Subclack: Feature-Oriented Programming with Behavioral Feature Interfaces

Thomas Thüm
University of Magdeburg
Germany

Sven Apel
University of Passau
Germany

Andreas Zelend
University of Augsburg
Germany

Reimar Schröter
University of Magdeburg
Germany

Bernhard Möller
University of Augsburg
Germany

ABSTRACT

Feature-oriented programming extends object-oriented programming to support feature modularity. Feature modules typically cut across class boundaries to implement end-user-visible features. Customized program variants can be composed automatically given a selection of desired feature modules. We propose behavioral feature interfaces based on design by contract for precise localization of faulty feature modules. There are three different approaches for feature-module composition, which are considered to be equivalent in the literature. We discuss advantages and disadvantages for each approach with regard to behavioral feature interfaces. Based on our insights, we present Subclack as a new approach for feature-module composition combining the advantages of all existing approaches. In our examples, we use contracts defined in an feature-oriented extension of the Java Modeling Language, and discuss how they can be checked by means of runtime assertions.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.13 [**Software Engineering**]: Reusable Software; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Design, Languages, Verification

Keywords

Feature-Oriented Programming, Behavioral Feature Interfaces, Design by Contract, Runtime Assertions, Java Modeling Language, Explicit Contract Refinement

MASPEGHI'13 July 01, 2013, Montpellier, France.

1. INTRODUCTION

Feature-oriented programming is an approach to modularize software according to the features it provides [25, 9]. A *feature* is a prominent or distinctive user-visible program characteristic [21]. Features often cut across multiple classes, and each class may contribute to multiple features. In feature-oriented programming, classes reside in *feature modules*. Technically, feature modules are rooted in mixin composition [17, 28]. By composing different subsets of feature modules, one can generate customized program variants automatically [25, 9].

Design by contract is an approach for the formal specification of program behavior [24], which can be used for program verification. Methods are specified by means of *method contracts*, defining preconditions required from callers, and postconditions ensured by callees. Furthermore, class invariants define properties that are assumed to hold before and after execution of public methods. An advantage of contracts compared to other specification techniques is that they can be used for documentation, runtime assertion checking, test-case generation, and formal verification [19, 12]. Furthermore, contracts facilitate blame assignment to locate faulty method implementations [24]. That is, we can assign blame to the caller if a method's precondition is violated, and to the callee if a method's postcondition is violated.

Applying design by contract to feature-oriented programming is beneficial to efficiently specify and verify feature modules. First, we modularize the *specification* of features similarly to the *implementation* of features, and compose feature specifications based on a feature selection to automatically generate the specification of a program variant [32, 27, 10]. Second, we exploit similarities between different program variants and their specifications to efficiently verify all compositions of feature modules [30, 31].

However, the location of faulty feature modules based on contracts is not straightforward. In previous work [10], we extended an existing tool for feature-oriented programming named FeatureHouse [3] with support for design by contract. With this extension, programmers can implement feature modules in Java, specify the modules with method contracts defined in an extension of the *Java Modeling Language* (JML) [13], and compose these modules automatically based on a selection of desired features. The result is a composed Java program including a composed specification in JML, which can be tested or verified by any given JML tool. How-

ever, identifying faulty feature modules is not always possible with our FeatureHouse extension. The reason is that methods may be composed from fragments defined in several feature modules, and it is not clear how to identify feature modules that violate contracts.

Our goal is to establish behavioral interfaces between feature modules, which we refer to as *behavioral feature interfaces*. To achieve this goal, we propose to translate contracts into program variants that check the expected behavior at the boundaries of feature modules. In experiments based on our FeatureHouse extension, we noticed problems when feature modules introduce new class invariants. While looking at other tools for feature-oriented programming, we found that each tool follows a slightly different semantics. Overall, we identified three approaches for the composition of feature modules. For each composition approach, we propose a strategy to generate contracts establishing behavioral feature interfaces. However, while all approaches are considered equivalent in the literature, we show that each approach has unique strengths and weaknesses when considering contracts. Based on our findings, we present *Subclack* as a new approach for the composition of feature modules. We demonstrate how to establish behavioral feature interfaces with Subclack and discuss how it combines the advantages of all other approaches.

In our examples, we use a feature-oriented extension of Java and JML, but our results can be generalized to other object-oriented languages and other specification languages supporting design by contract. We illustrate the different approaches for feature-module composition by means of runtime assertion checking, because it is an illustrative application of contracts. However, our results are also crucial for other verification techniques, because they often rely on a particular semantics of feature-oriented programming, and because each verification technique should establish behavioral feature interfaces to facilitate precise error location.

In summary, we make the following contributions:

- We illustrate that the semantics of feature-oriented programming is usually defined by means of tools. We identify and distinguish between three approaches for the composition of feature modules that are often considered as equivalent in the literature.
- We propose behavioral feature interfaces to identify faulty feature modules. We discuss how the identified approaches for the composition of feature modules can be extended to establish behavioral feature interfaces.
- We discuss problems of all three approaches for feature-module composition with respect to behavioral feature interfaces, and propose the new approach Subclack solving these problems.

2. FEATURE-ORIENTED PROGRAMMING

In feature-oriented programming, a *feature module* encapsulates the implementation of a feature [25, 9]. In tools for feature-oriented programming, such as FeatureC++ [5], Jampack [8], Mixin [8], FeatureHouse [3], and Classbox/J [11], a feature module encapsulates a set of classes and class refinements. A *class refinement* can add new members to a class or refine existing members. Class refinements are essentially mixins [17], and feature module composition is a form of mixin-layer composition [28]. Different program variants can be generated automatically by composing feature modules in different combinations.

Base

```
class Account {
  int balance = 0;
  int update(int x) { balance += x; return balance; }
}
```

DailyLimit

```
refines class Account {
  final int DAILY_LIMIT = -1000;
  int withdrawToday = 0;
  int update(int x) {
    if (withdrawToday + x < DAILY_LIMIT)
      throw new RuntimeException("out of limit");
    if (x < 0) withdrawToday += x;
    return original(x);
  }
  void newDay() { withdrawToday = 0; }
}
```

History

```
refines class Account {
  int lastTransaction = 0;
  int update(int x) {
    int result = original(x);
    lastTransaction = x;
    return result;
  }
}
```

Figure 1: Feature modules may introduce classes or refine classes introduced by other feature modules.

In Figure 1, we present a feature-oriented implementation of a bank account consisting of the features *Base*, *DailyLimit*, and *History*. Feature module *Base* introduces a class `Account` representing a simple bank account, storing the current balance in a field `balance` and allowing withdrawal or deposition of money using method `update`. Feature module *DailyLimit* applies a refinement to class `Account` (keyword `refines`). The class refinement introduces a field `withdrawToday` to store the total withdrawal of the day. Similarly, feature module *History* applies a refinement to class `Account` to store the last update operation in field `lastTransaction` and a refinement to method `update`. In a method refinement, keyword `original` refers to the old implementation of the method being refined.

Feature *Base* represents a valid program variant in itself and may be composed with feature *DailyLimit*, *History*, or both, which results in four variants that can be generated from these feature modules.[1] As said before, when composing feature modules, classes are introduced and class refinements are applied. There is no consensus on how class refinements should be applied. Actually, there are three different approaches for the composition of feature modules, to which we refer to as *Jampack*, *Subclassing*, and *Inlining*. In Table 1, we give an overview on the implementations of these approaches in tools and formalizations. Each of these approaches gives rise to a slightly different semantics of feature-oriented programming, as the semantics is usually defined by the code generation step involved.

Jampack Approach. All refinements of a class are composed into a single class in the generated program. The composed class then includes the code of the basic class as well as

[1]In our example, there is only a single base implementation that is subject to refinement (e.g., feature *Base*). However, feature-oriented programming also allows programmers to define alternative or optional feature modules as a basis.

Feature-module composition	Tools	Formalizations
Jampack approach	JAMPACK [8], FEATUREHOUSE [3]	—
Subclassing approach	FEATUREC++ [5], MIXIN [8]	Lightweight Feature Java [15]
Inlining approach	CLASSBOX/J [11]	Feature algebra [6], extended feature algebra [20]

Table 1: Approaches for feature-module composition in feature-oriented programming.

of all refinements introduced by other features. Methods and method refinements are each translated into methods of the composed class. All methods and method refinements being subject to refinement are renamed to have fresh names (i.e., the last method refinement of each refinement chain and non-refined methods keep their names). Each `original` call is replaced by a call to the refined method. In Figure 2a, we show the result of composing the features *Base*, *DailyLimit*, and *History* of Figure 1 with the Jampack approach. Class `Account` of feature *Base* and its refinements in feature *DailyLimit* and feature *History* are translated into one compound class `Account`. The method `update` from feature *Base* is renamed to `update$$Base`, and the `original` call in the method refinement in feature *DailyLimit* is replaced by a call to `update$$Base`. Analogously, method `update` from feature *DailyLimit* is renamed to `update$$DailyLimit`. The Jampack approach is used by the tool JAMPACK of the AHEAD tool suite [8] and by the tool FEATUREHOUSE [3].

Subclassing Approach. Each class refinement is translated into an ordinary class; classes and their refinements are connected via inheritance, such that a method refinement overrides the refined method and replaces this method by means of dynamic method dispatch. All classes and class refinements that are subject to refinement are renamed using a fresh name, and marked as `abstract` to indicate that they are not intended to be instantiated. The `original` call is translated into a call to the superclass method. In Figure 2b, we show the result of composing the features *Base*, *DailyLimit*, and *History* with the Subclassing approach: class `Account` of feature *Base* as well as the refinements of the features *DailyLimit* and *History* have been translated into separate classes; the class in feature *Base* is renamed into `Account$$Base` and the class implementing the refinement of feature *DailyLimit* inherits from class `Account$$Base`. The class in feature *DailyLimit* is renamed analogously. The class refinement from feature *History* is not renamed, because there is no later refinement in our example. The Subclassing approach is formalized in Lightweight Feature Java [15], and used by the tools FEATUREC++ [5] and MIXIN of the AHEAD tool suite [8].

Inlining Approach. Much like in the Jampack approach, all refinements of a class and the class itself are composed into a single class in the generated program, whereas, in contrast to the Jampack approach, each method is merged with all method refinements using method inlining. In Figure 2c, we show the result of composing the features *Base*, *DailyLimit*, and *History* with the Inlining approach. The composition result is similar to the result of the Jampack approach except that the methods `update$$Base` and `update$$DailyLimit` are being inlined in method `update`. A design decision of the Inlining approach is whether method refinements can access local variables of the methods they wrap or not. Accordingly, renaming of local variables might be required. In our example, the result does not depend on

(a) Jampack approach

```
class Account {
  int balance = 0;
  final int DAILY_LIMIT = -1000;
  int withdrawToday = 0;
  int lastTransaction = 0;
  private int update$$Base(int x) {
    balance += x; return balance;
  }
  int update$$DailyLimit(int x) {
    if (withdrawToday + x < DAILY_LIMIT)
      throw new RuntimeException("out of limit");
    if (x < 0) withdrawToday += x;
    return update$$Base(x);
  }
  int update(int x) {
    int result = update$$DailyLimit(x);
    lastTransaction = x;
    return result;
  }
  void newDay() { withdrawToday = 0; }
}
```

(b) Subclassing approach

```
abstract class Account$$Base {
  int balance = 0;
  int update(int x) { balance += x; return balance; }
}
abstract class Account$$DailyLimit
    extends Account$$Base {
  final int DAILY_LIMIT = -1000;
  int withdrawToday = 0;
  int update(int x) {
    if (withdrawToday + x < DAILY_LIMIT)
      throw new RuntimeException("out of limit");
    if (x < 0) withdrawToday += x;
    return super.update(x);
  }
  void newDay() { withdrawToday = 0; }
}
class Account extends Account$$DailyLimit {
  int lastTransaction = 0;
  int update(int x) {
    int result = super.update(x);
    lastTransaction = x;
    return result;
  }
}
```

(c) Inlining approach

```
class Account {
  int balance = 0;
  final int DAILY_LIMIT = -1000;
  int withdrawToday = 0;
  int lastTransaction = 0;
  int update(int x) {
    if (withdrawToday + x < DAILY_LIMIT)
      throw new RuntimeException("out of limit");
    if (x < 0) withdrawToday += x;
    balance += x;
    int result = balance;
    lastTransaction = x;
    return result;
  }
  void newDay() { withdrawToday = 0; }
}
```

Figure 2: Composition of the feature modules *Base*, *DailyLimit*, and *History* with different approaches.

this design decision, as there are no local variables affected. The Inlining approach with renaming of local variables is discussed for Feature Algebra [6]. The Inlining approach with the access of local variables is discussed for Extended Feature Algebra [20] and applied in CLASSBOX/J [11].

3. CONTRACTS IN FEATURE MODULES

Design by contract has been introduced to increase the reliability of object-oriented programs by support for specifying behavioral interfaces [24]. Behavioral interfaces between objects or classes document their intended behavior and ease the localization of errors. Enriching syntactic interfaces with behavioral properties defined in contracts enables blame assignment [19]. If the precondition of a method is not established, the caller is blamed, and if the postcondition of a method is not established, the callee is blamed. One way to achieve blame assignment is to translate contracts into assertions that are checked at runtime [24, 19, 12]. A runtime assertion is inserted at the beginning of each method body to check the precondition and all invariants of the class. Similarly, the postcondition and all invariants are checked when the method execution ends.

JML is a formal specification language supporting design by contract in Java [13]. In Figure 3, we added JML annotations to our running example. Feature *Base* is a valid Java program; JML annotations are given in comments. Class `Account` contains an invariant stating that the balance of the account must not be negative. Furthermore, method `update` is annotated with a contract: the precondition states that the method should only be called when the invariant is fulfilled, the postcondition states that the method correctly updates the balance and returns the updated balance. More language constructs of JML are discussed elsewhere [13].

In prior work, we proposed and discussed five approaches to define contracts for feature modules [32]. Here, we use explicit contract refinement, because it subsumes all others [32]: *explicit contract refinement* allows programmers to define contracts for methods and method refinements, while preconditions and postconditions of refined methods can be accessed using keyword `original`. The semantics of `original` in contracts is similar to the semantics within the method body; when composing features the keyword is replaced by the precondition or postcondition of the refined method. Thus, a feature may not only introduce new fields and methods, but also new invariants and contracts, which only need to be fulfilled if the feature is selected.

In Figure 3, we give JML specifications for the features *DailyLimit* and *History*. Feature *DailyLimit* introduces a new invariant stating that the value of the new field `withdrawToday` does not exceed the limit. Furthermore, the precondition of method `update` is strengthened to maintain the invariant. In feature *History*, the postcondition of method `update` is strengthened to express that the last transaction is stored correctly; the precondition is not refined.

In feature *DailyLimit*, we slightly changed the implementation of method `update` compared to Figure 1. The reason is that contracts help to avoid defensive programming.[2] In this example, we formulated a precondition requiring callers

[2]Defensive programming is a practice *"to protect every software module against the slings and arrows of outrageous fortune"* and *"to include as many checks as possible, even if they are redundant with checks made by callers"* [24].

Base

```
class Account {
  //@ invariant balance_non_negative: balance >= 0;
  int balance = 0;
  //@ requires balance + x >= 0;
  //@ ensures balance == \old(balance) + x &&
  //@   \result == balance;
  int update(int x) { balance += x; return balance; }
}
```

DailyLimit

```
refines class Account {
  final int DAILY_LIMIT = -1000;
  //@ invariant withdraw_in_limit:
  //@   withdrawToday >= DAILY_LIMIT;
  int withdrawToday = 0;
  //@ requires original &&
  //@   withdrawToday + x >= DAILY_LIMIT;
  //@ ensures original;
  int update(int x) {
    if (x < 0) withdrawToday += x;
    return original(-x);
  }
  void newDay() { withdrawToday = 0; }
}
```

History

```
refines class Account {
  int lastTransaction = 0;
  //@ requires original;
  //@ ensures original && lastTransaction == x;
  int update(int x) {
    int result = original(x);
    lastTransaction = x;
    return result;
  }
}
```

Figure 3: In explicit contract refinement, contracts can be defined for methods and method refinements; contracts for method refinements can refer to the original precondition or postcondition.

to make sure that the daily withdrawal is within the limit, and thus the branching statement throwing an exception is dispensable. While one could argue against the design in our example, it is well accepted that contracts are a means against defensive programming [24].

Contracts in feature modules are beneficial for several reasons. First, we can precisely define formal specifications in addition to ambiguous, informal specifications such as with JAVADOC. Second, we can avoid redundant checks as in defensive programming that unnecessarily bloat the source code. Third, contracts may be used to check feature modules for correctness by means of runtime assertion checking, test-case generation, static analysis [27], or deductive verification [31, 30]. Finally, contracts can be used to achieve blame assignment for features, as we detail next.

4. BEHAVIORAL FEATURE INTERFACES

Design by contract can be used to establish behavioral interfaces between classes [19]. All method contracts and class invariants defined for a particular class constitute the *behavioral class interface*. It can be understood as an extension of the syntactic class interface. The latter is given by the union of all class members and their signatures. Typically, conformance to syntactic class interfaces is checked using compilers, whereas conformance to behavioral class interfaces is checked by means of runtime assertion checking, test-case generation, static analysis, or deductive verification [19, 12].

However, behavioral class interfaces are not sufficient to locate faulty feature modules, because several feature mod-

ules may contribute to a given class or method. We propose to generalize the notion of behavioral interfaces to features. The union of all method contracts and class invariants defined in a particular feature module constitute the *behavioral feature interface*. When checking the conformance of feature modules to behavioral feature interfaces, it is necessary to check whether each method call from one feature module to another feature module establishes all contracts. As mentioned, with these checks, we can lift blame assignments from methods and classes to feature modules, and thus identify faulty feature modules.

For illustration, consider our example in Figure 3 again. Compared to Figure 1, we deliberately introduced a bug in the body of method `update` defined in feature module *DailyLimit*: `original(x)` is replaced by `original(-x)`. With this change, we consider feature module *DailyLimit* as faulty, because it may violate the precondition of method `update` defined in feature module *Base*, even if the precondition of method `update` in feature module *DailyLimit* is fulfilled. Thus, feature module *DailyLimit* does not conform to the behavioral feature interface of *Base*.

When checking conformance of method refinements, it is not sufficient to only check contracts for each single method; to identify violations of behavioral feature interfaces, contracts between methods and all respective method refinements need to be checked, too. Assume we would compose all three feature modules shown in Figure 3. For the behavioral class interface of class `Account` it is sufficient to check a contract for method `update` that is composed from all three given contracts (composition is achieved by replacing the keyword `original` to respective preconditions and postconditions as introduced in Section 3). The behavioral class interface is sufficient from an outside perspective (i.e., for clients of that class). However, the behavioral feature interface consists of three contracts for method `update`, which are necessary for locating the faulty feature implementation, *DailyLimit* in our case.

5. COMPOSITION OF CONTRACTS

How behavioral feature interfaces can be utilized in feature-oriented programming, depends on the approach used for feature-modules composition. For each approach discussed in Section 2, we propose how to compose the corresponding contracts to establish behavioral feature interfaces as far as possible. These composed contracts can then be checked by means of runtime assertion checking. As we want to reuse existing tools for translating contracts to runtime assertions, such as JMLc [12], the resulting contracts need to be valid JML statements. Although we exemplify our approaches by means of runtime assertions, behavioral feature interfaces can also be used for static analysis or deductive verification. We discuss drawbacks of existing approaches for feature-module composition regarding behavioral feature interfaces, and present a new approach named *Subclack* based on our insights. All approaches are exemplified by the composition of feature modules *Base* and *DailyLimit*.

Jampack Approach. When using Jampack, all invariants written in any of the composed features are included in the generated classes (see Figure 4a). Furthermore, the contract of a generated method is obtained by recursively replacing all occurrences of `original` in the contract with the precondition or postcondition of the refined method. Contracts for renamed methods such as `update$$Base` are treated in the same way. Thus, violations of preconditions and postconditions of each method and method refinement can be recognized at runtime.

Problems occur regarding the scope of class invariants: refined and thus renamed methods may intentionally not satisfy all invariants of the class. For example, the method `update$$Base` in Figure 4a does not satisfy the invariant `withdraw_in_limit`, because feature *DailyLimit* introduces this invariant and refines the method `update` to actually satisfy the new invariant. While the refined method `update` satisfies this invariant, method `update$$Base` does not.

Thus, the question arises of how to specify that refined methods do not need to satisfy all invariants. In JML, the keyword `helper` expresses that a certain method does not need to fulfill invariants. However, when using keyword `helper` for all renamed methods, we lose parts of behavioral feature interfaces, because renamed methods no longer need to satisfy any invariants defined by previous features. For example, method `update$$Base` should not establish invariant `withdraw_in_limit` as it is introduced in a later feature, but the invariant `balance_non_negative` should be established. One solution to tackle this problem is to translate invariants of refined features directly to preconditions and postconditions of renamed methods, as shown in Figure 4a. This way, we can establish behavioral feature interfaces, but this may involve many additional preconditions and postconditions making the specification hard to read.

Subclassing Approach. Most specification approaches in object-oriented programming, and especially JML [13], implement specification inheritance [16]. *Specification inheritance* states that contracts of superclass and subclass methods are conjoined such that each given contract must be fulfilled by the subclass' method. Furthermore, specification inheritance forces subclasses to fulfill all invariants defined in superclasses. Specification inheritance is required when considering subtype polymorphism, because we may call a method on an instance of a certain class or any subclass thereof. In this case, we can call the method by establishing the precondition of any overridden method and can rely on the corresponding postcondition.

The problem with invariants in the Jampack approach can be avoided in the Subclassing approach by copying invariants to the generated class of the respective feature. In Figure 4b, we illustrate that invariant `balance_non_negative` is copied to class `Account$$Base`. Because of specification inheritance in JML, invariants need to be fulfilled by the source feature and all features providing a refinement to this class. For example, the classes `Account$$Base` and `Account` need to establish invariant `balance_non_negative`, whereas only `Account` needs to establish `withdraw_in_limit`.

While specification inheritance is reasonable for object-oriented programming, assuming specification inheritance between features, as generated by the Subclassing approach, is too restrictive. Sometimes features need to break contracts of previous features [32]. In Figure 4b, method `update` of class `Account` actually does not fulfill the contract defined for the superclass `Account$$Base`, since the method implementation in feature *DailyLimit* relies on the additional precondition stating that the daily withdrawal is within the limit. Hence, we cannot generate contracts for refined methods, and thus method `update` in class `Account$$Base` has no contract. As a consequence, we lose behavioral feature in-

(a) Jampack approach

```
class Account {
  //@ invariant balance_non_negative: balance >= 0;
  int balance = 0;
  final int DAILY_LIMIT = -1000;
  //@ invariant withdraw_in_limit:
  //@   withdrawToday >= DAILY_LIMIT;
  int withdrawToday = 0;
  //@ requires balance+x >= 0 && balance_non_negative;
  //@ ensures balance == \old(balance) + x &&
  //@   \result == balance && balance_non_negative;
  private /*@helper@*/ int update$$Base(int x) {
    balance += x; return balance;
  }
  //@ requires balance + x >= 0 &&
  //@   withdrawToday + x >= DAILY_LIMIT;
  //@ ensures balance == \old(balance) + x &&
  //@   \result == balance;
  int update(int x) {
    if (x < 0) withdrawToday += x;
    return update$$Base(-x);
  }
  void newDay() { withdrawToday = 0; }
}
```

(b) Subclassing approach

```
abstract class Account$$Base {
  //@ invariant balance_non_negative: balance >= 0;
  int balance = 0;
  int update(int x) { balance += x; return balance; }
}
class Account extends Account$$Base {
  final int DAILY_LIMIT = -1000;
  //@ invariant withdraw_in_limit:
  //@   withdrawToday >= DAILY_LIMIT;
  int withdrawToday = 0;
  //@ requires balance + x >= 0 &&
  //@   withdrawToday + x >= DAILY_LIMIT;
  //@ ensures balance == \old(balance) + x &&
  //@   \result == balance;
  int update(int x) {
    if (x < 0) withdrawToday += x;
    return super.update(-x);
  }
  void newDay() { withdrawToday = 0; }
}
```

(c) Inlining approach

```
class Account {
  //@ invariant balance_non_negative: balance >= 0;
  int balance = 0;
  final int DAILY_LIMIT = -1000;
  //@ invariant withdraw_in_limit:
  //@   withdrawToday >= DAILY_LIMIT;
  int withdrawToday = 0;
  //@ requires balance + x >= 0 &&
  //@   withdrawToday + x >= DAILY_LIMIT;
  //@ ensures balance == \old(balance) + x &&
  //@   \result == balance;
  int update(int x) {
    if (x < 0) withdrawToday += x;
    x = -x;
    //@ assert balance + x >= 0;
    //@ assert balance_non_negative;
    //@ ghost int oldBalance = balance;
    balance += x;
    //@ assert balance == oldBalance + x;
    //@ assert balance_non_negative;
    return balance;
  }
  void newDay() { withdrawToday = 0; }
}
```

Figure 4: Composition of the features *Base* and *DailyLimit* with different approaches, each with a different strategy to compose contracts.

Approach	Method contracts	Class invariants
Jampack	+	−
Subclassing	−	+
Inlining	−	−
Subclack	+	+

Table 2: Approaches for feature-module composition and support for behavioral feature interfaces.

terfaces defined in terms of method contracts, and we may not be able to locate faulty features.

Inlining Approach. The composition of contracts using the Inlining approach is similar to that presented for Jampack. However, a difference is that refined methods do not appear in the derived Java program as distinct methods. To establish behavioral feature interfaces, we propose to include JML assertions into the merged body of a method. We use the JML keyword `assert` for this, which is intended to express any properties that should hold between two statements. Using `assert`, we can include the preconditions and postconditions of all methods in the refinement chain.

The Inlining approach has a problem with invariants similar to Jampack. Class invariants are only checked before and after executing composed methods, but class invariants are a part of the behavioral interface of a feature and should also be checked. Hence, we propose to insert further assertions to check the class invariant right before and after each inlined method of that class. In Figure 4c, we inserted `assert balance_non_negative` for the invariant defined in feature *Base* to check that the method refinement in *DailyLimit* does not violate this invariant. A problem with these assertions is that we need copies of them, if there are several calls to the original implementation, making the source code hard to read, even worse than for the Jampack approach.

Subclack Approach. On the one hand, the Jampack approach and the Inlining approach are problematic, as the visibility of class invariants is not supported. Though, we can introduce additional preconditions, postconditions, and assertions, they limit the readability of the specifications due to exhaustive specification cloning. The readability is an issue, as a programmer needs to understand a specification to identify a faulty feature upon violations of the runtime assertions. On the other hand, the Subclassing approach has the problem that we can only check the contract of the last method refinement, because specification inheritance is too restrictive for feature-oriented method refinement. In Table 2, we summarize which approaches provide good support for method contracts and class invariants.

To avoid both problems, we propose the *Subclack* approach.[3] It translates classes and their refinements to a class hierarchy with inheritance similar to the Subclassing approach, to achieve the desired visibility of class invariants. In contrast to the Subclassing approach, refined methods are renamed as in the Jampack approach, because then method refinements no longer need to support specification inheritance. The reason is that specification inheritance only applies to method overriding in object-oriented programming (i.e., methods in superclasses and subclasses with the same name and signature).

[3]Subclack is a portmanteau of *Subcl*assing and Jamp*ack*.

```
abstract class Account$$Base {                    Subclack approach
  //@ invariant balance_non_negative: balance >= 0;
  int balance = 0;
  //@ requires balance + x >= 0;
  //@ ensures balance == \old(balance) + x &&
  //@   \result == balance;
  int update$$Base(int x) {
    balance += x;
    return balance;
  }
}
class Account extends Account$$Base {
  //@ invariant withdraw_in_limit:
  //@   withdrawToday >= DAILY_LIMIT;
  int withdrawToday = 0; final int DAILY_LIMIT = -1000;
  //@ requires balance + x >= 0 &&
  //@   withdrawToday + x >= DAILY_LIMIT;
  //@ ensures balance == \old(balance) + x &&
  //@   \result == balance;
  int update(int x) {
    if (x < 0) withdrawToday += x;
    return update$$Base(-x);
  }
  void newDay() { withdrawToday = 0; }
}
```

Figure 5: Composition of features *Base* and *DailyLimit* with the Subclack approach.

In Figure 5, we illustrate the Subclack approach using our running example. The result of composing the features *Base* and *DailyLimit* is similar to the result of the Subclassing approach. The subtle difference is that method `update` from feature *Base* is renamed into `update$$Base`. Because the method has a different name than its refinement in class `Account`, specification inheritance does not apply to this method. Consequently, we can add a method contract to method `update$$Base`, which need not to be established before and after a call to method `update`. Instead, the added contract for method `update$$Base` needs to be fulfilled only when the method refinement of feature *DailyLimit* calls the method as defined in feature *Base*. Hence, we establish a behavioral interface between the features *DailyLimit* and *Base*. If feature *DailyLimit* does not fulfill the precondition of method `update$$Base`, we assign blame to feature *DailyLimit*. Similarly, if feature *Base* does not fulfill the postcondition, we identify feature *Base* as faulty.

Using the Subclack approach, class invariants apply to the class refinement (or class introduction) that introduced the invariant as well as to all subsequent class refinements. There is no need to generate any additional preconditions, postconditions, or assertions to check class invariants. At the same time, we check method contracts between features, and establish behavioral feature interfaces allowing us to locate faulty feature implementations.

6. RELATED WORK

Thüm et al. propose five approaches to define contracts for feature modules [32]. We discuss behavioral feature interfaces in the context of explicit contract refinement, because this approach is more flexible than others. We noticed an interesting connection between approaches for feature-module composition and approaches to define contracts: consecutive contract refinement is an approach to define contracts in feature modules assuming specification inheritance between features. For consecutive contract refinement, the Subclassing approach does not have any drawbacks. Similarly, others assume specification inheritance for aspect-oriented programming [23, 1] and delta-oriented programming [18]. However, we found that specification inheritance is often too restrictive to specify feature modules [32].

Beside discussions how to specify feature modules, researchers investigated how to check the conformance of feature modules to contracts by means of model checking [7], static analysis [27], and theorem proving [31, 30, 14]. All these approaches focus on behavioral class interfaces rather than behavioral feature interfaces, and thus cannot always locate faulty feature modules.

Before checking the conformance of feature modules to behavioral feature interfaces, it is useful to check syntactic feature interfaces. Type systems were proposed to check type safety in all possible combinations of the feature modules [29, 15, 2]. Similarly, Apel et al. discuss modifiers restricting the access across feature modules syntactically [4].

Others apply design by contract to aspect-oriented programming. Klaeren et al. define propositional dependencies between aspects in assertions [22], whereas we focus on arbitrary specifications. Zhao and Rinard discuss aspect invariants that are local to a single aspect [33], whereas we define invariants in feature modules such that they are added to existing classes and must be established by subsequent class refinements. Lorenz and Skotiniotis [23] propose three advice categories with respective strategies for runtime-assertion generation, selected based on a static analysis: *agnostic* and *obedient* disallowing to refine method contracts, and *rebellious* allowing refinements analogous to specification inheritance. Similarly, Agostinho et al. propose to check contracts of the base system before and after each advice [1]. Contrary to both approaches, we allow arbitrary refinements, the runtime-assertion strategy is the same for all method refinements, and we allow alternative or optional base feature modules. Rebêlo et al. discuss gray-box contracts that allow to specify which methods are called within a piece of advice [26], whereas we consider black-box contracts.

7. CONCLUSIONS AND FUTURE WORK

We discussed Jampack, Subclassing, and Inlining as alternative approaches for the composition of feature modules in feature-oriented programming. We illustrated how to generate Java programs annotated with JML specifications for each composition approach. The generated JML specifications can then be used by existing tools for runtime assertion checking, test-case generation, or formal verification.

We discussed problems of all three approaches concerning the visibility of contracts and invariants in inheritance hierarchies. Based on these insights, we proposed the Subclack approach for the composition of feature modules with contracts, which combines the advantages of Subclassing and Jampack. Contrary to previous approaches, the Subclack approach supports behavioral feature interfaces without the need to clone specifications. Behavioral feature interfaces can be used to identify faulty feature modules.

In future work, we plan to provide tool support for the Subclack approach and behavioral feature interfaces based on FeatureHouse. Preliminary analyses of examples have shown that the Subclack approach is useful when applying design by contract to feature-oriented programming. However, further evaluation is required to qualitatively and quantitatively assess the benefits of the Subclack.

Acknowledgments

We thank Martin Kuhlemann, Marko Rosenmüller, Don Batory, and Fabian Benduhn for interesting discussions and the anonymous reviewers for comments on earlier drafts. This work is partially funded by the German Research Foundation (SA 465/34-2, AP 206/2, AP 206/4, and MO 690/7-2).

8. REFERENCES

[1] S. Agostinho, A. Moreira, and P. Guerreiro. Contracts for Aspect-Oriented Design. In *SPLAT*, pages 1:1–1:6. ACM, 2008.

[2] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type Safety for Feature-Oriented Product Lines. *ASE*, 17(3):251–300, 2010.

[3] S. Apel, C. Kästner, and C. Lengauer. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *TSE*, 39(1):63–79, 2013.

[4] S. Apel, S. S. Kolesnikov, J. Liebig, C. Kästner, M. Kuhlemann, and T. Leich. Access Control in Feature-Oriented Programming. *SCP*, 77(3):174–187, 2012.

[5] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *GPCE*, volume 3676 of *LNCS*, pages 125–140. Springer, 2005.

[6] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An Algebraic Foundation for Automatic Feature-Based Program Synthesis. *SCP*, 75(11):1022–1047, 2010.

[7] S. Apel, W. Scholz, C. Lengauer, and C. Kästner. Detecting Dependences and Interactions in Feature-Oriented Design. In *ISSRE*, pages 161–170. IEEE, 2010.

[8] D. Batory. A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite. In *Proc. Generative and Transformational Techniques in Software Engineering*, pages 3–35. Springer, 2006.

[9] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *TSE*, 30(6):355–371, 2004.

[10] F. Benduhn. Contract-Aware Feature Composition. Bachelor's thesis, University of Magdeburg, Germany, 2012.

[11] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/J: Controlling the Scope of Change in Java. In *OOPSLA*, pages 177–189. ACM, 2005.

[12] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An Overview of JML Tools and Applications. *STTT*, 7(3):212–232, 2005.

[13] P. Chalin, J. Kiniry, G. T. Leavens, and E. Poll. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In *FMCO*, volume 4111 of *LNCS*, pages 342–363. Springer, 2005.

[14] F. Damiani, O. Owe, J. Dovland, I. Schaefer, E. B. Johnsen, and I. C. Yu. A Transformational Proof System for Delta-Oriented Programming. In *FMSPLE*, pages 53–60. ACM, 2012.

[15] B. Delaware, W. Cook, and D. Batory. Fitting the Pieces Together: A Machine-Checked Model of Safe Composition. In *ESECFSE*, pages 243–252. ACM, 2009.

[16] K. K. Dhara and G. T. Leavens. Forcing Behavioral Subtyping through Specification Inheritance. In *ICSE*, pages 258–267. IEEE, 1996.

[17] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and Mixins. In *POPL*, pages 171–183. ACM, 1998.

[18] R. Hähnle and I. Schaefer. A Liskov Principle for Delta-Oriented Programming. In *ISOLA*, volume 7609 of *LNCS*, pages 32–46. Springer, 2012.

[19] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. Parkinson. Behavioral Interface Specification Languages. *CSUR*, 44(3):16:1–16:58, 2012.

[20] P. Höfner and B. Möller. An Extension for Feature Algebra. In *FOSD*, pages 75–80. ACM, 2009.

[21] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990.

[22] H. Klaeren, E. Pulvermueller, A. Rashid, and A. Speck. Aspect Composition Applying the Design by Contract Principle. In *GCSE*, volume 2177 of *LNCS*, pages 57–69. Springer, 2001.

[23] D. H. Lorenz and T. Skotiniotis. Extending Design by Contract for Aspect-Oriented Programming. *CoRR*, abs/cs/0501070, 2005.

[24] B. Meyer. Applying Design by Contract. *Computer*, 25(10):40–51, 1992.

[25] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *ECOOP*, volume 1241 of *LNCS*, pages 419–443. Springer, 1997.

[26] H. Rebêlo, G. T. Leavens, R. M. F. Lima, P. Borba, and M. Ribeiro. Modular Aspect-Oriented Design Rule Enforcement with XPIDRs. In *FOAL*, pages 13–18. ACM, 2013.

[27] W. Scholz, T. Thüm, S. Apel, and C. Lengauer. Automatic Detection of Feature Interactions using the Java Modeling Language: An Experience Report. In *FOSD*, pages 7:1–7:8. ACM, 2011.

[28] Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *TOSEM*, 11(2):215–255, 2002.

[29] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *GPCE*, pages 95–104. ACM, 2007.

[30] T. Thüm, I. Schaefer, S. Apel, and M. Hentschel. Family-Based Deductive Verification of Software Product Lines. In *GPCE*, pages 11–20. ACM, 2012.

[31] T. Thüm, I. Schaefer, M. Kuhlemann, and S. Apel. Proof Composition for Deductive Verification of Software Product Lines. In *VAST*, pages 270–277. IEEE, 2011.

[32] T. Thüm, I. Schaefer, M. Kuhlemann, S. Apel, and G. Saake. Applying Design by Contract to Feature-Oriented Programming. In *FASE*, volume 7212 of *LNCS*, pages 255–269. Springer, 2012.

[33] J. Zhao and M. C. Rinard. Pipa: A Behavioral Interface Specification Language for AspectJ. In *FASE*, volume 2621 of *LNCS*, pages 150–165. Springer, 2003.