infer that private contact data will not be send via the messaging feature to an untrusted receiver. This kind of information would help to make analysis techniques smarter and more efficient, as we will discuss in the next section.

Features are considered to interact operationally, if the occurrence of specific control and data flows, diverges from the combination of the flows of the individual features involved [8, 30, 40]. For example, two feature interact at the level of the control flow if there are control flows that occur *only* when the two features are combined, and that are not just the addition or union of the control flows induced by the two individual features.

3. EXPLORING FEATURE INTERACTIONS

Although some researchers studied feature interactions in real-world systems empirically [21, 28, 30, 33, 37, 48, 56], the community does not yet entirely understand the nature of feature interactions with all its facets, in depth and breadth. In particular, we are interested in the following research question:

> Which kinds and how many feature interactions occur in real-world systems, in terms of order and visibility?

We argue that answering this question is imperative to make progress in developing and evaluating general and efficient solutions to the feature-interaction problem. In the remaining section, we outline the requirements and challenges of an exploratory study to answer this question, we report on preliminary results, and we discuss perspectives of further research on feature interactions.

3.1 Exploratory Study

At first glance, the setup of an exploratory study that answers our research question is straightforward. Just take a representative set of systems, identify all feature interactions, and collect statistics about the number and distribution of their order and visibility. However, taking a closer look, there are many challenges, such as how to select representative systems among the multitude of domains (e.g., Internet applications vs. automotive systems vs. software product lines), the problem of obtaining and defining proper specifications, or the fact that some interactions are notoriously hard to discover, even if we have the minimal set of interacting features at hand (e.g., revealing a feature interaction based on a specification may be undecidable; or measuring non-functional feature interactions has to take measurement bias into account).

We concentrate on the most severe challenge for the envisioned exploratory study: The key problem is that detecting all feature interactions in a given system (according to a set of specifications) is often computationally infeasible. This was exactly the reason for why we need to develop better detection methods, for which we

Table 1: Overview of the subject systems

	$|\mathcal{F}|$	LOC	Description
LINUX[1]	9 102	5 986 427	Operating-system kernel, Version 2.6.28.7
BUSYBOX[2]	792	191 615	Standard UNIX utilities, Version 1.18.5
GCC[3]	171	2 648 177	GNU compiler collection, Version 4.4.0
APACHE[4]	9	230 277	Web server, Version 2.2

[1] http://www.kernel.org
[2] http://www.busybox.net/
[3] http://gcc.gnu.org/
[4] http://httpd.apache.org/

need the data set of the envisioned exploratory study—a chicken-and-egg problem.

Certainly, it is possible to accumulate a considerable data set based on smaller systems, for which computing and analyzing all feature combinations is feasible. Such systems have been used in several case studies [7, 21, 28, 30, 33, 37, 48, 56], so that they can form the basis of the data set. Still, to establish a representative data set, also information on larger systems needs to be included. The only way to cut the Gordian knot, is to apply a whole array of different detection approaches based on different sampling strategies, heuristics, and random choices to accumulate information on feature interactions (including the combinations that have been analyzed to reveal them), and to utilize this information to tune the detection approaches and strategies to find further interactions, and so on. But, one word of caution: Analyzing a set of mature and deployed systems tells only half the story. At later development stages, many feature interactions may have been resolved (or properly controlled) already, which might bias our conclusions. Hence, also earlier development stages of the systems under study should be analyzed.

In the face of this huge endeavor, it is certainly reasonable to expect many studies by many researchers, rather than one, and some studies have already been conducted whose results can be used right away [7, 21, 28, 30, 33, 37, 48, 56]. But, to be successful, we need to progress systematically, guided by a classification and model like the one we propose or similar ones. This paper is meant as a call to the community to start the endeavor and to overcome the feature-interaction challenge.

3.2 Preliminary Results

To illustrate the kind and usefulness of the data obtained by the envisioned exploratory study, we report on some of our preliminary results. As subjects, we selected four real-world systems that have been used before in the literature: LINUX, BUSYBOX, GCC, and APACHE. In Table 1, we summarize relevant background information on the subject systems.

In our exploratory study, we collected information on different kinds of feature interactions, one kind per subject system:

Structural interactions in LINUX**:** With the help of Jörg Liebig and his code-analysis tool CPPSTATS [37], we analyzed the nesting structure of C-preprocessor directives in the LINUX kernel. In particular, we determined the presence condition of each code block. Each (syntactically) unique presence condition represents a structural feature interaction of n-th

order, where n is the number of unique macro names in the presence condition.[3]

Operational interactions in BUSYBOX**:** Using the analysis tool TYPECHEF [38], we determined which control flows in BUSYBOX occur in which feature combinations. Fortunately, BUSYBOX provides a variability model that lists all configurable features and their dependencies. Each control flow across method boundaries that occurs only when, at least, n features are selected represents an operational feature interaction of order $n - 1$.

Functional interactions in GCC**:** We obtained information on the functional feature interactions in GCC from a previously published study on interaction testing [21]. The authors of the study analyzed which of the faults reported for GCC are configuration faults, that is, faults that arise only with certain combinations of command-line parameters. The minimal number of participating parameters determines the order of the corresponding functional feature interaction (minus one).

Non-functional interactions in APACHE**:** Based on the data set of Siegmund et al. [48], we computed all feature interactions that have an effect on performance (i.e., all combinations of features whose performance is not the sum of the contributions to performance of the individual features involved). In this study, we considered a subset of APACHE's plugins (i.e., loadable modules) as features. The number of plugins that exhibit a (measurable) performance interaction determines the order of this non-functional feature interaction (minus one).

In Figure 1, we show the results of our analysis of the four subject systems. Notice that the y-axes of Figure 1a and 1b are in logarithmic scale, to enable us to show the broad spectrum of interaction-order frequencies. Looking at the results, we make the following observations:

O_1**:** Feature interactions occur in all systems, and we found interactions of all kinds, including structural, operational, functional, and non-functional interactions.

O_2**:** Higher-order interactions (i.e., with an order larger than one) occur in all subject systems, many with orders around 2 to 10, up to an order of 33 (in LINUX).

O_3**:** The major fraction of feature interactions we found have orders lower than 10.

O_4**:** We found many more internal feature interactions than external feature interactions, and external interactions of higher orders are rare in the subject systems.

O_5**:** Although all distributions are right-skewed, there is no obvious strong correlation between them, which is not too surprising as the distributions have been obtained from different subject systems.

We believe that this kind of observations can have an influence on the judgment of contemporary approaches as well as on the development of further work. Nevertheless, the results presented here are not meant to be generalizable. Still, we carefully discus some of the observations next, in the context of perspectives for feature-interaction research.

---

[3] We are aware that not all macros correspond to features that are visible to the end user. In this sense, we deliberately take also internal variability into account.
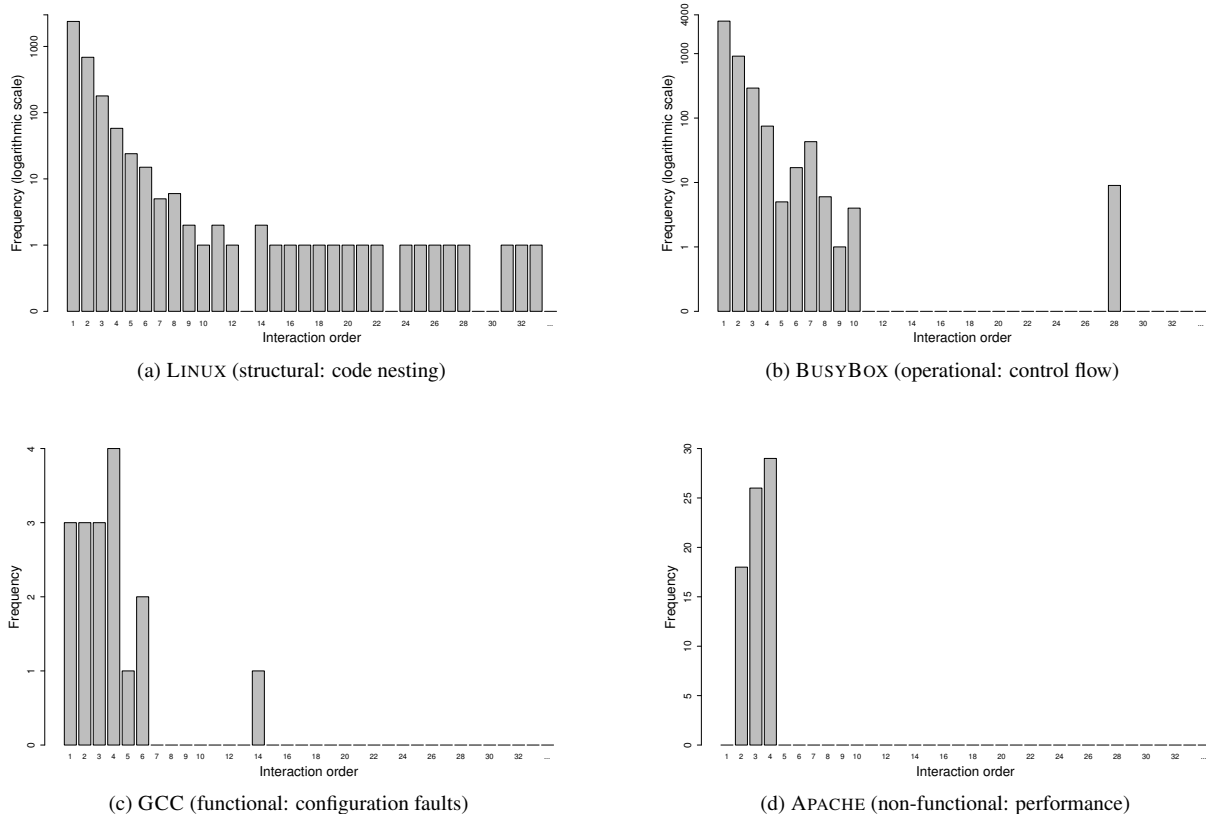
(a) LINUX (structural: code nesting)

(b) BUSYBOX (operational: control flow)

(c) GCC (functional: configuration faults)

(d) APACHE (non-functional: performance)

Figure 1: Distribution of different kinds feature interactions in four subject systems

3.3 Perspectives

As the data set collected by the envisioned exploratory study grows, it will form, more and more, a viable basis for further research on feature interactions. Next, we will discuss some promising directions.

Information on the order of feature interactions that occur in real-world systems is very valuable for tuning existing approaches of interaction detection, management, and resolution. For example, in feature-interaction detection, it is best practice to assume that a major fraction of feature interactions is first order. This assumption is the basis for several sampling-based analysis techniques that aim at covering all pairs of features, but disregard larger feature combinations [15, 27, 49]. If it turns out that this assumption misses the point—and this is what our preliminary results suggest—this would have definitive implications for the precision of the approaches and the development of alternative approaches. Also, work on feature-based [36] and variability-aware analyses [50], would benefit from knowledge on the distribution of feature-interaction orders (e.g., finding a considerable number of higher-order feature interactions in real-world systems, as it was the case in some of our subject systems, would be a strong argument for variability-aware analyses and against sampling; or search strategies and precision levels could be tailored to certain kinds of interactions).

Another interesting issue arises from the observation that different kinds of feature interactions are differently difficult to detect. For example, internal feature interactions, such as control-flow interactions, can be detected statically. Detecting external interactions, such as performance interactions, requires more heavyweight and often dynamic analysis techniques. It would be very interesting to see whether the occurrences of different kinds of feature interactions correlate. For example, is there a correlation between the occurrences of control-flow or data-flow interactions and performance interactions? It would be even more interesting, if we found that the occurrences of internal feature interactions have a certain power to predict external feature interactions. For example, can we predict the occurrences of memory-consumption interactions based on data-flow interactions with reasonable accuracy?

4. CONCLUSION

The feature-interaction problem is a major threat to feature modularity and compositional reasoning. Although being addressed by researchers and practitioners over years, there is still a lack of comprehensive empirical data on which kinds of feature interactions occur in real-world systems and to what extent. In this position paper, we set out the goal to explore the nature of feature interactions in the wild, in particular, in terms of their order and visibility.

A characterization and classification of feature interactions and a set of preliminary data obtained from four subject systems give us confidence that knowledge on the distribution of feature-interaction orders and visibilities will help us to answer a number of important questions: Which strategies for interaction detection, management, and resolution are superior in which circumstances? Do different kinds of feature interactions correlate? Can we predict feature interactions of one kind based on the knowledge about feature interactions of another kind?

However, as an analysis of possible experimental setups and corresponding challenges reveals, establishing a comprehensive overview and gaining deeper insights about the nature of feature inter-

actions will be a huge endeavor. To this end, the paper is meant as a call for contributions of the community to accept and address the feature-interaction challenge.

Acknowledgments

5. REFERENCES

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

[2] S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology (JOT)*, 8(5):49–84, 2009.

[3] S. Apel, T. Leich, and G. Saake. Aspectual feature modules. *IEEE Transactions on Software Engineering (TSE)*, 34(2):162–180, 2008.

[4] S. Apel, W. Scholz, C. Lengauer, and C. Kästner. Detecting dependences and interactions in feature-oriented design. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 161–170. IEEE, 2010.

[5] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Detection of feature interactions using feature-aware verification. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 372–375. IEEE, 2011.

[6] S. Apel, A. von Rhein, T. Thüm, and C. Kästner. Feature-interaction detection based on feature-based specifications. *Computer Networks*, 57(12):2399–2409, 2013.

[7] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for product-line verification: Case studies and experiments. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 482–491. IEEE, 2013.

[8] D. Batory, P. Höfner, and J. Kim. Feature interactions, products, and composition. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 13–22. ACM, 2011.

[9] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6):355–371, 2004.

[10] 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.

[11] T. Bowen, F. Dworack, C. Chow, N. Griffeth, G. Herman, and Y.-J. Lin. The feature interaction problem in telecommunications systems. In *Proceedings of the International Conference on Software Engineering for Telecommunication Switching Systems (SETSS)*, pages 59–62. IEEE, 1989.

[12] G. Bruns. Foundations for features. In *Feature Interactions in Telecommunications and Software Systems VIII*, pages 3–11. IOS Press, 2005.

[13] G. Bruns, P. Mataga, and I. Sutherland. Features as service transformers. In *Feature Interactions in Telecommunications Systems V*, pages 85–97. IOS Press, 1998.

[14] 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.

[15] M. Calder and A. Miller. Feature interaction detection by pairwise analysis of LTL properties: A case study. *Formal Methods in System Design*, 28(3):213–261, 2006.

[16] R. Crespo, M. Carvalho, and L. Logrippo. Distributed resolution of feature interactions for Internet applications. *Computer Networks*, 51(2):382–397, 2007.

[17] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[18] A. Dominguez. *Detection of Feature Interactions in Automotive Active Safety Features*. PhD thesis, University of Waterloo, 2012.

[19] R. Donaldson and M. Calder. Modular modelling of signalling pathways and their cross-talk. *Theoretical Computer Science*, 456(0):30–50, 2012.

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

[21] B. Garvin and M. Cohen. Feature interaction faults revisited: An exploratory study. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 90–99. IEEE, 2011.

[22] 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.

[23] R. Hall. Fundamental nonmodularity in electronic mail. *Automated Software Engineering*, 12(1):41–79, 2005.

[24] T. Härder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, 1983.

[25] J. Hay and J. Atlee. Composing features and resolving interactions. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, pages 110–119. ACM, 2000.

[26] 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.

[27] P. Jayaraman, J. Whittle, A. Elkhodary, and H. Gomaa. Model composition in product lines and feature interaction detection using critical pair analysis. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, LNCS 4735, pages 151–165. Springer, 2007.

[28] M. Johansen, Ø. Haugen, F. Fleurey, E. Carlson, J. Endresen, and T. Wien. A technique for agile and automatic interaction testing for product lines. In *Testing Software and Systems*, LNCS 7641, pages 39–54. Springer, 2012.

[29] C. Kästner, S. Apel, and K. Ostermann. The road to feature modularity? In *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD)*, pages 5:1–5:8. ACM, 2011.

[30] C. Kästner, S. Apel, S. ur Rahman, M. Rosenmüller, D. Batory, and G. Saake. On the impact of the optional feature problem: Analysis and case studies. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 181–190. Software Engineering Institute, 2009.

[31] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, LNCS 1241, pages 220–242. Springer, 1997.

[32] C. Kim, C. Kästner, and D. Batory. On the modularity of feature interactions. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 23–34. ACM, 2008.

[33] D. Kuhn, D. Wallace, and A. Gallo, Jr. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering (TSE)*, 30(6):418–421, 2004.

[34] K. Lee, K. Kang, M. Kim, and S. Park. Combining feature-oriented analysis and aspect-oriented programming for product line asset development. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 103–112. IEEE, 2006.

[35] C. Lengauer and S. Apel. Feature-oriented system design and engineering. *International Journal of Software and Informatics (IJSI)*, 5(1–2, Part II):231–244, 2011. Special Issue on Foundations and Practice of Systems and Software Engineering, Festschrift in Honor of Manfred Broy.

[36] H. Li, S. Krishnamurthi, and K. Fisler. Verifying Cross-Cutting Features as Open Systems. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, pages 89–98. ACM, 2002.

[37] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 105–114. ACM, 2010.

[38] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable analysis of variable software. In *Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 81–91. ACM, 2013.

[39] 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.

[40] J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 112–121. ACM, 2006.

[41] C. Mohan. Interactions between query optimization and concurrency control. In *Proceedings of the International Workshop on Research Issues on Data Engineering: Transaction and Query Processing (RIDE-TQP)*, pages 26–35. IEEE, 1992.

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

[43] K. Ostermann, P. Giarrusso, C. Kästner, and T. Rendel. Revisiting information hiding: Reflections on classical and nonclassical modularity. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, LNCS 6813, pages 155–178, 2011.

[44] K. Pomakis and J. Atlee. Reachability analysis of feature interactions: A progress report. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 216–223. ACM, 1996.

[45] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, LNCS 1241, pages 419–443. Springer, 1997.

[46] T. Repasi, S. Giessl, and C. Prehofer. Using model-checking for the detection of non-functional feature interactions. In *Proceedings of the International Conference on Intelligent Engineering Systems (INES)*, pages 167–172. IEEE, 2012.

[47] M. Rosenmüller, S. Apel, T. Leich, and G. Saake. Tailor-made data management for embedded systems: A case study on Berkeley DB. *Data & Knowledge Engineering (DKE)*, 68(12):1493–1512, 2009.

[48] N. Siegmund, S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake. Predicting performance via automated feature-interaction detection. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 167–177. IEEE, 2012.

[49] 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 & Software Technology (IST)*, 55(3):491–507, 2013.

[50] T. Thüm, S. Apel, C. Kästner, M. Kuhlemann, I. Schaefer, and G. Saake. Analysis strategies for software product lines. Technical Report FIN-004-2012, University of Magdeburg, 2012.

[51] 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.

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

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

[54] M. Weiss and B. Esfandiari. On feature interactions among web services. *International Journal of Web Services Research (IJWSR)*, 2(4):22–47, 2005.

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

[56] C. Yilmaz, M. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering (TSE)*, 32(1):20–34, 2006.

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