Detection of Feature Interactions using Feature-Aware Verification

Sven Apel [1], Hendrik Speidel [1], Philipp Wendler [1], Alexander von Rhein [1], Dirk Beyer [1,2]
[1] University of Passau, Germany
[2] Simon Fraser University, B.C., Canada

***Abstract*—A software product line is a set of software products that are distinguished in terms of features (i.e., end-user–visible units of behavior). Feature interactions —situations in which the combination of features leads to emergent and possibly critical behavior— are a major source of failures in software product lines. We explore how *feature-aware verification* can improve the automatic detection of feature interactions in software product lines. Feature-aware verification uses product-line–verification techniques and supports the specification of feature properties along with the features in separate and composable units. It integrates the technique of *variability encoding* to verify a product line without generating and checking a possibly exponential number of feature combinations. We developed the tool suite SPLverifier for feature-aware verification, which is based on standard model-checking technology. We applied it to an e-mail system that incorporates domain knowledge of AT&T. We found that feature interactions can be detected automatically based on specifications that have only local knowledge.**

I. Introduction

A *software product line* is a family of software products that share a common set of features and differ in others. A *feature* is an end-user–visible behavior of a software product that is of interest for some stakeholder. A *feature interaction* is a situation in which the composition of multiple features leads to emergent behavior that does not occur when one of them is absent. The feature-interaction problem (i.e., the problem of predicting and detecting feature interactions) has been studied and addressed before and is still a major challenge [8].

Our aim is to explore how product-line–verification techniques [9], [12], [14] (i.e., efficiently verifying that all products of a product line satisfy their specification) can be used to automatically detect feature interactions. Especially, we concentrate on two challenges that arise in feature-oriented software product lines: A first challenge, which was formulated by Hall [10], is to detect feature interactions based on specifications that do not have global system knowledge. The background is that the specification of a feature should not need to be aware of all other features of the system. It is desirable to specify *and* implement features in separate and composable units, while still being able to detect feature interactions [5], [10].

A second challenge, which applies to product-line analysis in general [2], [9], [11], [12], [14], is to detect feature interactions without the need of generating and checking all individual products. Typically, many different feature combinations are possible, so detecting feature interactions by generating all possible combinations may not be feasible.

We call our approach of verifying the absence (or detecting the presence) of feature interactions *feature-aware verification*. We base it on a number of ingredients. First, we provide a specification language to specify a feature's temporal properties in a separate and composable unit (along with its implementation). Second, we use the technique of *variability encoding* (which is based on configuration lifting [14]) to verify a complete product line in a single run ensuring that all possible feature combinations are free of critical feature interactions. Third, we use off-the-shelf model-checking techniques, rather than relying on modifications and extensions of existing model checkers.

We have developed the tool suite SPLverifier for feature-aware verification, and we use it in a case study —an e-mail client that was developed as a product line— to investigate the potential of feature-aware verification for detecting feature interactions. In this paper, we provide a brief overview of our approach, tool suite, and case study. Further details are available in an accompanying technical report [6] as well as on the Web.[1]

II. Background and Problem Statement

The goal of feature orientation is to make features explicit in design and code, for example, in the form of composable feature modules [1]. Feature modules are composed by superimposition [3]. Basically, superimposition merges the code of all features recursively based on nominal and structural similarity. Typically, there is a partial or total order of features defined, because feature composition is not generally commutative [4]. In our case study, we use the tool FeatureHouse [3] for composition.

In Figure 1, we depict excerpts of three feature modules taken from our case study. Feature *EmailClient* implements a basic e-mail client, feature *Encrypt* encrypts outgoing e-mails, and feature *Forward* forwards incoming e-mails to another host. Note that encryption is asymmetric and relies on the availability of proper keys—a circumstance that gives rise to an undesired feature interaction, as we will explain shortly.

EMailClient is the base feature in our example. It introduces a structure email for representing e-mails and the two functions outgoing and incoming for handling incoming and outgoing e-mails. Composing it with feature *Encrypt*, the existing structure email is extended by the two new fields isEncrypted

[1] http://www.fosd.de/FAV/

Feature *EMailClient*

```
// representation of e−mail
struct email {
  int id; char *from; char *to; char *subject; char *body;
};

// outgoing e−mails are processed by this function before they leave the system
void outgoing (struct client *client, struct email *msg) { ... }

// incoming e−mails reach the client at this point and are stored in a mailbox
void incoming (struct client *client, struct email *msg) { ... }
```

Feature *Encrypt*

```
// extending the e−mail structure by information on encryption
struct email {
  int isEncrypted;
  char *encryptionKey;
};

// encrypt a given e−mail, if the public key of the receiver is known
void encrypt (struct client *client, struct email *msg) { ... }

// override function outgoing to encrypt e−mails before they are sent
void outgoing (struct client *client, struct email *msg) {
  encrypt (client, msg);
  original (client, msg); // invoke the overridden function
}
```

Feature *Forward*

```
// forward an e−mail to another host
void forward (struct client *client, struct email *msg) { ... }

// override function incoming to forward e−mails automatically
void incoming (struct client *client, struct email *msg) {
  forward (client, msg);
  original (client, msg); // invoke the overridden function
}
```

Fig. 1. A feature-oriented implementation of an e-mail client in C (excerpt).

and encryptionKey, function encrypt is added, and the existing function outgoing is overridden to intercept outgoing e-mails and to encrypt them using function encrypt; keyword original invokes the overridden function. Feature *Forward* introduces a function forward and overrides the existing function incoming to forward incoming e-mails to another host.

As there are different feature-oriented languages and tools available [4], we concentrate on a common set of functionality: A feature module may add new fields, functions, and structures as well as refine existing functions by overriding.

Typically, features can be composed in different combinations. The compositional flexibility gives rise to feature interactions. A feature interaction is a situation in which new behavior emerges from the composition of two or more features that cannot easily be deduced from the behavior of the features involved. The emergent behavior can be undesired and associated with unexpected program states [8].

In our example, the features *Encrypt* and *Forward* have been developed independently of each other, only based on feature *EMailClient*. The composition of all three features leads to an undesired feature interaction that 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 (*Forward* does not know whether an e-mail is encrypted). This situation violates the specification of feature *Encrypt*, which states that e-mails that have been encrypted initially must never be sent unencrypted over the network.

```
 1 automaton EncryptSpec {
 2   introduction {
 3     shadow struct email { int in_encrypted; };
 4   }
 5
 6   before void incoming(_:struct client*, msg:struct email*) {
 7     msg−>in_encrypted = isEncrypted(msg);
 8   }
 9
10   after void outgoing(_:struct client*, msg:struct email*) {
11     if(msg−>in_encrypted == 1 && !isEncrypted(msg)) { fail; }
12   }
13 }
```

Fig. 2. Automaton-based specification of feature *Encrypt*.

Hall notes that the detection of feature interactions based on *feature-local* specifications is an open problem [10]. That is, the specification of a feature should not necessarily be aware of all other features of the system, but only of the ones it uses and extends directly. In our example, we need a specification of the desired behavior of feature *Encrypt* that states that e-mails that are received in encrypted form must not be sent in plain text—without referring to other independently developed features such as *Forward*.

Note that documenting dependencies and interactions among features by means of a feature model cannot solve the feature-interaction problem, as in real software projects the feature model is often not consistent with the specifications and implementations of the corresponding features, due to hidden implementation dependencies, evolution, and bugs [15].

III. Specifying Features

To be able to reason about feature interactions, each feature needs a formal specification of its behavior and the constraints that have to be fulfilled if it is selected (i.e., if it is present in the desired product). A key goal of feature-oriented programming is to implement and specify features in separate and composable units. Ideally, a feature's specification refers only to itself and a certain basis (i.e., the features that it extends and uses directly). We would like to explore to what extent this is possible. Beside missing global domain knowledge, scalability is a motivation for feature-local specifications. A system in which every feature has to be aware of every other feature does not scale well with regard to program comprehension when the number of features increases.

We have developed a language to specify features in separate and composable units, which we illustrate —due to the lack of space— by an example. In Figure 2, we show the specification of feature *Encrypt* (declared by keyword automaton). When the client receives an encrypted e-mail (lines 6–8), the status (encrypted or not) of the e-mail is stored (line 7) into a field that has been attached as a shadow to structure email (line 3). When an e-mail that was encrypted leaves the system (lines 10–12), it must still be encrypted; if not, the e-mail client reaches an error state defined by keyword fail (line 11).

In our specification language, an automaton specifies a *safety property* over the behavior of a feature. That is, it defines in which circumstances related to the feature the execution of the overall system reaches an error state; all other behaviors are accepted and thus considered safe. Technically, the source code of the features is instrumented with labels that indicate error states such that the model checker can perform a reachability analysis based on the corresponding control-flow graph.

IV. Detecting Interactions

Based on feature-local specifications, we explore two options of detecting feature interactions in a product line: generate all products and check them one at a time (Sect. IV-A), and generate one product that contains all features and check it in a single pass using variability information (Sect. IV-B).

A. Detecting Interactions in Products

Each feature comes with an implementation and a specification. Once the features of a product are selected, the composer (e.g., FeatureHouse) generates the corresponding code, which is checked against the specifications of the features selected for the product. To this end, we use a model checker that *statically* determines whether the execution of the composed product can reach an error state, as defined by the specifications of the features involved. If that happens, we know that the composition violates the constraints of at least one participating feature and indicates a feature interaction (or a defect in a single feature).

To verify that all products of a product line are free of interactions, we have to generate and check all products individually, which we call the *brute-force approach*.

B. Detecting Interactions in Product Lines

An alternative to the brute-force approach is to verify a *product simulator* that contains all features of a product line. In the product simulator, each feature can be enabled or disabled, according to the feature model. The state space of the product simulator subsumes the state spaces of all valid products of the product line such that the model-checking procedure can benefit from it during the checking process. It allows the model checker to detect feature interactions more efficiently, because not all individual feature combinations have to be unfolded in the model checker's state space.

The product simulator is obtained by *variability encoding*. The procedure of variability encoding is a modification of the regular composition process in order to create the product simulator incorporating *all* features of the product line. First, variability encoding defines for each feature a global boolean feature variable that models the presence or absence of the feature. Second, variability encoding introduces for each function refinement a dispatcher function that dispatches between the refined and the refining function depending on whether the feature that contains the refinement is selected. Third, variability encoding stores the dependency constraints between features (i.e., the feature model) using a boolean predicate over the boolean feature variables. Finally, the program execution is encapsulated in a conditional block that is executed only if the constraints imposed by the feature model are satisfied; this way, execution paths that are associated with invalid feature combinations are not considered by the model checker.

After variability encoding, we check the resulting product simulator against the specification of all features of the product line. We initialize the boolean variables of the features using a nondeterministic choice so that the model checker must assume that all feature combinations defined by the feature model may occur. This way, the model checker checks all possible feature combinations, that is, all combinations of feature code, without generating all individual products.

V. Case Study

To explore the feasibility of feature-aware verification for feature-interaction detection, we have developed the tool suite SPLverifier and applied it to a case study. SPLverifier and the case study are available on the project's Web site.

A. Implementation

SPLverifier is based on several existing tools and on tools that we developed for the purpose of feature-aware verification. For composition and variability encoding, we use FeatureHouse[2] (i.e., we compose features and inject guards by means of superimposition). For model checking, we use the tool CPAchecker [7], which allows to check temporal safety properties of C code, by means of either explicit or symbolic model checking. We translate the automata-based specifications to ACC aspects[3], and we use the ACC compiler to inject assertions in source code locations that have been specified by the corresponding automaton and that are relevant to a safety property.

B. The E-Mail Client Case Study

Hall's e-mail system consists of 10 features that give rise to 27 feature interactions [10]. It is divided into a client and a server. For our case study, we concentrate on the client because, for now, we do not focus on interactions in distributed scenarios. We implemented the features of the e-mail client in C with FeatureHouse following the specification of Hall (including a base program and two helper features). Furthermore, we included an entry function to trigger events in the client. Based on the work of Hall, we developed for every relevant feature a specification in the form of one or several automata. As discussed previously, a key requirement was to specify the features' behavior and safety properties based on local knowledge.

C. Experiments

We conducted a number of experiments with the e-mail product line. First, we generated all of its 40 products and checked them using CPAchecker. It turned out that with feature-aware verification, we were able to detect all feature interactions of the e-mail client based on the feature-local specifications of the input features. If the model checker does not report a counterexample (i.e., none of the safety properties

[2]http://www.fosd.de/fh/

[3]Aspect-oriented extension of C: http://research.msrg.utoronto.ca/ACC/

has been violated), we can be certain that the composition does not contain a feature interaction that violates the specification of the features involved.

Interestingly, we even found an interaction in *our* implementation that has not been documented by Hall. It occurs when both features *Decrypt* and *Forward* are selected: if a host forwards an e-mail automatically to another host that cannot decrypt this e-mail. This finding encourages us that our approach is useful to detect unknown interactions. We also checked the entire e-mail client product line using variability encoding. Again, we were able to detect all feature interactions, but without generating all possible feature combinations.

Finally, we compared the performance of the brute-force approach and the variability-encoding approach. We found that variability encoding is superior in cases in which the number of products that contain interactions is low (or zero). More details on the measurements are available in the accompanying technical report [6].

VI. Related Work

In the literature, there are two approaches of product-line verification that can be used for the detection of feature interactions: (1) check features as far as possible in isolation and (2) check the entire product line in a single pass.

The first approach has been proposed first by Li et al. [13]. They verify features modularly based on formal transition systems and CTL. Verifying a feature, it is determined which parts of a specification the feature satisfies and which parts have to be satisfied by other features. This information constitutes, in fact, a semantic interface of the feature, which is used during the verification of its composition with other features. Li et al. have a slightly different verification scenario in mind: they check to what extent a feature satisfies a specification that a product has to fulfill. In our approach, each feature comes with its own specification that states which properties have to hold when it is selected.

The second approach (i.e., check an entire product line in a single pass) has been proposed first by Lauenroth et al. [12] and Classen et al. [9]. They developed model checking approaches that take product-line variability into account. Basically, they enrich the state graph of a product line with variability information, much like in the variability-encoding approach. Based on their work, we explored whether and how product-line verification techniques can be used for feature-interaction detection. We emphasize the implementation and specification of features in separate and composable units, which was not the focus of Lauenroth et al. and Classen et al. Finally, we pursue an approach that is based entirely on off-the-shelf model checking, rather than on special developments and extension of existing model checkers.

Post and Sinz propose the notion of configuration lifting to efficiently verify variable C code [14]. The key idea is to replace each conditional preprocessor directive (e.g., #ifdef) by a corresponding if statement thus making it accessible to a software verification tool. Variability encoding is inspired by their approach.

VII. Conclusion

Feature-aware verification is an approach to detect feature interactions in product lines. First, we implement *and* specify features in separate and composable units, and detect interactions based on feature-local specifications. Locality of feature specifications is important for scalability and distributed feature composition. Second, we use the technique of variability encoding to avoid redundant verification effort due to product similarities. We used, extended, and developed a tool chain that supports feature-aware verification based on off-the-shelf model checking technology. We were able to automatically detect critical feature interactions (including a previously undocumented interaction) in our e-mail system.

Acknowledgment

We are grateful to J. Atlee, A. Classen, and M. Rosenthal for their comments to earlier drafts of this paper. We thank S. Boxleitner for his implementation of the e-mail client. This research was supported by the German DFG grants AP 206/2 and AP 206/4, and by the Canadian NSERC grant RGPIN 341819-07.

References

[1] S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *J. Object Technology*, 8(5):49–84, 2009.
[2] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type Safety for Feature-Oriented Product Lines. *Automated Software Engineering*, 17(3):251–300, 2010.
[3] S. Apel, C. Kästner, and C. Lengauer. FeatureHouse: Language-Independent, Automated Software Composition. In *Proc. ICSE*, pages 221–231. IEEE, 2009.
[4] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An Algebraic Foundation for Automatic Feature-Based Program Synthesis. *Science of Computer Programming*, 75(11):1022–1047, 2010.
[5] S. Apel, W. Scholz, C. Lengauer, and C. Kästner. Detecting Dependences and Interactions in Feature-Oriented Design. In *Proc. ISSRE*, pages 161–170. IEEE, 2010.
[6] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Feature-Aware Verification. Technical Report MIP-1105, University of Passau, September 2011.
[7] D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *Proc. CAV*, LNCS 6806, pages 184–190. Springer, 2011.
[8] M. Calder, M. Kolberg, E. Magill, and S. Reiff-Marganiec. Feature Interaction: A Critical Review and Considered Forecast. *Computer Networks*, 41(1):115–141, 2003.
[9] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In *Proc. ICSE*, pages 335–344. ACM, 2010.
[10] R. Hall. Fundamental Nonmodularity in Electronic Mail. *Automated Software Engineering*, 12(1):41–79, 2005.
[11] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type Checking Annotation-Based Product Lines. *ACM TOSEM*, 2011. To appear.
[12] K. Lauenroth, S. Toehning, and K. Pohl. Model Checking of Domain Artifacts in Product Line Engineering. In *Proc. ASE*, pages 269–280. IEEE, 2009.
[13] H. Li, S. Krishnamurthi, and K. Fisler. Verifying Cross-Cutting Features as Open Systems. In *Proc. FSE*, pages 89–98. ACM, 2002.
[14] H. Post and C. Sinz. Configuration Lifting: Verification meets Software Configuration. In *Proc. ASE*, pages 347–350. IEEE, 2008.
[15] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *Proc. GPCE*, pages 95–104. ACM, 2007.