Applying Design by Contract to Feature-Oriented Programming

Thomas Thüm[1], Ina Schaefer[2], Martin Kuhlemann[1], Sven Apel[3], and Gunter Saake[1]

[1] University of Magdeburg, Germany
[2] University of Braunschweig, Germany
[3] University of Passau, Germany

Abstract. *Feature-oriented programming* (FOP) is an extension of object-oriented programming to support software variability by *refining* existing classes and methods. In order to increase the reliability of all implemented program variants, we integrate *design by contract* (DbC) with FOP. DbC is an approach to build reliable object-oriented software by specifying methods with contracts. Contracts are annotations that document and formally specify behavior, and can be used for formal verification of correctness or as test oracles. We present and discuss five approaches to define contracts of methods and their refinements in FOP. Furthermore, we share our insights gained by performing five case studies. This work is a foundation for research on the analysis of feature-oriented programs (e.g., for verifying functional correctness or for detecting feature interactions).

1 Introduction

Feature-oriented programming (FOP) [21,7] is a programming paradigm supporting software variability by modularizing object-oriented programs along the features they provide. A feature is an end-user-visible program behavior [15]. Code belonging to a feature is encapsulated in a feature module. A feature module can introduce classes or modify existing classes by adding or refining fields and methods. A program variant is generated by composing the feature modules of the desired features. We use formal methods to increase the reliability of all program variants that can be generated from a set of feature modules.

Design by contract (DbC) [20] has been proposed as a means to obtain reliable object-oriented software. The key idea is to specify each method with a contract consisting of a precondition and a postcondition. The precondition formulates assumptions of the method that the caller of the method has to ensure. The postcondition provides guarantees that the method gives such that the caller can rely on it. Additionally, class invariants specify properties of objects that hold before and must hold after a method call. DbC can be used for formal specification and documentation of program behavior as well as for formal verification or testing of functional correctness. We integrate DbC with FOP to increase the reliability of FOP.

```
class Array {                                                          Base
  Item[] data; //@ invariant data != null;
  Array(Item[] data) { this.data = data; }
  /*@ requires \nothing;
    @ ensures (\forall int i; 0 < i && i < data.length;
    @   data[i-1].key <= data[i].key); @*/
  void sort() { /* heap sort algorithm */ }
}
class ArrayWithInverse extends Array { /* ... */ }
class Item {
  int key; Object value; //@ invariant value != null;
  Item(key, value) { this.key = key; this.value = value; }
}
```

Fig. 1. *Design by contract with Java and JML*: method contracts and class invariants are embedded in comments.

FOP adds another dimension of modularization and code reuse to object-oriented programs besides inheritance. While in class-based inheritance, subclasses must satisfy the behavioral subtyping principle [17], method refinement (i.e., method overriding in FOP) is different in nature from code reuse by inheritance. A feature may change the behavior of an existing method arbitrarily to meet feature-specific requirements. For example, a security feature may restrict the allowed parameter values of a method by strengthening the precondition. Thus, when integrating DbC with FOP, the question arises how method contracts of refined methods should be defined.

We present and discuss five new approaches to specify contracts of methods which we refine using FOP. We consider the strengths and weaknesses of each approach with respect to strictness, expressiveness, complexity, and specification clones. Furthermore, we discuss the refinement of class invariants and evaluate the practical applicability of the presented approaches using five case studies. This paper is the first to focus on the specification of feature-oriented programs using DbC. Previous work focused on ensuring consistency of feature-oriented programs using type checking [3,10] and model checking [5,6]. With our systematic analysis of the different approaches to specify feature-oriented programs using DbC, we provide the foundation for future research on the formal analysis of feature-oriented programs, including the formal verification of functional correctness, feature interaction detection, and test case generation.

2 Background

Figure 1 shows our running example — a Java program that is annotated with the *Java Modeling Language (JML)* [16] to specify its behavior using DbC. Class `Array` is specified by an invariant (using the keyword `invariant`) that states that field `data` should not be null. Invariants have to be established by the class constructors, they can be assumed before every method call and have to be reestablished afterwards. Method `sort()` of class `Array` is specified by a method contract. The precondition of the contract is expressed in the `requires` clause

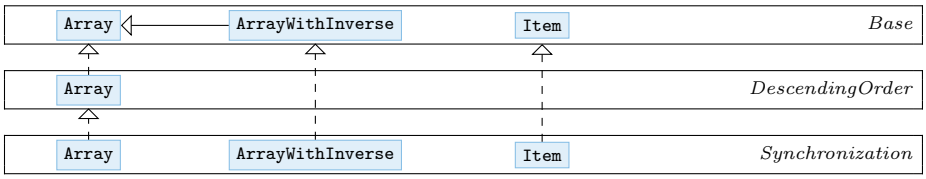

Fig. 2. Feature-oriented class refinement (dashed arrows) and object-oriented inheritance (solid arrows) are concepts for reuse that are orthogonal to each other.

and has to be ensured by the caller of the method. Here, the precondition is simply true. In JML, behavioral subtyping [17] for subclasses is achieved by specification inheritance. This means that all subclasses inherit the invariants of their superclasses and that overriding methods must also satisfy the contracts of the overridden methods. The `ensures` clause expresses the postcondition of a contract and has to be guaranteed by the method. In our example, the postcondition states that the resulting array is sorted. Contracts can also be denoted by Hoare triples [13]. Given a method m with precondition ϕ and postcondition ψ, the contract of method m is denoted by $\{\phi\}m\{\psi\}$.

Feature-oriented programming (FOP) is an extension of object-oriented programming (OOP) aiming at better reuse capabilities across families of object-oriented programs [21]. Classes are split into pieces distributed over feature modules; modules that implement end-user-visible features. A particular program can be derived automatically by combining the feature modules of the required features [2]. A feature module can introduce new classes, methods, and fields. If a method with a particular name already exists in a previously composed feature module, the existing method is refined [2]. Method refinement is similar to overriding with object-oriented inheritance, but the FOP keyword `original` is used instead of `super`. The main difference is that `original` is bound at the time the feature modules are composed. Figure 2 visualizes the FOP refinement of the classes of Figure 1 (`Array`, `ArrayWithInverse`, `Item`) with the feature modules *Base*, *DescendingOrder*, and *Synchronization*. *Base* contains the classes `Array`, `ArrayWithInverse`, and `Item`. *DescendingOrder* contains a class refinement `Array` which refines class `Array` of *Base* to invert the sorting order of implemented arrays. *Synchronization* contains refinements for all classes of *Base*; as a result, these classes support multithreading.

3 Contracts for Feature-Oriented Programming

We present five approaches for the integration of DbC into FOP and discuss advantages and disadvantages of each approach.

Plain Contracting The application of DbC to FOP should be as simple as possible to facilitate creation and maintenance of contracts for programmers.

```
refines class Array {                                                    StableSort
  /*@ requires original;
    @ ensures original && ...sorting is stable...; @*/
  void sort() { /* merge sort algorithm */ }
}
```

Fig. 3. *Explicit contract refinement*: feature *StableSort* overrides method `sort()` with an implementation of a stable sorting algorithm. Both, precondition and postcondition maintain the refined contract indicated by the keyword `original` and refine it.

Plain contracting is the simplest possible approach allowing programmers to define contracts only for method introductions and not for method refinements. As a consequence, method refinements may not change the behavior of the refined method. Consider the example in Figure 1. Assume that an additional feature *Quicksort* refines the class `Array` by overriding the body of method `sort()` with a Quicksort implementation. The contract of method `sort()` does not have to be changed, because the new implementation does not affect sorting. Given a set of selected features and a total order on those features, a tool can decide for every method whether it is a method introduction or a method refinement [3]. Then, we can automatically check that no method refinement comes with a contract.

On the one hand, allowing programmers to introduce, but not to refine contracts comes with advantages. First, we only need to specify a method once even if it is refined by several other feature modules, and thus the effort for specification (i.e., writing contracts) is minimal. Second, the source code is easier to understand as the same contract holds in every possible combination of features. This is beneficial since a programmer needs to know the contract for every called method (e.g., to find out whether the precondition is fulfilled at every position where the method is called). On the other hand, this approach might be too restrictive. With plain contracting, we are not able to specify feature-oriented programs, where the refinement of a method also requires the refinement of a contract. For instance, if we replace an instable sorting algorithm with a stable one, we may need to express that callers can rely on this property if the according feature is present. In Section 6, we evaluate whether this restriction is an actual problem in practice.

Explicit Contract Refinement When refining a method, we may also need to refine the corresponding contract if the method behavior is changed such that it no longer satisfies the original contract. The refinement of contracts can be supported by the same linguistic means as method refinement, which should raise the acceptance of DbC in FOP. Explicit contract refinement allows programmers to use the keyword `original` to refer the refined precondition and postcondition in the contract refinement.

As an example for explicit contract refinement, in Figure 3, we assume that feature *Base* is identical to the previous example and that a new feature *StableSort* replaces the sorting algorithm by a stable sorting algorithm; here, merge

sort. In order to provide a contract, which states that the result is sorted and the algorithm is stable, we refer to the existing postcondition and conjoin it with a definition of stability (which we left out for brevity). Keyword `original` may appear anywhere in the precondition or postcondition (not necessarily at the beginning) or it may not appear at all.

Explicit contract refinement is a flexible approach where contracts can be refined by including the previous contract if appropriate; preconditions and postconditions can be refined individually. However, the approach may lead to complex and less understandable specifications, especially, when several refinements for the same method contract exist and some, but not all refinements, refer to the previous contract. It may be unclear what a method actually needs to ensure and what it can rely on, because this may depend on the feature selection. In particular, contracts depend on the feature from which the method is called.

Consecutive Contract Refinement Consecutive contract refinement is an approach with which new contracts can be defined for method refinements but contracts for refined methods may not be invalidated. The central idea of the approach is to adapt contract subtyping to FOP. Contract subtyping is widely used in OOP and ensures that contracts defined in a certain class must be fulfilled in all subclasses, too. The main difference to contract subtyping in OOP is that features may be present or not, and thus the feature selection influences the resulting method contract.

Given an original method m with precondition ϕ and postcondition ψ, we can refine m with a new method implementation m' with precondition ϕ' and postcondition ψ'. Then, the refined method m' needs to ensure the original contract $\{\phi\}m'\{\psi\}$ and the new contract $\{\phi'\}m'\{\psi'\}$. As a result, the method can be used in all places where method m is called, and the caller can rely on the contract of the refined method m. For example, re-consider the feature *StableSort* in Figure 3. With consecutive contract refinement, the example would look the same except for the replacement of '`original`' with '`true`' in the precondition and postcondition, because the contract of the refined method holds implicitly.

The main advantage of consecutive contract refinement compared to explicit contract refinement is that existing contracts remain valid even if a method is refined. This way, callers can rely on contracts defined in a particular feature independent on the presence of other features, because refinements cannot invalidate the contract. This advantage comes with a reduced applicability, since we cannot encode cases in which a feature weakens an existing contract.

Contract Overriding Contract overriding is a special case of explicit contract refinement where the keyword `original` is never used. Contract overriding allows programmers to replace the contract when refining a method, but does not allow programmers to reference or reuse refined contracts. In contrast to consecutive contract refinement, contracts defined in previous features do not need to be fulfilled. In previous work, we used contract overriding to verify SPL products by proof composition [23]. In this previous work, we additionally enforced

compatibility between contracts and their refinements. A contract refinement is compatible to a previous contract, if every method that fulfills the refined contract also satisfies the contract of the refined method.

The main problem with contract overriding are specification clones, because there is no way to adapt original contracts. The CPA (copy, paste, adapt) principle is the only option to refine contracts, which may result in many specification clones and, thus, a high specification effort. Another serious disadvantage is that the meaning of a contract is unclear for callers, because it heavily depends on the actual feature selection and on the composition ordering. Furthermore, if two features refine the same method contract using contract overriding, we may get undesired contracts if both features are selected (known as feature interaction problem of FOP [2]). We could introduce derivative contracts (i.e., a contract that is only included if two or more features are selected) but derivative contracts can introduce further specification clones.

Pure-Method Refinement Preconditions and postconditions in JML may also contain calls to methods that are free of side-effect and are guaranteed to terminate (known as *pure* method in JML [16]). If a pure method is used in a contract, the contract depends on the result of this (pure) method call. Pure methods called in contracts open a further possibility for contract refinement, because pure methods can be refined as any other method in FOP – this allows programmers to refine contracts as a spin-off. With pure-method refinement, instead of actually refining a contract itself, a pure method used in a contract is refined and, thus, indirectly contracts based on the feature selection are modified.

In Figure 4, the example of pure-method refinement is based on an publicly available case study[4], which we have decomposed into features. Class `ExamDataBase` stores the results of student exams. Array `students` contains the students and their points, whereas a `null`-value refers to a free position in the array. The method `consistent()` checks whether all students have at least zero points. The method `validStudent()` is used in the contract of method `consistent()` and is refined by a class refinement of feature module *BackOut*; this refinement allows students to back out from an exam. Hence, the contract of method `consistent()` is refined by changing the body of method `validStudent()`.

Pure methods in contracts support fine-grained contract refinement. Even parts of preconditions or postconditions can be refined, which would otherwise require to clone contracts and modify them. Such specification clones may lead to similar problems as code clones [14]. For example, when updating a contract, we may forget to update clones of this contract and introduce inconsistencies. Hence, specification clones should be avoided whenever possible requiring more sophisticated specification approaches such as pure-method refinement.

Pure-method refinement is expressive, because method refinements do neither depend on refined methods nor must relate to them in any way (e.g., weakening or strengthening existing contracts). A further advantage is that no new keywords and no linguistic concepts are needed for contract refinement, because

[4] `http://verifythis.cost-ic0701.org/post?pid=database-system-for-managing-exams`

Base

```
class ExamDataBase {
  /*@ ensures \result == (\forall int i; 0 <= i
    @     && i < students.length && validStudent(students[i]);
    @   students[i].points >= 0); @*/
  boolean consistent() {
    for(int i=0; i<students.length; i++)
      if (validStudent(students[i]) && students[i].points < 0)
        return false;
    return true;
  }
  /*@ pure @*/ boolean validStudent(Student student) {
    return student != null;
  }
}
class Student {
  //@ invariant matrNr > 0 && firstname != null && surname != null;
  int matrNr; String firstname, surname;
}
```

BackOut

```
refines class ExamDataBase {
  /*@ pure @*/ boolean validStudent(Student student) {
    return original(student) && !student.backedOut;
  }
}
refines class Student {
  //@ invariant !backedOut || backedOutDate != null;
  Date backedOutDate = null; boolean backedOut = false;
}
```

Fig. 4. *Pure-method refinement*: the contract of method `consistent()` contains a call to the pure method `validStudent()`. Feature *BackOut* refines the contract of `consistent()` indirectly by refining method `validStudent()`. By refining one pure method, we can refine several contracts indirectly at the same time.

traditional FOP mechanisms can be used. Hence, it is easy to understand the meaning of contracts, if the refinements of all pure methods therein are clear. The main disadvantage of pure-method refinement is that it strongly relies on the concept of pure methods being allowed to be called in contracts. Furthermore, the flexibility for refining methods by FOP may cause contracts which are hard to understand (e.g., if we have several refinements of the same method, some strengthening, some weakening, and some overriding).

4 Refinement of Invariants

DbC involves the specification of methods by contracts and classes by invariants usually expressing invariant properties of the fields. In the following, we assume that contract refinement is carried out with any of the previously discussed approaches and discuss how programmers can refine invariants analogously.

If invariants can be introduced in features, an invariant only needs to be established for the resulting program if the corresponding feature is selected (e.g., in Figure 4 feature *BackOut* introduces fields together with an invariant). Thus, we can build variable specifications using invariant introductions. Similarly to

contracts, invariants can be refined explicitly or implicitly (i.e., with or without a keyword referring to invariants defined in previously composed feature modules). When using *explicit invariant refinement*, we can use the keyword `original` to reference the previous definition of the invariant and combine it with the previous invariants. Applying *consecutive contract refinement* means that features can only add new invariants that need to hold as well. We can apply the concept of *pure-method refinement* to invariants. If an invariant contains a pure method call, the pure method can be refined using FOP method refinement. Finally, *contract overriding* can also be applied to invariants, where existing invariants can be overridden by features which we refer to as *invariant overriding*.

Allowing the refinement of invariants provides additional flexibility for the specification of feature-oriented programs. Every feature module can change invariants provided by previously composed feature modules. Depending on the approach chosen for refinement of contracts, we find it intuitive to refine invariants using the same means. However, the introduction and refinement of invariants allows that particular invariants only need to be fulfilled if a certain feature is present. As a result, it can be difficult to examine those combinations of features for which a certain invariant is present. The refinement of invariants has huge consequences as an invariant must hold for *all* methods of a class, and a change influences many callers and callees at the same time. Furthermore, the flexibility with invariant refinement can easily result in specifications that cannot be satisfied by any implementation. In Section 6, we evaluate whether the refinement of invariants is actually useful in practice.

5 Comparison

After presenting five alternative approaches of refining contracts, we now want to compare them based on properties directly related to specifications and give some intuition which approach is useful under which circumstances. We compare the approaches according to four properties which are different perspectives on the specification of programs: strictness, expressiveness, complexity, and specification clones. While strictness and expressiveness may indicate that an approach can not be applied to certain feature-oriented programs, the other criteria refer to properties that are nice to have.

Strictness can be used to classify all presented approaches regarding allowed and disallowed refinements from a logical point of view. Given a certain contract C, a refined contract C' may be strengthened with respect to method calls (e.g., by adding a further postcondition) or weakened (e.g., by requiring a further precondition). Strengthening means that every method fulfilling C' also fulfills C and weakening means that every method fulfilling C also fulfills C'. Further possibilities are to refine the contract with an equivalent one (e.g., by commuting preconditions or leaving the contract as-is) or to refine the contract in arbitrary way. In Figure 5, we illustrate the strictness relation by a Venn diagram. The intersection of weakened and strengthened contracts is the set of equivalent contracts. As plain contracting disallows any refinement of contracts,

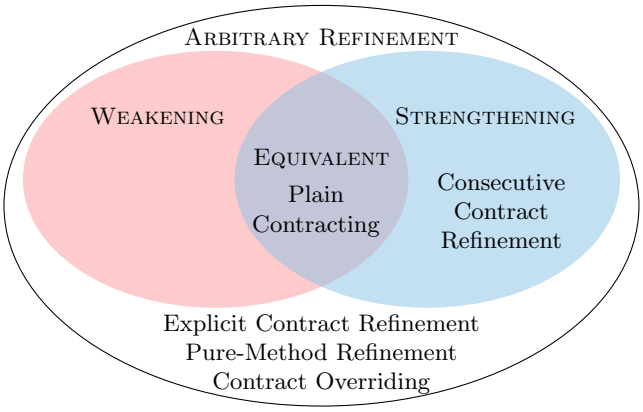

Fig. 5. Comparison of the presented approaches of contract refinement regarding strictness. Approaches may allow or disallow weakening and strengthening of contracts resulting in four categories. For example, disallowing both means to allow only contract refinements if they are equivalent to the original contract.

the contracts are equivalent for every method refinement. Consecutive contract refinement allows only to strengthen the original contract. All other presented approaches allow arbitrary refinements.

Expressiveness refers to whether we can specify all meaningful properties of feature-oriented programs. Given a particular program, we need to know whether we can express its specification with a certain approach or not. There is a connection to strictness: approaches allowing arbitrary refinements are more expressive than approaches allowing only strengthened contracts and similarly, strengthening is more expressive than equivalent contracts. In Table 1, we give an overview on the expressiveness of all presented approaches. The low expressiveness of plain contracting and consecutive contract refinement is simply based on their strictness. Contract overriding has a lower expressiveness compared to other approaches allowing arbitrary refinements, because derivative contracts may be needed if two features refine the same contract (see Section 3).

	Plain Contracting	Explicit Refinement	Consecutive Contract Ref.	Pure Method Refinement	Contract Overriding
Expressiveness	− −	+ +	0	+ +	+
Complexity	+ +	− −	+	0	− −
Specification Clones	+ +	0	+	+ +	− −

Table 1. Comparison of the presented approaches for the refinement of contracts. + + means that the approach is good with respect to the property (i.e., the approach has high expressiveness, contracts have a low complexity, specification clones can be avoided). Intuitively, − − refers to the worst and 0 to a neutral evaluation.

Complexity indicates whether it is easy for a programmer to manually retrieve the resulting contract of a certain method for a particular feature combination. An approach, in which determining the contract has the lowest complexity, is beneficial for programmers that need to create and maintain specifications because mistakes, such as wrong contracts or wrong implementations, can have expensive outcomes (e.g., it is expensive to detect errors using verification or testing). Thus, we expect contract specifications to have a low complexity. Contract overriding has the highest complexity, as contracts can be arbitrarily refined by each feature, and contracts can depend on the presence of every single feature. Contracts created by explicit contract refinement have a lower complexity since no derivative contracts are needed. Using pure-method refinement, contracts can only be refined at predefined positions. Consecutive contract refinement only allows a programmer to strengthen contracts meaning that if a certain feature is selected, then all methods need to establish the contracts defined therein, independent of other features. Clearly, the complexity is even lower if we do not allow refinements at all using plain contracting, because a contract is either not present or the same for all feature selections.

Specification clones are identical or very similar contracts. We expect that specification clones lead to similar problems as code clones (see Section 3). Hence, a specification approach should help to avoid specification clones. We consider contract overriding as the worst approach regarding specification clones, as it provides no ability to reuse contracts such that the only option is to copy and adapt contracts. A better approach is the explicit refinement of contracts and invariants because the keyword `original` can be used to reference preconditions and postconditions of a previous contract. With consecutive contract refinement, all contracts are implicitly reused such that we expect even less specification clones. The best approaches in terms of avoiding clones are plain contracting and pure-method refinement. Plain contracting completely disallows contract refinements, and with pure-method refinement even parts of contracts can be refined which allows to reuse existing contracts.

6 Evaluation

In order to evaluate the practicability of the five proposed specification approaches, we performed two case studies by creating feature-oriented programs including their specifications from scratch and three case studies by decomposing already specified object-oriented programs into feature modules. All our case studies are implemented and specified in feature-oriented extensions of Java and JML, but we expect similar results for other object-oriented languages and contract-based specification languages. The advantage of Java and JML is that many tools as well as specified and verified sample programs exist. However, it turned out that most existing examples are too small to be decomposed into features (i.e., only three of them were suitable for decomposition).

In Table 2, we present some statistics of our feature-oriented sample programs. They have between two and eight classes consisting of two to 42 fields

	ExamDB	Paycard	DiGraph	BankAccount	IntList
Classes, fields	4, 10	8, 42	8, 13	2, 7	2, 2
Methods (pure)	29 (8)	18 (5)	48 (22)	10 (0)	12 (0)
Features, variants	4, 8	4, 6	4, 8	6, 24	5, 16
Method refinements (pure)	2 (2)	3 (1)	0 (0)	4 (0)	4 (0)
Contracts (in core features)	25 (17)	10 (4)	43 (27)	8 (2)	7 (1)
Invariants (in core features)	5 (4)	6 (2)	12 (12)	4 (1)	3 (2)
Contract refinements	0	1	0	2	1
Contracts with method calls (refined, multiple)	8 (7, 4)	2 (2, 0)	29 (0, 10)	0 (0, 0)	0 (0, 0)
Invariant refinements	0	0	0	0	0
Invariants with method calls (refined, multiple)	0 (0, 0)	0 (0, 0)	5 (0, 0)	0 (0, 0)	0 (0, 0)

Table 2. Results of case studies

and ten to 48 methods. Some methods are declared as being pure. Our case studies have four to six features where six to 24 combinations of features are considered valid and can be used to generate different program variants. The programs are specified by seven to 43 contracts and three to twelve invariants.

With respect to *strictness and expressiveness* of the approaches, we found that four of five case studies could not be specified using plain contracting, because contract refinement was required. Only, the DiGraph case study could be specified with plain contracting; it does not contain a single method refinement as it is a library and the features chosen for decomposition do not cross-cut method implementations. But, method and contract refinement may be necessary when extracting further features or extending DiGraph with a new features. Contract strengthening is sufficient for three of five case studies. We specified the IntList and the Paycard case studies using consecutive contract refinement. Thus, for these case studies strengthening is sufficient. ExamDB and BankAccount rely on contract weakening. While contract strengthening is commonly used for OOP, it is not suited for any feature-oriented program. In larger programs, we expect even more examples where contract weakening is needed.

Our results show that some, but not all feature-oriented method refinements *require contract refinements*. For example, the BankAccount case study contains four method refinements, but for only two of them the contract was refined. Converse, pure-method refinement requires the refinement of methods per definition, but some method refinements may be introduced only to refine contracts (i.e., the method refinement is not needed for implementation of features but only to express their specification). For example, in the ExamDB case study, we newly introduced two refinements of pure methods to actually refine seven contracts each containing a call to the pure method.

The *granularity of contract refinement* can influence the suitability of the individual approaches. The case study ExamDB requires fine-grained refinement of contracts. In Figure 4, the contract of method `consistent()` is refined using pure-method refinement. The contract quantifies over all valid students, and feature *BackOut* can actually influence which students are valid (students that are backed-out are considered as invalid). In this example, only a small part of a contract needs to be refined, while most of it remains unchanged. Hence, we used pure-method refinement for ExamDB to express these fine-granular refinements. All other approaches would lead to specification clones. But, we also observed the danger that pure-method refinement is applied *accidentally*. When decomposing an existing system into features, the implementation may require the refinement of certain methods. If one of such methods is declared as pure, it may also be used in contracts. But then, we may accidentally refine contracts or invariants simply by refining these methods. If we choose to disallow pure-method refinement, we also need to make sure that either no pure method can be refined or that no method referenced in contracts or invariants can be refined. The same holds if we create a feature-oriented program from scratch.

In the case study Paycard, we used a *combination of two approaches*. We used pure-method refinement to refine two contracts, because the refinement was fine-grained. But, for another contract refinement, we used consecutive contract refinement as the whole original contract should be established as-is and refined by a further contract. The experience with our case studies showed that even combinations of presented approaches may be useful.

Not a single case study required the *refinement of invariants* (see Figure 2). Still, in all case studies except from DiGraph, invariants were introduced by several, optional features resulting in invariants that only hold for products of particular feature combinations. But, we found no case where a feature needed to refine the invariant defined by another feature. However, we had to split invariants into several smaller invariants when decomposing the ExamDB and Paycard case studies into features. Splitting was possible as the invariant actually was a conjunction, which can always be decomposed into several invariants. We cannot conclude that the refinement of invariants can generally be avoided, but at least *in our case studies* the introduction of invariants by features was sufficient. This is a positive result according to the strong disadvantages of invariant refinement discussed in Section 4.

In our case studies, we also analyzed whether a *global specification* that holds for all program variants is sufficient as suggested by Liu et al. [18]. Their example is that every pacemaker variant shall generate a pulse when no heartbeat is detected. In Table 2, we observe that only between 14 and 68 percent of all contracts and between 25 and 100 percent of invariants are given in core features. A *core* feature is a feature that is included in every program variant [8]. The core features together build-up the part that is common to all program variants. From the above figures, we can conclude that in none of our case studies a global specification is sufficient and specifications in form of contracts should be given for every feature as we propose in this paper.

In summary, our evaluation showed that contract refinement is needed when applying DbC to FOP. It is not always sufficient to only strengthen contracts (already in our small case studies) such that an approach for contract refinement should also allow weakening. From our qualitative and quantitative analysis, pure-method refinement is the most promising approach because contracts can be strengthened or weakened and fine-grained refinements are supported as well. Pure-method refinement may be combined with consecutive contract refinement to better support coarse-grained refinements. In our experience, invariant introductions should be used instead of invariant refinements whenever possible.

7 Related Work

In previous work, we considered formal verification of feature-oriented programs based on JML specifications. We proposed proof composition with the proof assistant Coq for efficient deductive verification of all program variants and applied a specification approach similar to contract overriding [23]. For the detection of feature interactions, we composed specifications with implicit contract refinement and analyzed program variants using ESC [22]. In each work, we proposed one specification approach and focused on verification issues. Our experience was that it is not clear what is the best way to specify feature-oriented programs using DbC. In this work, we propose three further specification approaches and compare all approaches regarding practicability by means of five case studies.

Specification using DbC has been considered for other program modularization techniques than FOP. Bruns et al. [9] and Hähnle et al. [12] discuss DbC for delta-oriented programming (DOP). DOP is an extension of FOP where feature modules (known as delta modules) can also remove methods, fields, and classes. A delta module can add or remove invariants and contracts. Since a feature module only refines existing methods, it is not reasonable to consider the removal of contracts or invariants for FOP.

DbC has been applied to aspect-oriented programming [24,19,1]. The aspect-oriented around advice corresponds roughly to feature-oriented method refinement and thus aspect-oriented programming can be seen as a superset of FOP [4]. Zhao and Rinard [24] proposed Pipa, a DbC specification language for AspectJ. AspectJ programs with Pipa annotations are translated into Java programs with JML annotations to allow programmers to reuse existing JML tools. Lorenz and Skotiniotis [19] analyze advice contracts in terms of runtime assertions. They propose three advice categories with an according runtime assertion strategy each: *agnostic* and *obedient* disallowing contract refinement (similar to contract overriding with equivalent contracts) and *rebellious* allowing contract strengthening (similar to contract overriding with compatible contracts). Agostinho et al. [1] discuss the interaction between classes and aspects while proposing agnostic pieces of advice. All these approaches force programmers to create specification clones, because they do not support contract weakening, which is needed in two of our case studies. Furthermore, the absence of aspects is not considered, while optional features in FOP are essential for software variability.

Most specification approaches for OOP assume behavioral subtyping [17] for subclasses which are the means to reuse code. Dhara and Leavens [11] propose specification inheritance to achieve behavioral subtyping which also is pursued in Eiffel [20] and JML [16]. With consecutive implicit refinement, we transfered the notion of behavioral subtyping to feature-oriented method refinement, but two of five case studies cannot be specified using this approach, as it is too restrictive.

8 Conclusion

In order to increase the reliability of feature-oriented programs, we discussed five approaches to integrate DbC with FOP and evaluated them by means of five case studies. We found that feature-oriented method refinement often requires the refinement of contracts such that the program specification depends on the actual selection of features. In contrast, the refinement of invariants can be avoided in our case studies. Furthermore, we identified the trade-off between expressiveness and complexity: while high expressiveness allows programmers to specify arbitrary feature-oriented programs, the complexity of contracts increases.

Acknowledgment

We thank Fabian Benduhn and anonymous reviewers for comments on earlier drafts of this paper. Apel's work is supported by the German Research Foundation (DFG – AP 206/2, AP 206/4, and LE 912/13). Saake's work is supported by the German Research Foundation (DFG – SA 465/34-1).

References

1. S. Agostinho, A. Moreira, and P. Guerreiro. Contracts for Aspect-Oriented Design. In *Proc. Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT)*. ACM, 2008.
2. S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *J. Object Technology (JOT)*, 8(5):49–84, 2009.
3. 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.
4. S. Apel, T. Leich, and G. Saake. Aspectual Feature Modules. *IEEE Trans. Software Engineering (TSE)*, 34(2):162–180, 2008.
5. S. Apel, W. Scholz, C. Lengauer, and C. Kästner. Detecting Dependences and Interactions in Feature-Oriented Design. In *Proc. Int'l Symposium Software Reliability Engineering (ISSRE)*, pages 161–170. IEEE, 2010.
6. S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Detection of Feature Interactions using Feature-Aware Verification. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 372–375. IEEE, 2011.
7. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Software Engineering (TSE)*, 30(6):355–371, 2004.

8. D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–708, 2010.
9. 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. Formal Verification of Object-Oriented Software (FoVeOOS)*, pages 61–75. Springer, 2010.
10. B. Delaware, W. Cook, and D. Batory. A Machine-Checked Model of Safe Composition. In *Proc. Workshop Foundations of Aspect-Oriented Languages (FOAL)*, pages 31–35. ACM, 2009.
11. K. K. Dhara and G. T. Leavens. Forcing Behavioral Subtyping through Specification Inheritance. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 258–267. IEEE, 1996.
12. R. Hähnle and I. Schaefer. A Liskov Principle for Delta-oriented Programming. In *Proc. Int'l Conf. Formal Verification of Object-Oriented Software (FoVeOOS)*, pages 190–207. Technical Report 2011-26, Department of Informatics, Karlsruhe Institute of Technology, 2011.
13. C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Comm. ACM*, 12(10):576–580, 1969.
14. E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do Code Clones Matter? In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 485–495. IEEE, 2009.
15. 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.
16. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *Software Engineering Notes (SEN)*, 31(3):1–38, 2006.
17. B. H. Liskov and J. M. Wing. A Behavioral Notion of Subtyping. *Trans. Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.
18. J. Liu, S. Basu, and R. Lutz. Compositional Model Checking of Software Product Lines using Variation Point Obligations. *Automated Software Engineering (ASE)*, 18(1):39–76, 2011.
19. D. H. Lorenz and T. Skotiniotis. Extending Design by Contract for Aspect-Oriented Programming. *Computing Research Repository (CoRR)*, abs/cs/0501070, 2005.
20. B. Meyer. Applying Design by Contract. *Computer*, 25(10):40–51, 1992.
21. C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 419–443. Springer, 1997.
22. W. Scholz, T. Thüm, S. Apel, and C. Lengauer. Automatic Detection of Feature Interactions using the Java Modeling Language: An Experience Report. In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD)*, pages 7:1–7:8. ACM, 2011.
23. T. Thüm, I. Schaefer, M. Kuhlemann, and S. Apel. Proof Composition for Deductive Verification of Software Product Lines. In *Proc. Int'l Workshop Variability-intensive Systems Testing, Validation and Verification (VAST)*, pages 270–277. IEEE, 2011.
24. J. Zhao and M. C. Rinard. Pipa: A Behavioral Interface Specification Language for AspectJ. In *Proc. Int'l Conf. Fundamental Approaches to Software Engineering (FASE)*, pages 150–165. Springer, 2003.