Computer Networks 57 (2013) 2399–2409

Contents lists available at SciVerse ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

# Feature-interaction detection based on feature-based specifications

Sven Apel [a,*], Alexander von Rhein [a], Thomas Thüm [b], Christian Kästner [c]

[a] Department of Computer Science and Mathematics, University of Passau, Innstr. 33, 94032 Passau, Germany
[b] School of Computer Science, University of Magdeburg, P.O. Box 4120, 39016 Magdeburg, Germany
[c] Institute for Software Research, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA

ARTICLE INFO

*Article history:*
Available online 16 April 2013

*Keywords:*
Feature orientation
Feature interaction
Feature-based specification
Modularity
Software product lines

ABSTRACT

Formal specification and verification techniques have been used successfully to detect feature interactions. We investigate whether feature-based specifications can be used for this task. *Feature-based* specifications are a special class of specifications that aim at modularity in open-world, feature-oriented systems. The question we address is whether modularity of specifications impairs the ability to detect feature interactions, which cut across feature boundaries. In an exploratory study on 10 feature-oriented systems, we found that the majority of feature interactions could be detected based on feature-based specifications, but some specifications have not been modularized properly and require undesirable workarounds to modularization. Based on the study, we discuss the merits and limitations of feature-based specifications, as well as open issues and perspectives. A goal that underlies our work is to raise awareness of the importance and challenges of feature-based specification.



## 1. Introduction

Feature interactions are phenomena that have been studied for years in telecommunication systems [1]. A *feature interaction* occurs when a feature (i.e., an end-user-visible unit of behavior) influences the behavior of another feature in an unexpected way (e.g., regarding the expected control flow or visible behavior). Typically, the behavioral influence induced by a feature interaction cannot be easily deduced from the individual behaviors of the features involved [1]. The feature-interaction problem is to detect, manage, and resolve interactions among features.

Feature interactions are often desired and need to be planned and managed properly. For example, in a database system, the features for managing transactions and collecting statistics interact: Statistics are collected about transactions at runtime, and access to statistics data is done under the umbrella of transactional control. But feature interactions may be inadvertent and even critical. Consider the canonical example of a building-automation system with flood and fire control [2]. Fire control activates sprinklers when sensors detect a fire. Flood control cuts off water supply when water is detected on the floor. Individually, both features operate as desired, but they interact in an inadvertent and critical way: When a fire is detected, the fire-control feature activates sprinklers, the flood-control feature detects standing water, turns off the water main, and the building burns down.

Two converging developments, software product lines and software ecosystems, make the feature-interaction problem especially challenging. First, recent *product-line and generator technology* gives rise to systems with many features and even more valid feature combinations [3,4]. Systems such as the Linux kernel provide thousands of configuration options, of which myriads of system variants can be generated, only by selecting the desired features (with only 33 independent features, one can generate a

* Corresponding author. Tel.: +49 8515093225.
*E-mail address:* apel@uni-passau.de (S. Apel).

1389-1286/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.comnet.2013.02.025

distinct system variant for each person on the planet). Anticipating, detecting, understanding, and handling all possible feature interactions in such systems is a major challenge. Second, the advent of *software ecosystems* (further) shifts software-engineering practice to an *open world* [5]. Much like in the telecommunication systems studied in the early days of feature-interaction research, there is no central instance that oversees the development process or system operation of a software ecosystem. Ecosystems such as Android, Firefox, and Eclipse are composed of many features (plugins, extensions, services, apps), mostly developed and tested separately by different parties. The independent development of features and the composition and deployment at customer site worsens the feature-interaction problem, because desired interactions cannot be planned properly and undesired ones cannot easily be anticipated.

In the remainder of the paper, we abstract from the specifics of telecommunication systems, product lines, and software ecosystems, by building our discussion on the paradigm of *feature orientation* [4]. A key idea of feature orientation is to make features explicit in design and code. Decomposing the design and implementation of a system along its features into composable units (components, services, plugins, aspects) has four benefits:

- Features can be implemented *independently* of one another.
- Based on a user's feature selection, the corresponding implementation units can be composed *automatically* by a generator.
- Features and their implementations can be *reused* in several systems.
- Different feature combinations result in different system *variants*.

In the past, researchers and practitioners proposed various approaches to tackle the feature-interaction problem [1]. Here, we concentrate on approaches that leverage formal specification and verification techniques to detect inadvertent and undesired feature interactions in feature-oriented systems [1,6–12]. The idea is to verify that a certain feature combination satisfies a corresponding specification.

For example, in the building-automation system from above, the specification should state that, if flood and fire control are active, fire control has precedence over flood control. This example illustrates an interesting aspect of specification. The example specification is global and concerns the overall system behavior. Global specification is problematic as it impairs modularity. Even if all features are known in advance, it is desirable to specify and implement them modularly. This way, the amount of global knowledge and control is minimized and thus reduces complexity and dependencies among features. In an open world, it is even impossible to specify a system globally, as not all features are known (e.g., it is challenging to anticipate all plugins of Firefox or Eclipse).

Clearly, it is desirable and natural to specify a feature-oriented system in terms of the features it provides. In this scenario, each feature comes with an implementation in the form of a composable unit (component, plugin, aspect, etc.), and a specification that states which properties must hold when the feature is present in the system [7], called *feature-based specification* [13]. A feature-based specification refers only to its own implementation and the implementation of the features it imports and refers to (e.g., via function calls).

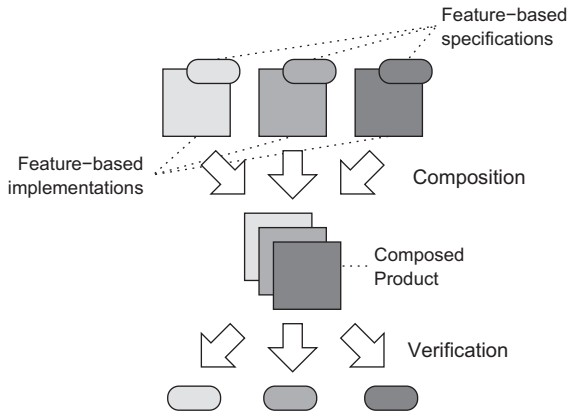


**Fig. 1.** Feature-interaction detection based on verification and feature-based specifications.

Having said that, readers may object that feature interactions are inherently anti-modular [1,14]—they emerge from the interplay of multiple features and cannot be detected by looking solely at individual features. So, techniques for feature-interaction detection have to consider (at some point) combinations of features (e.g., entire system variants). Nevertheless, the key research question we will address here is whether we can detect feature interactions based on feature-based specifications, as illustrated in Fig. 1. In other words, can we attain modularity at the level of specifications? In our example, we would need a specification of the desired behavior of the fire-control feature without referring to other independently developed features such as flood control.

In particular, we make two contributions to the state of the art:

- We devise the concept of feature-based specifications, and raise awareness of their importance and relation to the feature-interaction problem.
- We report on our experience with feature-based specification in the verification of 10 feature-oriented systems, and we discuss the merits and limitations of feature-based specification, as well as open issues and perspectives.

The focus we set in this paper is on feature interactions in *software systems*. In Section 3.4, we discuss whether and how our results generalize to other kinds of systems such as networked systems.

## 2. Feature orientation and feature interactions

A feature-oriented system is decomposed into composable units along the features it provides [4]. A key idea of

product-line engineering is to compose features in different combinations giving rise to different system variants (variants, for short). The resulting variants share certain features but differ in other features. A product line is a set of system (or product) variants that can be created by a set of features.

### 2.1. Basic concepts

Throughout the paper, we use a number of metavariables denoting basic concepts of feature-oriented systems: $F$ is the set of features, $f_1,\ldots,f_n \in F$ denote individual features, $V \subseteq 2^F$ is the set of valid system variants (i.e., a product line), and $v_1,\ldots,v_m \in V$ denote individual system variants, with $v_i \subseteq F$. In product-line engineering, typically, a feature model defines which system variants are valid [3], but details about this are not relevant for the remaining sections.

A unique property of feature-oriented systems is that features are explicit in design and code. That is, ideally, for each feature there is one composition unit that implements the feature. This way, a system variant can be generated based on a selection of desired features solely by composing the corresponding implementation units: $impl(f)$ denotes the implementation unit of feature $f$, and $impl(v)$ denotes the composition of the features (i.e., of their implementation units) of variant $v$:

$$impl(v) = impl(f_1) \bullet \ldots \bullet impl(f_n) \tag{1}$$

where $f_1,\ldots,f_n$ are the features of variant $v$ (with $v = \{f_1,\ldots,f_n\}$), and $\bullet: I \times I \to I$ is the composition operator over the set $I$ of valid implementation units.

Typically, feature implementations depend (or build) on other feature implementations. Independently, of the implementation language, we say in this case that the implementation of one feature *imports* the implementation (or parts of it) from another feature (e.g., via import declarations in Java or #include directives in C).

Several composition operators for feature implementations have been proposed, including component aggregation, service orchestration, plugin mechanisms, aspect weaving, and superimposition [3,4,15,16]. For our discussion, the choices of programming language, import mechanism, and composition operator are not relevant, as long they allow the programmer to implement a feature within a single implementation unit and to compose multiple units upon feature selection. The mapping between features and implementation units may be more complex [17,18]; for now, we suffice with a simple one-to-one mapping.

**Example.** For illustration, we use the running example of a simple, feature-oriented e-mail client [9], inspired by Hall [14], including optional support for message encryption and forwarding. For the purpose of the example, features are implemented in C, in terms of feature modules, and composed by means of superimposition [19], as illustrated in Fig. 2. Feature *EmailClient* implements a basic e-mail client; it introduces a structure email for representing e-mails and the two functions outgoing and incoming for handling incoming and outgoing e-mails. Feature *Encrypt* encrypts outgoing e-mails; it extends the existing structure email by adding the two new fields isEncrypted and encryptionKey and function encrypt, and it overrides the existing function outgoing to actually encrypt outgoing e-mails (keyword original invokes the overridden function). Feature *Forward* forwards incoming e-mails to another host; it introduces a function forward and overrides the existing function incoming to forward incoming e-mails.

Once a user selects a set of desired features, superimposition merges the code of all feature modules recursively based on nominal and structural similarity. If two features contain structures or classes with equal names, their members are superimposed, as well; see Fig. 2, for an example. This process recurses until basic program elements such as fields and functions are reached. Typically, there is a given total order over features, because feature composition is not commutative [19]. In our case studies, we use the tool FeatureHouse [20] for superimposition.

### 2.2. Feature-oriented specification

To verify the correctness of a feature-oriented system, one has to prove that its implementation satisfies its specification. The question we raised in the introduction was whether feature-based specifications can be used to detect feature interactions. To this end, we distinguish between three types of specifications [13]:

- A *global specification* $spec(V)$ defines the properties of each system variant of a product line $V$.
- A *variant-based specification* $spec(v)$ defines the properties of a single system variant $v$.
- A *feature-based specification* $spec(f)$ defines the properties of feature $f$ that apply to all variants that contain $f$.

**Example.** In our e-mail example, a possible global specification is that an outgoing e-mail message must have valid sender and receiver addresses. In temporal logic, we can express this global specification (which is essentially a safety property) as follows[1]:

$$\mathbf{AG}\ \text{outgoing}(\text{email } e) \Rightarrow valid_{addr}(e.\text{from}) \wedge valid_{addr}(e.\text{to}) \tag{2}$$

where $valid_{addr}$ is an auxiliary predicate stating the validity of a host address.

A feature-based specification that concerns only feature *Encrypt* is that the encryption key must be valid:

$$\mathbf{AG}\ \text{outgoing}(\text{email } e) \Rightarrow (e.\text{isEncrypted} \Rightarrow valid_{key}(e.\text{encryptionKey})) \tag{3}$$

where $valid_{key}$ is an auxiliary predicate stating the validity of an encryption key. The formula states that, whenever an outgoing e-mail message is sent, the corresponding encryption key must be valid. Note that, for system variants that do not contain feature *Encrypt*, this specification

[1] The combination of the two operators **A** and **G** states that the proposition must hold globally for all execution paths.

*impl*(*EMailClient*)

```
1  // representation of e-mail messages
2  struct email {
3    int id; char *from; char *to; char *subject; char *body;
4  };
5  // outgoing e-mails are processed by this function before they leave the system
6  void outgoing (struct client *client, struct email *msg) { ... }
7  // incoming e-mails reach the client at this point and are stored in a mailbox
8  void incoming (struct client *client, struct email *msg) { ... }
```

*impl*(*Encrypt*) imports *impl*(*EmailClient*)

```
9  // extending the e-mail structure by information on encryption
10 struct email {
11   int isEncrypted; char *encryptionKey;
12 };
13 // encrypt a given e-mail, if the public key of the receiver is known
14 void encrypt (struct client *client, struct email *msg) { ... }
15 // override function outgoing to encrypt e-mails before they are sent
16 void outgoing (struct client *client, struct email *msg) {
17   encrypt (client, msg);
18   original (client, msg); // invoke the overridden function
19 } ...
```

*impl*(*Forward*) imports *impl*(*EmailClient*)

```
20 // forward an e-mail to another host
21 void forward (struct client *client, struct email *msg) { ... }
22 // override function incoming to forward e-mails automatically
23 void incoming (struct client *client, struct email *msg) {
24   forward (client, msg);
25   original (client, msg); // invoke the overridden function
26 }
```

*impl*(*Forward*) • *impl*(*Encrypt*) • *impl*(*EmailClient*)

```
27 // basic e-mail structure, including information on encryption
28 struct email {
29   int id; char *from; char *to; char *subject; char *body;
30   int isEncrypted; char *encryptionKey;
31 };
32 // encrypt outgoing e-mails and send them
33 void outgoing (struct client *client, struct email *msg) {
34   encrypt (client, msg);
35   ... // original code of function outgoing of feature EmailClient
36 }
37 // handle incoming e-mails and forward them automatically
38 void incoming (struct client *client, struct email *msg) {
39   forward (client, msg);
40   ... // original code of function incoming of feature EmailClient
41 }
42 // encrypt a given e-mail, if the public key of the receiver is known
43 void encrypt (struct client *client, struct email *msg) { ... }
44 // forward an e-mail to another host
45 void forward (struct client *client, struct email *msg) { ... }
```

**Fig. 2.** Three feature implementations of the e-mail client in the form of feature modules, written in C [9] (*impl*(*EmailClient*), *impl*(*Encrypt*), *impl*(*Forward*)), and their composition using superimposition (*impl*(*Forward*) • *impl*(*Encrypt*) • *impl*(*EmailClient*)).

cannot even be checked, as it contains references to *Encrypt*'s implementation.

The three types of specification, give rise to three corresponding verification procedures for feature-interaction analysis of an entire product line $V$:

- Feature-interaction detection based on a global specification $spec(V)$:

$$\forall v \in V : \ impl(v) \models spec(V) \tag{4}$$

- Feature-interaction detection based on variant-based specifications $spec(v)$:

$$\forall v \in V : \ impl(v) \models spec(v) \tag{5}$$

- Feature-interaction detection based on feature-based specifications $spec(f)$:

$$\forall v \in V : \ \forall f \in v : \ impl(v) \models spec(f) \tag{6}$$

Using a global specification, a verification tool needs to check only a single specification, whereas the number of product-based specifications grows, in the worst case, exponentially with the number of features. In contrast, the number of feature-based specifications grows only linearly with the number of features.

Note that, in Eqs. (4)–(6), we verify every system variant individually. This strategy is naive and can be optimized in that individual features are verified separately as far as possible, or verification takes advantage of similarities among system variants; a recent survey describes several strategies in detail [13], but their differences are orthogonal to the problem of modularity of feature specifications. In the remainder of the paper, we assume the naive variant-based strategy, for simplicity.

Finally, as suggested in Eqs. (4)–(6), we can verify the implementations directly using proper verification technology—*without* the need of extracting intermediate models manually [9].

### 2.3. Modularity of feature-based specifications

What does modularity precisely mean to specifications in feature-oriented systems? The answer consists of two parts. First, rather than assigning a specification to one or more system variants (such as with global and variant-based specifications), using feature-based specifications, each feature has its own specification (which may be empty). Every system variant that contains the feature has to fulfill the feature's specification (cf. Eq. (6)).

But there is more to feature-based specification than only assigning specifications to individual features. It is certainly not the point of feature-based specification to formulate a global specification that concerns many features and system variants, and then assign it to a single feature. In a truly open world, when developing and integrating a feature, a programmer is not aware of other features, except certain features, whose implementation units are imported. A feature's specification has no global knowledge.

The *implementation base* $base_{impl}(f)$ of a given feature $f$ is determined by the imports of $f$:

$$base_{impl}(f) = \{f\} \cup \{g \mid imports(impl(f), impl(g))\} \tag{7}$$

where $imports(impl(f), impl(g))$ states that the implementation of feature $f$ imports the implementation (or parts of it) of feature $g$. The actual definition of *imports* depends on the underlying language and composition mechanism [21]. In our setting, imports are defined either explicitly via `import` or `#include` directives, or implicitly via references across feature boundaries (in the form of functions calls, field and global-variable accesses, and type references).

**Example.** From Fig. 2 we infer that $base_{impl}(Encrypt) = \{Encrypt, EmailClient\}$, because the implementation of feature *Encrypt* refers to the implementation of *EmailClient*. Similarly, we infer that $base_{impl}(Forward) = \{Forward, EmailClient\}$.

Likewise, a feature's specification may depend on the implementations *and* the specifications of other features, called its *specification base*:

$$base_{spec}(f) = \{f\} \cup \{g \mid imports(spec(f), spec(g)) \vee imports(spec(f), impl(g))\} \tag{8}$$

Based on Eqs. (7) and (8), we refine our notion feature-based specification: we call a specification of feature $f$ feature-based, iff $base_{spec}(f) \subseteq base_{impl}(f)$.

**Example.** In our e-mail example, the specification of feature *Encrypt* in Eq. (3) is feature-based: *Encrypt*'s specification refers only to its own implementation and the implementation of feature *EmailClient*, which belongs to its implementation base:

$$\underbrace{\{Encrypt, EmailClient\}}_{base_{spec}(Encrypt)} \subseteq \underbrace{\{Encrypt, EmailClient\}}_{base_{impl}(Encrypt)} \tag{9}$$

If the specification of feature *Encrypt* referred to feature *Forward*, it would not be feature-based anymore. In Section 2.4, we will discuss such an example in the context of feature-interaction detection.

### 2.4. Feature interactions

Specification and verification techniques have been used successfully for feature-interaction detection [1,6–10,12]. In a closed world, in which all features are known a priori, feature-interaction detection can take advantage of global or variant-based specifications, because each specification may import any other feature's specification or implementation, making desired and undesired interactions explicit (e.g., 'in all variants that contain both fire control and flood control, fire control takes precedence over flood control').

In an open world, not all features are known a priori. Here, global and variant-based specifications are not an option as they are necessarily incomplete. If a new feature is introduced, the results of analyses using global or variant-based specifications are obsolete. In an open world, feature-based specifications are desirable. But can we detect interactions between features that do not know of each other?

**Example.** Hall designed the e-mail system such that it actually contains a critical interaction between the features *Encrypt* and *Forward* (based on experience with interactions in real systems at AT&T) [14]. The interaction occurs if one host sends an encrypted e-mail to a second host that forwards the e-mail automatically to a third host. If the second host does not have the public key of the third host, it forwards the e-mail in plain text (the implementation of feature *Forward* does not know whether an e-mail is encrypted).[2]

Intuitively, the interaction between *Encrypt* and *Forward* violates the intention of feature *Encrypt*. But would it have been detected in an open world, using a feature-based specification? A straightforward specification would be the following[3]:

$$\mathbf{AG}\ (\mathsf{incoming}(\mathsf{email}\ e) \wedge e.\mathsf{isEncrypted}) \Rightarrow ((\mathsf{forward}(\mathsf{email}\ e) \Rightarrow e.\mathsf{isEncrypted})\ \mathbf{R}\ \mathsf{forward}(\mathsf{email}\ e)) \quad (10)$$

But this specification is not feature-based, because it refers to the implementation of feature *Forward*, which is not part of *Encrypt*'s implementation base (and possibly not known in an open world): $base_{spec}(Encrypt) \not\subseteq base_{impl}(Encrypt)$.

With this issue in mind, we can formulate a feature-based specification of feature *Encrypt*:

$$\mathbf{AG}\ (\mathsf{incoming}(\mathsf{email}\ e) \wedge e.\mathsf{isEncrypted}) \Rightarrow ((\mathsf{outgoing}(\mathsf{email}\ e) \Rightarrow e.\mathsf{isEncrypted})\ \mathbf{R}\ \mathsf{outgoing}(\mathsf{email}\ e)) \quad (11)$$

The difference to the first attempt is that the feature-based specification does not refer to function forward of feature *Forward*, but to the underlying implementation of feature *EmailClient*, so $base_{spec}(Encrypt) \subseteq base_{impl}(Encrypt)$.

Having such a modular specification of feature *Encrypt*, can we detect the critical interaction with feature *Forward*? For this example, the answer is yes, as we illustrate in Section 2.5).

### 2.5. Detection of feature interactions

There is a multitude of approaches to detect feature interactions. For illustration, we use the approach of *feature-aware verification* [9]: (1) we compose all feature modules of a product (or even product line), (2) we weave all specifications as assertions into the program code to signal undesired events and program states, and (3) using a model checker, we check whether the assertions can be reached in any execution path.

For example, assume we check the product that contains the features *EmailClient*, *Encrypt* and *Forward*, as shown in Fig. 2. As a use case, we consider the situation, in which a client Alice sends an e-mail to client Bob, and Bob has enabled forwarding to client Dan. Whether the original e-mail message is encrypted in the first place, and whether the clients have exchanged encryption keys is left open.

When running this use case in a model checker, it explores possible program executions until it arrives at a choice where the next step depends on the value of an unresolved decision (for example, whether the e-mail is encrypted). At this point, the model checker enumerates all possible choices and executes them sequentially. Once it has reached the end of the program in one path, it resumes exploring another path, until all paths have been fully explored or it reaches an assertion (see Fig. 3).

In our example, an assertion is reached when the model checker explores the path, in which the original e-mail is encrypted and Bob cannot encrypt it upon forwarding it to Dan (he does not have the correct key). Technically, the specification is encoded by adding a flag arrivedEncrypted to email and by including a corresponding assertion in the code of function outgoing.

## 3. Exploratory study

In the e-mail example, we are able to detect the feature interaction between the features *Encrypt* and *Forward* based on their modular, feature-based specifications. But is that generally possible? The underlying issue we want to address here is whether and to what extent feature-based specifications are capable of detecting feature interactions.

### 3.1. Research questions

In our exploratory study, we want to answer the following research questions:

- **RQ$_1$**: What fraction of specifications of feature-oriented systems are feature-based?
- **RQ$_2$**: Are feature-based specifications used to detect feature interactions?
- **RQ$_3$**: What are possible reasons for non-modularity of specifications?

### 3.2. Research method and sample systems

In order to answer our research questions, we conducted an exploratory study on the basis of 10 feature-oriented systems with existing specifications. Since not many such systems are publicly available, we included all systems we were able to locate (even when we found subsequently that some systems do not contain feature interactions):

- The **EMail** system of Hall [14] models an e-mail communication suite. The suite provides several features that can be activated or deactivated, for example, encryption, automatic forwarding, and e-mail signatures.
- The **Elevator** system has been developed by Plath and Ryan [22]. It is a model of an elevator that is extensible with various features such as stopping when the elevator is empty or priority service for a special floor.

[2] Recall, this is canonical example of a feature interaction inspired by experience with real-world systems [14]. Of course, one could alter the implementation such that the interaction disappears, but this is exactly the point: We want to find such inadvertent feature interactions, to resolve them.

[3] The operator **R** states that the proposition to the left must hold until (and including) the state described by the proposition to the right has been reached.

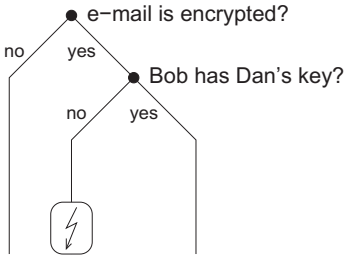


**Fig. 3.** Exploring possible program paths of the e-mail example using a model checker.

- The **MinePump** system is based on a work of Kramer et al. [23]. It simulates a water pump in a mining operation. The pump keeps the bottom of the mine shaft dry, but must be deactivated when the mine contains combustible methane gas. The system has several features that vary its behavior.
- The **CAN** system models a protocol for peer-to-peer networks [7]. It consists of basic routing mechanisms and of advanced features such as load measurement and the simulation of malicious peers.
- The **POSIX FS** system models a POSIX file system [7], including basic file-system operations such as move and create, and advanced features such as symbolic links and block management.
- The **BankAccount** system is a product line for bank-account management [10]. It models basic concepts such as accounts and users, as well as features for limiting withdrawals and for calculating interests and credit worthiness.
- The **DiGraph** system is a library for representing and manipulating directed graph structures [24]. Beside basic graphs, it supports various operations such as removal, traversal, and transposition, implemented as optional features.
- The **ExamDB** system is an exam database management system [24]. It manages exams to be passed by students, including features for subscription and backouting, bonus points, and statistics.
- The **Paycard** system is a smartcard-payment software that supports optionally transaction logging and statistics (e.g., operations failed and maximum record) [24].
- The **ListPL** system is a feature-oriented implementation of a list structure for storing integer values and corresponding basic operations [11]. Based on the feature selection, the list behaves like a stack or a sorted list.

Let us emphasize that all of the sample systems have been developed prior to and independently of our exploratory study. They have been used in prior studies to assess automated feature-interaction detection and product-line verification [7–11,14,22,24,25]. Only three systems (CAN, POSIX FS, and EMail) have been developed explicitly with feature-oriented specifications in mind [7,9].

Technically, the sample systems have been developed, either from scratch in a feature-oriented style, or by refactoring existing monolithic implementation into feature modules. EMail, Elevator, and MinePump are implemented in Java, and their specifications are given in terms of AspectJ [26] aspects that weave exceptions into the code (available at `fosd.net/FAV/`). CAN and POSIX FS have been implemented in FeatureAlloy [7], and their specifications are given in terms of assertions (available at `fosd.net/fh/`). The remaining five systems have been modeled and specified using the Java Modeling Language (JML) [27] (available at `spl2go.cs.ovgu.de/`).

For each sample system, we manually analyzed its specifications, and we determined which specifications were already feature-based. Furthermore, we consulted the systems' documentations as well as corresponding original publications to collect the feature interactions that have been detected and documented by the original authors. In the original systems, all feature interactions have been detected automatically, either model checking or deductive verification.

Of all known feature interactions, we determined how many feature interactions have been detected by the original authors using the existing feature-based specifications. We use the collected quantitative information to answer the research questions $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$.

Subsequently, we examined all specifications that were not feature-based in the original systems, and we analyzed whether we could rewrite the existing specifications (or corresponding implementations) to make them feature-based. To answer research question $\mathbf{RQ}_3$, we qualitatively assess possible reasons for non-modularity and discuss alternatives.

### 3.3. Results and discussion

In Table 1, we provide an overview of the quantitative results of our study. For each sample system, we show the overall number of features, system variants, specifications, and known interactions, as well as the number of feature-based specifications and interactions detected based on feature-based specifications.

Regarding the research questions $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$, the majority of all specifications of the original sample systems were feature-based (88%), and the majority of the known feature interactions of these systems could be detected using feature-based specifications (75%).

With regard to research question $\mathbf{RQ}_3$, a manual analysis of the remaining specifications that were not feature-based revealed that none of them was really global or variant-based. Every non-modular specification we found was concerned with exactly two features, and, in every case, the two features gave rise to a feature interaction.

For illustration, we discuss two examples taken from our study, one for a feature-based specification, and one for a specification that is not feature-based.

**Example** (*Feature-based*). Let us give an example of a specification that is feature-based. It is based on the elevator system of Plath and Ryan [22]. In its basic configuration, the system comes with several specifications defining the intended behavior of the elevator. One of these specifications is that "the elevator will not change

**Table 1**
Overview of the study results (|F|: number of feature; |V|: number of variants, specs: specifications; itns: interactions).

| Sample system | \|F\| | \|V\| | Specs | Known itns | Feature-based specs | Detected itns (feature-based) |
|---|---|---|---|---|---|---|
| EMail | 9 | 40 | 8 | 10 | 6 | 8 |
| Elevator | 6 | 20 | 8 | 8 | 8 | 8 |
| MinePump | 7 | 64 | 5 | 4 | 5 | 4 |
| CAN | 8 | 48 | 2 | 3 | 1 | 1 |
| POSIX FS | 8 | 64 | 10 | 4 | 7 | 1 |
| BankAccount | 6 | 24 | 5 | 0 | 5 | 0 |
| DiGraph | 4 | 8 | 4 | 0 | 4 | 0 |
| ExamDB | 4 | 8 | 4 | 0 | 4 | 0 |
| Paycard | 4 | 6 | 4 | 0 | 4 | 0 |
| ListPL | 5 | 16 | 6 | 3 | 5 | 2 |

direction while there are calls in the direction it is traveling" [22] (for simplicity, we consider only the downward direction):

$$\mathbf{AG}\ (\forall i > j:\ ((\text{floor} = i \wedge \text{elevBut.pressed}(j) \wedge \text{direction} = \text{down}) \Rightarrow \mathbf{A}\ (\text{direction} = \text{down}\ \mathbf{U}\ \text{floor} = j))) \quad (12)$$

In the basic elevator, this specification is not violated, but that may change when we add features that modify the elevator's behavior. To verify that a product satisfies the specification of Eq. (12), we encode it into the program code as an assertion that signals the undesired event or program state. Much like in Section 2.5, we can use a model checker to check whether the assertion can be reached.

Suppose we check the specification of Eq. (12) against the product that contains the features *BasicElevator* and *PriorityService* (gives priority to calls from the top floor). An assertion will be reached when the model checker explores the program path in which the elevator is going down, there is a call from a lower floor, and there is a priority call from the top floor. In this case, feature *PriorityService* will change the elevator direction ignoring the call from the lower floor, and the assertion will be triggered. Consequently, the model checker reports this assertion, which identifies the feature interaction.

The specification given in Eq. (12) is feature-based, because all referenced program elements (floor indicators, elevator buttons, and direction indicator) are implemented in the base system. Consequently, the specification can be verified solely with respect to the base system, and still it is useful to identify violations induced by other features that are unknown to the specification.

**Example** (*Not feature-based*). Let us give an example of a specification that is not feature-based. It is taken from the e-mail system and concerns the features *Verify* and *Encrypt*. In Fig. 4, we show the relevant parts of their implementations. In addition to the original e-mail example of Fig. 2, feature *Encrypt* introduces a function isReadable to indicate whether an e-mail is encrypted or plain text. Feature *Verify* implements support for signing e-mails and verifying signatures. To verify a signature, it computes a hash of the e-mail text. To this end, *Verify* needs the e-mail in plain text, which can be specified as follows:

$$\mathbf{AG}\ \text{verifySignature}(\text{email}\ e) \Rightarrow \text{isReadable}(e) \quad (13)$$

This specification is violated by an interaction between the features *Verify* and *Encrypt*. In this case, it may happen that the receiver of an e-mail message cannot decrypt a signed message (e.g., because of a missing or wrong key) and that thus the (possibly valid) signature cannot be verified. A model checker can detect this feature interaction by checking the reachability of assertions that signal this undesired behavior (a valid signature cannot be verified), much like in the example of Section 2.5.

The important point here is that, while the specification of feature *Verify* (Eq. (13)) can be used to detect the interaction between *Verify* and *Encrypt*, it is not feature-based! It refers to function isReadable of feature *Encrypt*, which is independent of feature *Verify* and not in the implementation base of *Verify*. To attain modularity, we could move function isReadable to feature *Verify* or to a common base feature (e.g., *EmailClient*), but this is clearly a hack, as from the point of the domain expert the function belongs to feature *Encrypt*.

We found that, in principle, we could transform all non-modular specifications to feature-based specifications: we could (1) move parts of the code of one feature to another feature that belongs to the implementation base of both interacting features or (2) we could create a new feature module to store the specification in question, with both interacting features as implementation base. However, these transformations would only mask the problem of broken modularity: we would end up with modules that contain entangled code that belongs to different, independent features. So, we conclude that there are indeed some feature interactions that cannot be detected properly by feature-based specifications (overall, 25% in our sample systems).

Another notable observation is that, in the JML sample systems, some specifications refine other specifications [13]. Much like C and Java code, specifications can be composed by means of superimposition. But, interestingly, this refinement relation between specifications does not impair modularity, if the refining specification invokes the refined specification via the keyword original (much like methods in the example of Fig. 2), as was always the case in our JML-based sample systems. The reason is that original does not require the presence of a particular feature. Merely it requires the presence of any feature that contains a corresponding contract, and a corresponding feature is always present in the implementation base (otherwise the product line would not be type safe [28]). So, specification

*impl*(*Encrypt*) imports *impl*(*EmailClient*)

```
1 ...
2 // signal whether an e-mail is readable (i.e., is plain text)
3 int isReadable(struct email * msg) { return !msg.isEncrypted; }
4 ...
```

*impl*(*Verify*) imports *impl*(*EmailClient*)

```
5 ...
6 // verify the signature of a given e-mail (needs to be plain text)
7 void verifySignature (struct email *msg) { ... }
8 ...
```

**Fig. 4.** Excerpts of the implementations of *Encrypt* and *Verify* of the e-mail system.

refinement seems to be useful to improve modularity of specifications for features, which shall be explored in further work.

### 3.4. Threats to validity

Next, we discuss threats to internal and external validity.

#### 3.4.1. Internal validity

Our study results are based on a set of sample systems. Especially, we rely on the correctness of the documentation of specifications and feature interactions provided by the original authors. As all have been documented in the original publications, and, due to the fact that we worked with most of the systems before, we consider this threat as minor.

#### 3.4.2. External validity

The selection of sample systems may threaten external validity. Do our findings apply to other kinds of systems, of other domains, specified and implemented with alternative languages? We cannot answer this question conclusively, but we aimed at controlling this threat in the context of our study goals. We deliberately limited our focus to feature-oriented systems (excluding networked systems), that have been developed and used for purposes other than our study, that contain specifications, and for which feature interactions are documented. Still, our set of sample systems covers multiple application domains and different specification and implementation languages. But, it is too small to investigate the influence of these different factors.

Another issue is the size of the samples systems. Do our findings apply to larger systems? We argue that feature-based specification scales as long as features can be implemented modularly. However, feature-interaction detection based on formal verification (e.g., model checking) may limit scalability, which is an orthogonal issue. Approaches based on testing seem to be appropriate in this case [29].

Finally, all sample systems contained feature interactions that concern the desired functional behavior. Although we see no reason for why feature-based specification of non-functional properties is not feasible, we cannot make verified statements about their ability to detect non-functional feature interactions [30].

## 4. Related work

The role of feature-based specifications in feature-oriented systems has been discussed before by Apel et al. [7]; we devised the concept, and analyzed and discussed its implications. Feature-based specifications have been used before for model checking [7,9,12] and deductive verification [10,31,32] of software product lines. Furthermore, Thüm et al. [24] discuss composition and refinement operators of feature-based specifications, and Poppleton [33] proposes a combination of feature-based and variant-based specification (which allows a programmer to specify each feature individually and to enrich the derived specification for every system variant). None of these papers actually explore the capability of feature-based specifications to detect feature interactions.

An interesting line of research aims at the modular verification of features [34–36]. The idea is to verify features of feature-oriented systems as far as possible in isolation. Properties of one feature that depend on the presence of other features are described in semantic interfaces, which are then used to discover feature interactions. This verification approach is promising to be combined with feature-based specifications. Other attempts to verify product lines and feature-oriented systems do not explicitly consider or discuss feature-based specifications in the light of feature-interaction detection, but could be combined with them [6,8,25].

A different line of research aims at the modularity of feature interactions in the sense that the code that coordinates the mutual behavior of interacting features is modularized [17,18,37,38]. Specification and feature-interaction detection is not considered explicitly.

Finally, there is a large body of research that investigates feature interactions in the telecommunications domain and networked systems [1,39,40]. While feature-based specifications are in these systems similarly relevant as in software systems, it is open whether our results can be generalized.

## 5. Conclusion

Feature-based specifications are appealing in an open world. The question we addressed was whether the modularity of feature-based specifications impairs the ability to

detect feature interactions. After introducing modularity to the specification of feature-oriented systems, we raised awareness of the potential strengths and weaknesses of feature-based specifications, compared to global and variant-based specifications. Based on this discussion, we conducted an exploratory study on 10 feature-oriented systems. In particular, we analyzed whether and how feature-based specifications have been used to detect feature interactions. We found that the majority of all specifications of the sample systems were feature-based (75%), and that the majority of the feature interactions in these systems could be detected using feature-based specifications (88%). That is, feature-based specifications are useful to detect feature interactions in feature-oriented systems, but in some cases they fail. This means that developers have to resort to undesirable workarounds such as moving code between features to regain modularity.

In summary, we can confirm the usefulness of feature-based specifications for feature-interaction detection, but recognize also their limitations. An underlying goal of this work was to raise awareness of the importance and challenges of feature-based specification, especially, in open feature-oriented systems. We believe that the trend in software-engineering practice will further go toward modularity, and feature-based specification will be an important foundation, on which approaches such as modular, feature-based verification will build. There are many open issues to address, including the role of specification and implementation languages used in the development of feature-oriented systems. There is also the question of whether we can come up with a classification of feature interactions that explains why some interactions are inherently anti-modular and require global specifications.

## Acknowledgements

Apel's and von Rhein's work is supported by the German Research Foundation (AP 206/2 and AP 206/4). Kästner's work has been supported by the European Research Council (ERC #203099).

## References

[1] M. Calder, M. Kolberg, E. Magill, S. Reiff-Marganiec, Feature interaction: a critical review and considered forecast, Computer Networks 41 (1) (2003) 115–141.
[2] K. Kang, J. Lee, P. Donohoe, Feature-oriented project line engineering, IEEE Software 19 (4) (2002) 58–65.
[3] K. Czarnecki, U. Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.
[4] S. Apel, C. Kästner, An overview of feature-oriented software development, Journal of Object Technology 8 (5) (2009) 49–84.
[5] D. Messerschmitt, C. Szyperski, Software Ecosystem: Understanding an Indispensable Technology and Industry, MIT Press, 2003.
[6] K. Lauenroth, S. Toehning, K. Pohl, Model checking of domain artifacts in product line engineering, in: Proceedings of the International Conference on Automated Software Engineering (ASE), IEEE, 2009, pp. 269–280.
[7] S. Apel, W. Scholz, C. Lengauer, C. Kästner, Detecting dependences and interactions in feature-oriented design, in: Proceedings of the International Symposium on Software Reliability Engineering (ISSRE), IEEE, 2010, pp. 161–170.
[8] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, J.-F. Raskin, Model checking lots of systems: efficient verification of temporal properties in software product lines, in: Proceedings of the International Conference on Software Engineering (ICSE), ACM, 2010, pp. 335–344.
[9] S. Apel, H. Speidel, P. Wendler, A. von Rhein, D. Beyer, Detection of feature interactions using feature-aware verification, in: Proceedings of the International Conference on Automated Software Engineering (ASE), IEEE, 2011, pp. 372–375.
[10] T. Thüm, I. Schaefer, M. Kuhlemann, S. Apel, Proof composition for deductive verification of software product lines, in: Proceedings of the International Conference on Software Testing, Verification and Validation (ICST) Workshops, IEEE, 2011, pp. 270–277.
[11] W. Scholz, T. Thüm, S. Apel, C. Lengauer, Automatic detection of feature interactions using the java modeling language: an experience report, in: Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD), ACM, 2011, pp. 7:1–7:8.
[12] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, D. Beyer, Strategies for product-line verification: case studies and experiments, in: Proceedings of the International Conference on Software Engineering (ICSE), IEEE, 2013, pp. 482–491.
[13] T. Thüm, S. Apel, C. Kästner, M. Kuhlemann, I. Schaefer, G. Saake, Analysis Strategies for Software Product Lines, Tech. Rep. FIN-004-2012, School of Computer Science, University of Magdeburg, 2012.
[14] R. Hall, Fundamental nonmodularity in electronic mail, Automated Software Engineering 12 (1) (2005) 41–79.
[15] M. Svahnberg, J. van Gurp, J. Bosch, A taxonomy of variability realization techniques: research articles, Software – Practice and Experience 35 (8) (2005) 705–754.
[16] S. Apel, T. Leich, G. Saake, Aspectual feature modules, IEEE Transactions on Software Engineering 34 (2) (2008) 162–180.
[17] C. Prehofer, Feature-oriented programming: a fresh look at objects, in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), LNCS, vol. 1241, Springer, 1997, pp. 419–443.
[18] J. Liu, D. Batory, C. Lengauer, Feature-oriented refactoring of legacy applications, in: Proceedings of the International Conference on Software Engineering (ICSE), ACM, 2006, pp. 112–121.
[19] S. Apel, C. Lengauer, B. Möller, C. Kästner, An algebraic foundation for automatic feature-based program synthesis, Science of Computer Programming 75 (11) (2010) 1022–1047.
[20] S. Apel, C. Kästner, C. Lengauer, FeatureHouse: language-independent, automated software composition, in: Proceedings of the International Conference on Software Engineering (ICSE), IEEE, 2009, pp. 221–231.
[21] S. Apel, W. Scholz, C. Lengauer, C. Kästner, Language-independent reference checking in software product lines, in: Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD), ACM, 2010, pp. 65–71.
[22] M. Plath, M. Ryan, Feature integration using a feature construct, Science of Computer Programming 41 (1) (2001) 53–84.
[23] J. Kramer, J. Magee, M. Sloman, A. Lister, CONIC: an integrated approach to distributed computer control systems, computers and digital techniques, IEE Proceedings E 130 (1) (1983) 1–10.
[24] T. Thüm, I. Schaefer, M. Kuhlemann, S. Apel, G. Saake, Applying design by contract to feature-oriented programming, in: Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE), LNCS, vol. 7212, Springer, 2012, pp. 255–269.
[25] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, Symbolic model checking of software product lines, in: Proceedings of the International Conference on Software Engineering (ICSE), ACM, 2011, pp. 321–330.
[26] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold, An overview of AspectJ, in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), LNCS, vol. 2072, Springer, 2001, pp. 327–353.
[27] P. Chalin, J. Kiniry, G. Leavens, E. Poll, Beyond assertions: advanced specification and verification with JML and ESC/Java2, in: Proceedings of the International Conference on Formal Methods for Components and Objects (FMCO), LNCS, vol. 411, Springer, 2006, pp. 342–363.
[28] S. Apel, C. Kästner, A. Größlinger, C. Lengauer, Type safety for feature-oriented product lines, Automated Software Engineering 17 (3) (2010) 251–300.
[29] I. Cabral, M. Cohen, G. Rothermel, Improving the testing and testability of software product lines, in: Proceedings of the International Software Product Line Conference (SPLC), LNCS, vol. 6287, Springer, 2010, pp. 241–255.
[30] N. Siegmund, S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, G. Saake, Predicting performance via automated feature-interaction detection, in: Proceedings of the International Conference on Software Engineering (ICSE), IEEE, 2012, pp. 167–177.
[31] D. Bruns, V. Klebanov, I. Schaefer, Verification of software product lines: reducing the effort with delta-oriented slicing and proof reuse,

in: Proceedings of the International Conference on Formal Verification of Object-Oriented Software (FoVeOOS), LNCS, vol. 6528, Springer, pp. 61–75.
[32] T. Thüm, I. Schaefer, S. Apel, M. Hentschel, Family-based theorem proving for deductive verification of software product lines, in: Proc. of GPCE, ACM, 2012, pp. 11–20.
[33] M. Poppleton, Towards feature-oriented specification and development with event-B, in: Proceedings of the International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ), LNCS, vol. 4542, Springer, 2007, pp. 367–381.
[34] H. Li, S. Krishnamurthi, K. Fisler, Verifying cross-cutting features as open systems, in: Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), ACM, 2002, pp. 89–98.
[35] H. Li, S. Krishnamurthi, K. Fisler, Modular verification of open features using three-valued model checking, Automated Software Engineering 12 (3) (2005) 349–382.
[36] J. Liu, S. Basu, R. Lutz, Compositional model checking of software product lines using variation point obligations, Automated Software Engineering 18 (1) (2011) 39–76.
[37] L. Blair, J. Pang, Aspect-oriented solutions to feature interaction concerns using AspectJ, in: Feature Interactions in Telecommunications and Software Systems VII, IOS Press, 2003, pp. 87–104.
[38] C. Kästner, S. Apel, S. ur Rahman, M. Rosenmüller, D. Batory, G. Saake, On the impact of the optional feature problem: analysis and case studies, in: Proceedings of the International Software Product Line Conference (SPLC), Software Engineering Institute, 2009, pp. 181–190.
[39] T. Bowen, F. Dworack, C. Chow, N. Griffeth, G. Herman, Y.-J. Lin, The feature interaction problem in telecommunications systems, in: Proceedings of the International Conference on Software Engineering for Telecommunication Switching Systems (SETSS), IEEE, 1989, pp. 59–62.
[40] M. Jackson, P. Zave, Distributed feature composition: a virtual architecture for telecommunications services, IEEE Transactions on Software Engineering 24 (10) (1998) 831–847.

**Sven Apel** is the leader of the Software Product-Line Group funded by the esteemed Emmy Noether Programme of the German Research Foundation (DFG). The group resides at the University of Passau, Germany. He received his Ph.D. in Computer Science in 2007 from the University of Magdeburg, Germany. His research interests include novel programming paradigms, software engineering and product lines, and formal and empirical methods. He is the author or coauthor of over a hundred peer-reviewed scientific publications. He is a member of the IFIP Working Group 2.11 (Program Generation), and serves regularly in program committees of highly ranked international conferences. His work received awards by the Ernst Denert Foundation and the Karin Witte Foundation.

**Alexander von Rhein** is a Ph.D. student in the Software Product-Line Group, funded by the esteemed Emmy Noether Programme of the German Research Foundation (DFG). The group resides at the University of Passau, Germany. He received his master degree in Computer Science in 2010 from the University of Passau, Germany. His research interests include software engineering and product lines, automated software verification, and formal and empirical methods. In 2011, he received a grant on product-line verification in Google's Summer of Code.

**Thomas Thüm** is a Ph.D. student at the University of Magdeburg, Germany. He published his first top-tier conference paper as an undergraduate student in 2009. In 2011, he received the Software Engineering Award of the Ernst Denert Foundation for the best Master's thesis. His main research interests are software product lines, formal specification, deductive verification, and variability modeling. Since 2010, he is the project leader of FeatureIDE, a tool for feature-oriented software development, which is used in research and teaching world-wide.

**Christian Kästner** is an assistant professor in the School of Computer Science at Carnegie Mellon University. He received his Ph.D. in 2010 from the University of Magdeburg, Germany, for his work on virtual separation of concerns. For his dissertation he received the prestigious GI Dissertation Award. His research interests include correctness and understanding of systems with variability, including work on implementation mechanisms, tools, variability-aware analysis, type systems, feature interactions, empirical evaluations, and refactoring.