Automatic Detection of Feature Interactions using the Java Modeling Language: An Experience Report

Wolfgang Scholz
University of Passau, Germany

Thomas Thüm
University of Magdeburg, Germany

Sven Apel, Christian Lengauer
University of Passau, Germany

ABSTRACT

In the development of complex software systems, interactions between different program features increase the design complexity. Feature-oriented software development focuses on the representation and compositions of features. The implementation of features often cuts across object-oriented module boundaries and hence comprises interactions. The manual detection and treatment of feature interactions requires a deep knowledge of the implementation details of the features involved. Our goal is to detect interactions automatically using specifications by means of design by contract and automated theorem proving. We provide a software tool that operates on programs in Java and the Java Modeling Language (JML). We discuss which kinds of feature interactions can be detected automatically with our tool and how to detect other kinds of interactions.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*class invariants, correctness proofs, formal methods, programming by contract, reliability*

General Terms

Design, Languages, Verification

Keywords

Feature Interaction, Software Product Lines, JML, FeatureHouse

1. INTRODUCTION

Feature-oriented software development (FOSD) is a programming paradigm that has gained momentum in recent years. In FOSD, a program is viewed as a composition of features [1]. A *feature* is a unit of composition which can cut across module boundaries and which is relevant for some stakeholder. Different feature combinations result in different programs. The set of programs that can be generated from a given set of features is called a *software product line* [20].

A major challenge of FOSD is that of feature interactions [8], which may arise when combining features to generate programs. We use the neutral term *feature interaction* for a behavior of a feature which is only expressed in combination with other features, and the biased term *feature conflict* for an unwanted feature interaction. We focus on the detection of *semantic* feature conflicts. A feature conflict is semantic if it cannot be detected at compile time without supplying an additional specification. *Syntactic* interactions can be detected using type systems and are considered elsewhere [14, 2]. The detection of feature conflicts can be a daunting task because of the crosscutting nature of features and the multiplicity of possible feature combinations. Our aim is to base it on formal methods and provide tool support.

In previous work, we analyzed how to detect feature interactions during the design phase [4]. Here, our aim is to detect feature interactions during the implementation phase that may not be detectable during the design phase or that are specific to a certain implementation. We base our approach on formal specification using design by contract. *Design by contract* is a common technique of specifying the behavior of methods and classes formally by means of method contracts and class invariants to increase the reliability of software [19]. Thus, design by contract can be used to specify programs. We take advantage of a prominent device for design by contract, namely the Java Modeling Language (JML) [17], because of the available verification tools.

We specify features formally in JML and propose how to generate the JML specification of a program along with its implementation. Then, we use the extended static checker *ESC/Java2* [11] to generate proof obligations that we solve with the automatic theorem prover *Simplify* [12]. The overall goal is to evaluate the practicability of JML and automated theorem proving for the detection of feature interactions. We make the following contributions:

1. We explore how to combine design by contract with feature-oriented software development for specification at the granularity level of classes and methods inside features.

2. We provide tool support for the composition of feature-oriented specifications given in the JML and the verification of the accompanying Java code against the union of the specifications of every individual product.

SPLC '11, August 21-26, 2011, Munich, Germany

```
class C {
  //@ invariant a >= 0;
  int a;
  /*@ requires (a > 0);
    @ ensures (a > 1); */
  void f() {
    a += a;
  }
}
```

Figure 1: Sample Java code, annotated with contracts written in JML

3. We use a small case study to shed light on which problem sizes can be proved automatically and discuss future challenges.

2. INTERACTION DETECTION USING FORMAL SPECIFICATIONS

Let us introduce the necessary background of our work. We use formal specifications to be able to reason about which feature interactions are considered flaws. We use design by contract as a means to provide the specifications needed to detect feature conflicts. As the specifications are given individually for each feature, the specification of a product is composed of several specification fragments associated with the individual features. Important to us is to find a balance between flexibility and restrictiveness in the specification of software product lines.

2.1 Formal Product Line Verification

The validity of a product line P can be checked against a given specification S. The notation for a product line complying with a specification is:

$$P \models S$$

In the setting of this work, a feature F comes with an implementation I_F and a specification S_F.

$$F = (I_F, S_F)$$

Using the n-ary composition operator comp of the tool suite FeatureHouse [3], we can derive the implementation $I_{F_1,\ldots,F_n}$ of a product by superimposition of the implementation fragments of its features $F_1, \ldots, F_n$.

$$I_{F_1,\ldots,F_n} = \text{comp}(I_{F_1}, \ldots, I_{F_n})$$

To obtain the specification with which an individual product must comply, we can also use superimposition.

$$S_{F_1,\ldots,F_n} = \text{comp}(S_{F_1}, \ldots, S_{F_n})$$

A product is valid if the composed implementation of its features complies with its combined specification.

$$\text{comp}(I_{F_1}, \ldots, I_{F_n}) \models \text{comp}(S_{F_1}, \ldots, S_{F_n})$$

The entire product line is valid if every derivable product is valid, i.e., if the above formula is valid for every permitted combination of features.

Here, we focus on product lines for which specifications are given for individual features [4], in contrast to scenarios, in which there is one specification for the entire product line.

2.2 Design by Contract

We explore design by contract [19] to specify each feature separately. Certain implementation elements are supplied with a specification: methods with pre- and postconditions, classes with invariants.

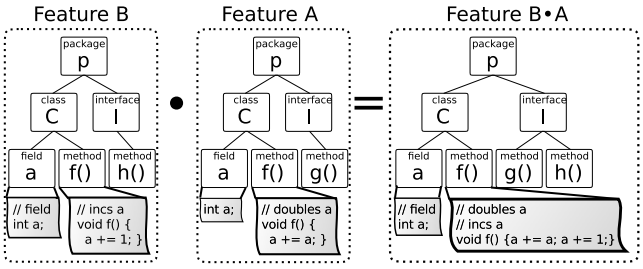

Figure 2: Sample composition of the FSTs of two features

Consider, for example, the Java program in Figure 1. Class `C` has an integer field `a` and a method `f`. A class invariant states that `a` ≥ 0 at all times. `f` has a precondition `a` > 0 and a postcondition `a` ≥ 1.

Design by contract is a software development process providing formal specifications at an early stage of the development. One benefit of formal specification is that it can be used by automated tools in later development stages.

2.3 Composition of Features

Features can be represented by feature structure trees (FSTs). A feature structure tree is a partial parse tree with names as the inner nodes and unparsed text as the leaf nodes. Features can be composed by superimposition of their FSTs. Superimposition is an algorithm to merge several FSTs hierarchically into one, which includes the superset of all names as a result. This can be done automatically by software tools like FeatureHouse [3].

Identically named FST elements of different features are merged to a single element in the composed program [3]. Thus, the composition of features relies on classes and members with different concerns being named differently.

For example, in Figure 2, feature A has a method `g` in a Java interface `I`. Feature B has a method `h` in an interface `I`. After composition of A and B, there will be both methods in interface `I`. However, both features introduce a method `f` in class `C`. After composition there will be only one method `f`, the code of which has to be obtained from both implementations. As there can be only one common implementation of the resulting method, the assumption is that the problems both originating instances of `f` are meant to solve conform to each other.

In feature-oriented software development, accidental name clashes pose a problem, since features are naturally subject to refinement and features may be implemented in isolation. For example, there can be unexpected method overrides. Unlike in module systems, where there are ways to avoid such name clashes [9], it is difficult to tackle this problem in feature-oriented systems.

Defining contracts for classes and methods can be a step into this direction, as it is likely that methods with different concerns also have incompatible contracts. Compared to Java comments as in Figure 2, contracts can be evaluated automatically by tools. The compatibility can be assessed by joining the contracts and checking their satisfiability, possibly by an automatic theorem prover.

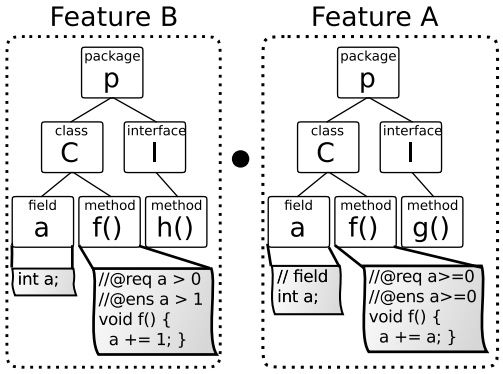

Figure 3: Example FSTs before parsing JML clauses

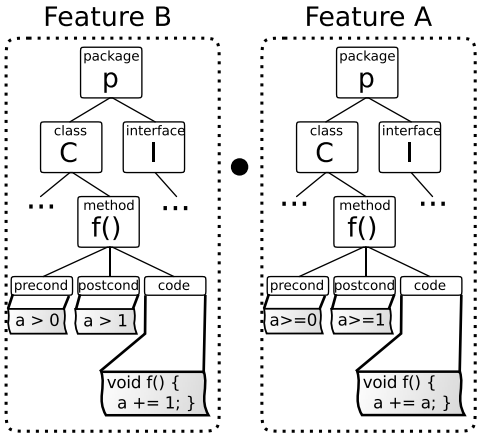

Figure 4: Processed FSTs after parsing JML clauses

2.4 Selectable Restrictiveness of Specification

A formal specification describes a solution space, which limits the set of correct implementations. Specifications can be more or less restrictive. An unrestrictive specification provides a wide solution space, leaving many possible solutions and implementations flexibility. If the solution space defined by the formal specification contains solutions that are regarded as incorrect, it is said that the solution space is underspecified. A restrictive specification narrows the solution space, leaving only limited flexibility for the implementation. In practice, any piece of software needs to be maintained and extended eventually. A restrictive specification unaware of eventual software extension is likely to forbid it. A specification that narrows down the solution space more than actually necessary is called overspecification. It is possible to review a specification at a later time of development, but with the validity of earlier defined properties at stake. Thus, the specification is to be designed with flexibility in mind but, on the other hand, must be strong enough to ensure that only correct implementations are valid.

The trade-off between flexibility and restrictiveness of specifications is especially important for product lines. If new functionality is required for an existing software product line, it is easier to introduce new features instead of refactoring old ones, as new optional features should not affect existing products. To be able to add new functionality without changing existing products, the existing specification should not be too restrictive. Otherwise, it may rule out potentially needful enhancements of the product line. On the other hand, underspecification poses the risk of unrecognized feature conflicts, as explained earlier.

3. TOOL PROTOTYPE

In this section, we present our prototype tool *SpeK*. First, we discuss existing tools that we extended to support the composition of specifications. Second, we give an overview of the parsing process and discuss the composition.

3.1 Parsing JML

FeatureHouse is a tool for feature composition which already supports a number of languages [3]. To use JML in a feature-oriented fashion, we extended FeatureHouse's existing Java support. We decided to extend FeatureHouse as it is open-source and can be extended easily. JML elements inside comment blocks are parsed using an additional layer of FST processing before and after superimposition. In this layer, JML clauses are extracted and rendered as child nodes of the method they specify. After superimposition, these child nodes are deconstructed and their content is woven into the method's FST node.

Consider again the example of Figure 2. Consider both features A and B optional. FeatureHouse generates, for both features, FSTs that have three levels: package, package members, and class members (see Figure 3).

The JML parsing layer in FeatureHouse inspects the comment block assigned to each node at the level of class members. Method nodes, which are given as leaf nodes, are transformed to inner nodes and their content is divided into the following subnodes (see Figure 4):

- One node contains the method's code.
- For each different type of JML clause detected, one child node is generated. Thus, the tool framework is able to assign an individual composition rule for each type.
- JavaDoc comments are distinguished from other comment types and will be concatenated on composition.
- Other comments are rearranged to form a block after composition.

In our example, method `f` has a JML precondition and a JML postcondition both for feature A and B.

The resulting code can be checked against the specification by run-time assertions or by theorem proving at compile-time. We focus on compile time checking using the extended static checker *ESC/Java2* [11] and *Simplify* [12] as theorem prover.

3.2 JML Composition Rules

With new terminal FST node types, new composition rules are needed in FeatureHouse. The standard way FeatureHouse handles two identically named FST method nodes is to choose the one which is given by the composition precedence. Identical method names either result from an accidental name clash or the two method implementations resemble the same problem. In the latter case, the composition rule for FST method nodes should merge the implementations into one. This should also be reflected by the composition rule for method specifications.

```
/*@ requires a >= 0;
  @ ensures a >= 0; */
void f() {
 a += a;
}
```

Feature A

```
/*@ requires a > 0;
  @ ensures a > 1; */
void f() {
 a += 1;
}
```

Feature B

```
/*@ requires (a >= 0)
  @   || (a > 0);
  @ ensures (a >= 0)
  @   && (a > 1); */
void f() {
 a += a;
 a += 1;
}
```

Product $B \bullet A$

```
/*@ requires (a > 0)
  @   || (a >= 0);
  @ ensures (a > 1)
  @   && (a >= 0); */
void f() {
 a += 1;
 a += a;
}
```

Product $A \bullet B$

Figure 5: Code of method `f` in different feature compositions. Here, the code was composed by concatenation of the originating methods' bodies. The composition order does not matter for specifications, but it does for code: product $A \bullet B$ is valid, product $B \bullet A$ is not.

For specification composition, we follow Liskov's substitution principle [18]:

> Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

Following this principle, if two methods are composed with a certain contract each, there may be method calls in the resulting program that rely on one or both of the two contracts. Thus, the contract of the composed method should require at least one of the two preconditions and should ensure both postconditions. We compose preconditions disjunctively and postconditions conjunctively (see Figure 5 and Figure 6). The result is a method contract that can be used seamlessly inside both features just as in the isolated feature.

More formally, let A and B be two features to be composed. Let `f` be a method that both features introduce with same name and type (for distinction, call them $\mathtt{f}_A$ and $\mathtt{f}_B$). Let P_A be the precondition and Q_A the postcondition of $\mathtt{f}_A$ and similarly P_B and Q_B the pre- and the postcondition of $\mathtt{f}_B$. The contracts of `f` for both features are:

$$\{P_A\}\ \mathtt{f}_A\ \{Q_A\}$$
$$\{P_B\}\ \mathtt{f}_B\ \{Q_B\}$$

Let $\mathtt{f}_{B \bullet A}$ be the composition of $\mathtt{f}_A$ and $\mathtt{f}_B$. Then, the contract of $\mathtt{f}_{B \bullet A}$ is:

$$\{P_A \vee P_B\}\ \mathtt{f}_{B \bullet A}\ \{Q_A \wedge Q_B\}$$

THEOREM 1. *$f_{B \bullet A}$ is a subtype of both f_A and f_B in the sense that any caller of f expecting the contract of f_A (and likewise f_B) may also call $f_{B \bullet A}$.*

For brevity, we give only an informal proof. It is easy to conclude that any caller supplying P_A (and likewise P_B) also supplies $P_A \vee P_B$ and that $Q_A \wedge Q_B$ satisfies any caller expecting Q_A (and likewise Q_B).

This composition rule makes method contracts more restrictive, but does not invalidate any caller code. If the composed contract is overspecified, we presume that it is the result of an accidental name clash.

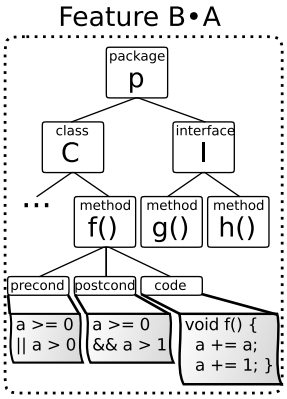

Figure 6: FST resulting from the composition $B \bullet A$

There is no similarly seamless way of composing the method bodies. As is standard in FEATUREHOUSE, the order of composition determines which feature's method body takes precedence. The other one will be overridden. There are different ways of composition. For instance, in our running example, both method bodies are concatenated (see Figure 5).

`invariant` clauses are treated separately, as they belong to the FST level of class members. We decided to concatenate invariant clauses on composition. The resulting product has the superset of invariants of all of its features. Thus, the set of invariants becomes monotonically more restrictive with additional features being composed.

`assignable` clauses may be used to restrict the variables that the method can update and are simply appended when composed. Thus, the set of assignable variables becomes monotonically less restrictive with each feature added to a feature selection.

The integration of composition rules for additional clauses can be done in a matter of minutes. They will be added in later case studies.

3.3 Transparency of the JML Parsing Process

After composition, the FST nodes generated by the JML layer are folded back into terminal method nodes (see Figure 7). Thus, the process of parsing and composing JML annotations is transparent to FEATUREHOUSE.

The combined implementation of a method may no longer comply with the combined contract. As the result of the combination is regular JML-annotated Java, a violation of the contract can be detected automatically by *ESC/Java2* or a similar prover framework which is able to handle JML.

4. CASE STUDY

We present a small case study as proof of concept. The product line *ListPL* has five features: *ListBase*, *Cons*, *Snoc*, *Stack* and *Sorted*. Feature *ListBase* is mandatory for all products. All other features are optional (see Figure 8). As these four features can be composed in any combination, the product line contains sixteen different products. Feature conflicts may occur in any combination.

Our tool *SpeK* and the case study are publicly available on the Web.[1]

[1] `http://www.fosd.de/spek`

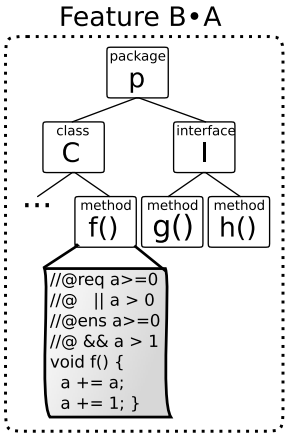

Figure 7: JML handling is transparent to FeatureHouse. JML is parsed after primary FST construction and adds additional granularity to the FST. After superimposition, the additional granularity is deconstructed.

4.1 Description of the Features

A short description of every feature is given below. FeatureHouse expects a total composition order of features. The position in the list reflects this ordering.

ListBase. Class `IntList` consists of an array of integer values. Method `push` adds a new element to the end of the array. The contract of `push` ensures that no element is removed and that the new element is located somewhere in the array.

Cons. A new clause is added to the specification of `push`, stating that the newly added element must be at the last array position.

Snoc. A method `snoc`, which inserts an element to the first position of the array, is introduced to `IntList`.

Stack. Methods `top`, `pop` and `isEmpty` are added to `IntList`. While `top` and `isEmpty` have only read access to the array, `pop` removes the last array entry and reduces the array's size.

Sorted. An invariant, which states that the array is sorted, is introduced to `IntList`. Method `push` is refined, calling a newly introduced helper method `sort` before its return. As *Sorted* is the last feature to be composed, the sorting is always done as last statement in `push`, regardless of any previous refinements.

Composing any feature with feature *ListBase* yields a valid implementation, according to the composed specification of that product, and the run-time behavior is as expected. Furthermore, all sixteen products can be composed and compiled without type errors (i.e., there are no syntactic feature conflicts). Yet, in some products containing three or more features, run-time errors occur under special circumstances, as there are semantic feature conflicts. Due to the limited scope of this case study, it is possible to anticipate all existing feature conflicts and evaluate the performance of the feature interaction detection manually:

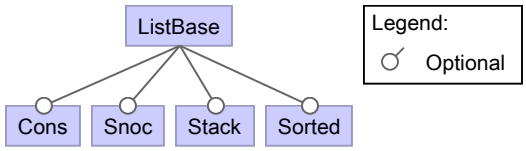

Figure 8: Feature model of the product line *ListPL*

1. Both specifications of features *Stack* and *Cons* imply that `i == l.push(i).top()` for all `i` and `l`, where `i` is of type `int` and `l` of type `IntList`. If feature *Sorted* is present, this property is violated, as the list is sorted after inserting `i`. The interaction should be detected in all products of *Sorted* and *Cons*. The following products contain this conflict:

   $Sorted \bullet Cons \bullet ListBase$,
   $Sorted \bullet Stack \bullet Cons \bullet ListBase$,
   $Sorted \bullet Snoc \bullet Cons \bullet ListBase$, and
   $Sorted \bullet Stack \bullet Snoc \bullet Cons \bullet ListBase$.

2. *Snoc* introduces method `snoc`, which modifies the array. As feature *Sorted* is unaware of method `snoc` in feature *Snoc*, no sorting is done after the modification. As a result the array is no longer sorted. This interaction should be detected in all products of *Sorted* and *Snoc*. The following products contain this conflict:

   $Sorted \bullet Snoc \bullet ListBase$,
   $Sorted \bullet Snoc \bullet Cons \bullet ListBase$,
   $Sorted \bullet Stack \bullet Snoc \bullet ListBase$, and
   $Sorted \bullet Stack \bullet Snoc \bullet Cons \bullet ListBase$.

3. Method `pop` in feature *Stack* also modifies the array. We expect interaction detection to recognize that the deletion of the top stack element does not violate the invariant of *Sorted*.

5. EXPERIMENTAL RESULTS

We used the prover framework *ESC/Java2* in combination with the prover *Simplify* to check every possible product of the product line *ListPL*. *ESC/Java2* generates verification conditions for every method and calls *Simplify* to verify them. *ESC/Java2* is a neither sound nor complete framework for extended static checking [11]. Hence, *ESC/Java2* may both miss and falsely signal errors. Although there are sound prover frameworks available, we believe that using *ESC/Java2* is appropriate to gather experience in the application of static prover frameworks for feature conflict detection. The fact that ESC/Java2 is a bounded checker makes it applicable to larger project sizes and avoids problems with non-termination. Furthermore, there is a proper plug-in for the development environment Eclipse available.

5.1 Recognized Conflicts

We used *ESC/Java2* with default program parameters. The interaction between *Sorted* and *Snoc* was detected in all products containing these two features. The prover generates a warning that method `snoc` violates the invariant of feature *Sorted* stating that the array is sorted:

```
IntList.java 53 Warning: Possible violation of object invariant
Associated declaration is "~/workspace/JMLIntListProducts/
voidSortedSnocListBase/IntList.java", line 15, col 6:
@ invariant (\forall int k; 0 <= k && k < data.length-1; ...
  ^
```

5.2 Unrecognized Conflicts

Using *ESC/Java2* with default parameters, the conflict between features *Sorted* and *Cons* is not detected. *Cons* provides an additional postcondition for method `push`, which makes the contract more restrictive. While the postcondition imposed by *ListBase* requires only that no old elements are removed and the new element is present after the call, *Cons* also requires that the new element is located at the last position of the array. The invariant of *Sorted* states that all array elements are sorted. Method `push` can only conform to both contracts if its parameter is the smallest element in the array, which is not guaranteed by its precondition. The insertion of a non-smallest element causes the sorting in method `push` to violate the postcondition of *Cons*. Although this is a violation of the contract for *Cons*, *ESC/Java2* fails to notice it.

The unrecognized conflict is due to the unsoundness of *ESC/Java2* when dealing with loops. Instead of guessing loop invariants, *ESC/Java2* only unrolls a loop a given number of times. Method `sort` is an implementation of the bubble sort, which consists of a doubly nested loop. If called with standard program parameters, *ESC/Java2* unrolls each loop only once, which seems to be not sufficient to recognize that the inner loop permutes array elements. *ESC/Java2* responses within two seconds that method `push` passed the analysis:

```
SortedConsListBase.IntList: push(int) ...
    [0.512 s 25567304 bytes]  passed
SortedConsListBase.IntList: sort() ...
    [0.0 s 25567304 bytes]  passed immediately
 [1.861 s 25567304 bytes total]
```

5.3 Adjustments to Detect All Interactions

In the command line version of the *ESC/Java2* tool, a program parameter can be specified to increase the number of times a loop is unrolled. In our case study, unrolling one additional loop step enables *ESC/Java2* to reason about the assignments inside the inner loop. The expenditure of time for *Simplify* to run the proof increases only slightly.[2]

Applying the additional step of loop unrolling to the whole program reveals that *Simplify* can no longer verify the postcondition of method `push` even in the simplest product *ListBase*, although the verification succeeded in the first place and produces the following warning:

```
IntList.java:33: Warning: Postcondition possibly not established
    }
    ^
Associated declaration is "IntList.java", line 15, col 2:
@ ensures( data[data.length-1] == newTop) ...
  ^
```

This is a false positive. It seems that the specification of method `push` in feature *ListBase*, stating that all array elements must be retained in the array, is too complex to be verified by *Simplify*. Using the JML statements `assert` and `assume`, we were able to provide an intermediate goal with which *Simplify* is able complete the proof:

```
//@ assert(\forall int k; 0<=k && k<data.length; tmp[k]==data[k]
    ==> (\exists int z; 0<=z && z<tmp.length; tmp[z]==data[k]));
//@ assume(\forall int k; 0<=k && k<data.length; tmp[k]==data[k]
    ==> (\exists int z; 0<=z && z<tmp.length; tmp[z]==data[k]));
```

[2]Average duration of the automated verification of product $Sorted \bullet Stack \bullet Snoc \bullet Cons \bullet ListBase$ over five runs: standard parameters $2.3s$, one additional loop step $3.1s$.

Finally, the prover framework is able to detect all interactions as expected (see Section 4.1). Our study demonstrated that it is possible to detect feature interactions automatically based on design-by-contract specifications. However, we found also cases in which manual intervention is necessary, which suggests to explore the trade-off between simplicity (checking only a subset of the state space) and soundness (checking the entire state space).

6. RELATED WORK

Type Checking. Type checking approaches can only handle type errors caused by feature interactions, which we call syntactic feature conflicts [13, 2, 5]. We aim at the detection of semantic feature conflicts by reasoning with given additional specification. None of the conflicts listed in Section 4.1 can be detected by type checking.

Model Checking. Several approaches use model checking for feature interaction detection [22, 6]. There is an overview of earlier approaches [24]. In contrast to these approaches, we also take code into consideration.

Recently, Classen et al. and Lauenroth et al. proposed to use model checking for verification of whole product lines [10, 16]. The model checker is given the feature model and the code of all features. The checker is able to check whether a given condition holds for all products of the product line. To our knowledge, there are no approaches that join model checking and design by contract.

Model-Driven Development. Poppleton extended Event-B with support for feature-orientation [21]. There is a subtle difference, as we focus on the ability to detect feature conflicts. The Event-B community is interested in the decomposition and composition of submodels for reasons of proof performance.

Our earlier approach on FeatureAlloy already posed the question of recognizability of semantic feature conflicts [4]. The current approach brings together specification and code and aims to provide the means for a safer merging of method code.

Proof Reuse. In the presented approach, every product to be analyzed for feature conflicts needs to be built individually. The number of products is exponential in the number of optional features. Although we do not yet see how to reduce this number to a polynomial scale without increasing unsoundness, Bruns et al. and Thüm et al. presented recent approaches that reduce the amount of effort spent in redundant proofs in software product lines [7, 23].

7. FUTURE WORK

The integration of design by contract into FOSD is still in development. We identify the following avenues of further work.

Full JML support. To support the full JML standard [17], the following topics need to be handled:

- additional method clauses
- model and ghost fields

- import definitions
- specification refinement chains
- heavyweight specification cases
- separate JML compilation units

We are aware that the subset of JML that we support presently is incomplete but we believe it to be useful. We plan to increase the support in future case studies.

Specification Composition Approaches. The JML keyword `also` is used in the refinement of inherited method specifications [17]. The similarity of the semantics of this keyword and our specification composition mechanism is subject of further research. In the work presented here, we investigated whether it is possible to detect semantic feature conflicts by annotating code with specifications. We experimented with a single composition mechanism for specifications in JML. In ongoing work, we are inspecting and analyzing several composition approaches, whereas some rely on Liskov's substitution principle and some do not.

Scalability. A comprehensive scalability analysis of the approach has not yet been conducted. We expect that the scalability of JML parsing and composition should be roughly the same as that of Java composition with FeatureHouse. The dominating amount of run time is spent on the proving process (*ESC/Java2* and *Simplify*). We expect that *ESC/Java2*'s behavior of concentrating on proving each method in isolation has a positive effect on scalability if the individual methods' verification conditions are kept small, for which the programmer is accountable.

Usability. Design by contract has been employed in industry where software reliability is of elevated concern. These industrial fields also have an increasing demand for software flexibility. We hope that with ongoing tool development and research, the hurdles of employing FOSD are going to be alleviated. Indeed, the problem of missing unsoundness warnings is being targeted [15].

Satisfiability. In our case study, we checked whether composed programs satisfy their composed specification. But, before actually trying to write corresponding code, it should be checked whether the specification is satisfiable at all, i.e., whether there exists at least one implementation fulfilling the specification. Satisfiability is especially important when specifications are composed from several features, where unsatisfiable specifications may occur more frequently. Checking satisfiability should be automated just like verification.

Comparison of Different Prover Frameworks. In our case study, we used *ESC/Java2* together with the prover *Simplify*. As proof performance varies greatly with different provers and prover frameworks, we expect a thorough analysis involving several tools to be a worthwhile refinement of our preliminary study. Further potential candidates for comparison are frameworks for static checking of JML-annotated Java, such as *JACK*[3], *KeY*[4], and *Krakatoa*[5].

[3] `http://www-sop.inria.fr/everest/soft/Jack/`
[4] `http://key-project.org/`
[5] `http://krakatoa.lri.fr/`

Comparison with Other Approaches. As tool development is ongoing, we hope that a meaningful comparison of the suitability of model checking and theorem proving for product line verification will be realizable soon.

8. CONCLUSION

To our knowledge this is the first approach of automatic composition of feature-oriented specifications in JML. While our software tool is still under development, we claim that the supported subset of JML is meaningful enough to reason that design by contract is suitable for the detection of conflicting feature interactions. In a small case study, we were able to detect feature conflicts semi-automatically, using *extended static checking*, which are not detectable without the specifications given in JML. Several interactions were found automatically, but a higher rate of detection was achieved with manual assistance. The detected conflicts resulted in violations of a class invariant and a method postcondition.

Practical limits are imposed by the capacity of current prover frameworks. As the prover framework *ESC/Java2* is both unsound and incomplete in principle, the ability to find feature conflicts depends heavily on the parameter settings of the prover framework and on the way certain specifications are formulated. The proof framework did not always report that certain code sections were not checked. Limits of design by contract for feature interaction detection were not discovered in this work.

In our case study, we were able to detect all feature conflicts of the types listed in Section 4.1 semi-automatically using design by contract. Whether this can be done for all possible feature conflicts is a question to be answered in the future. Future work should also evaluate other automatic and interactive theorem provers, especially those that are sound and complete, while our approach and tool to compose specifications in JML can be used.

9. ACKNOWLEDGMENTS

The work of Scholz, Apel, and Lengauer was funded by the DFG research grants AP 206/2 and AP 206/4. We are greateful to the anonymous reviewers for their helpful comments.

10. REFERENCES

[1] S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology (JOT)*, 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 (ASE)*, 17(3):251–300, 2010.

[3] S. Apel, C. Kästner, and C. Lengauer. FeatureHouse: Language-independent, automated software composition. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 221–231. IEEE Computer Society, 2009.

[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 Computer Society, 2010.

[5] S. Apel, W. Scholz, C. Lengauer, and C. Kästner. Language-independent reference checking in software product lines. In *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD)*, pages 65–71. ACM Press, 2010.

[6] L. Bousquet. Feature interaction detection using testing and model-checking—Experience report. In *World Congress on Formal Methods*, pages 622–641. Springer, 1999.

[7] D. Bruns, V. Klebanov, and I. Schaefer. Verification of software product lines: Reducing the effort with delta-oriented slicing and proof reuse. In *Proc. Int'l Conf. Formel Verification of Object-Oriented Software (FoVeOOS)*, volume 6528 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2010.

[8] M. Calder, M. Kohlberg, E. H. Magill, and A. Reiff-Marganiec. Feature interaction: A critical review and considered forecast. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 41(1):115–141, 2003.

[9] F. Calliss. A comparison of module constructs in programming languages. *SIGPLAN Not.*, 26:38–46, 1991.

[10] A. Classen, P. Heymans, P. Schobbens, A. Legay, and J. Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. In J. Kramer, J. Bishop, P. Devanbu, and S. Uchitel, editors, *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 335–344. ACM Press, 2010.

[11] D. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer-Verlag, 2005.

[12] D. Detlefs, G. Nelson, and J. Saxe. Simplify: A theorem prover for program checking. *Journal of the ACM*, 52:365–473, 2005.

[13] C. Kästner and S. Apel. Type-checking software product lines – A formal approach. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 258–267. IEEE Computer Society, 2008.

[14] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2011. To appear.

[15] J. Kiniry, A. Morkan, and B. Denby. Soundness and completeness warnings in ESC/Java2. In *Proceedings of Conference on Specification and Cerification of Component-based Systems (SAVCBS)*, pages 19–24. ACM Press, 2006.

[16] K. Lauenroth, K. Pohl, and S. Toehning. Model checking of domain artifacts in product line engineering. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 269–280. IEEE Computer Society, 2009.

[17] G. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. JML reference manual draft, 2008. Revision: 1.235.

[18] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16:1811–1841, 1994.

[19] B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.

[20] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[21] 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)*, pages 367–381. Springer-Verlag, 2007.

[22] B. Stepien and L. Logrippo. Feature interaction detection using backward reasoning with LOTOS. In *Protocol Specification, Testing and Verification XIV (PSTV 94)*, pages 71–86. Chapman & Hall, 1995.

[23] T. Thüm, I. Schaefer, M. Kuhlemann, and S. Apel. Proof composition for deductive verification of software product lines. In *Proceedings of the International Workshop on Variability-intensive Systems Testing, Validation & Verification (VAST)*, 2011. To appear.

[24] H. Velthuijsen. Issues of non-monotonicity in feature-interaction detection. In *Feature Interactions in Telecommunications Systems*, 3, pages 31–42. IOS Press, 1995.