The Journal of Systems and Software 152 (2019) 83–107

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

Feature-oriented contract composition

Thomas Thüm[a], Alexander Knüppel[a,*], Stefan Krüger[b], Stefanie Bolle[a], Ina Schaefer[a]

[a] TU Braunschweig, Germany
[b] Paderborn University, Germany

ARTICLE INFO

Article history:
Received 18 January 2018
Revised 7 July 2018
Accepted 16 January 2019
Available online 17 January 2019

Keywords:
Feature-oriented programming
Software product lines
Design by contract
Deductive verification
Formal methods

ABSTRACT

A software product line comprises a set of products that share a common code base, but vary in specific characteristics called features. Ideally, features of a product line are developed in isolation and composed subsequently. Product lines are increasingly used for safety–critical software, for which quality assurance becomes indispensable. While the verification of product lines gained considerable interest in research over the last decade, the subject of how to specify product lines is only covered rudimentarily. A challenge to overcome is composition; similar to inheritance in object-oriented programming, features of a product line may refine other features along with their specifications. To investigate how refinement and composition of specifications can be established, we derive a notion of feature-oriented contracts comprising preconditions, postconditions, and framing conditions of a method. We discuss six mechanisms to perform contract composition between original and refining contracts. Moreover, we identify and discuss desired properties for contract composition and evaluate which properties are established by which mechanism. Our three main insights are that (a) contract refinement is seldom but crucial, (b) the Liskov principle does not apply to features, and (c) it is sufficient to accommodate techniques from object-orientation in the contract-composition mechanisms for handling frame refinements.

1. Introduction

Today, industrial software systems are rarely being developed monolithically, but exhibit an assembling of previously developed implementation artifacts. The goal of software product-line engineering is to systematically reuse these artifacts between a set of similar software products (Pohl et al., 2005b; Clements and Northrop, 2001; Czarnecki and Eisenecker, 2000). Despite higher costs in the beginning, the benefits are a reduction on the development time and lower development and maintenance costs in the long run (Pohl et al., 2005b). To acquire those benefits, implementation artifacts must be of high quality. Hence, quality-assurance techniques, such as code reviews, testing, and formal methods, have become critical for software product lines (Carmo Machado et al., 2014; Thüm et al., 2014).

Many quality-assurance techniques for product lines require a specification of the expected behavior of all products (Beohar et al., 2016). To verify a product-line, all the behaviour of all its individual products must adhere to their specifications. While there exist many approaches to specify product lines (Thüm et al., 2014; Benduhn et al., 2015; Beohar et al., 2016), their implementations often act rather as a proof of concept and are only rarely evaluated empirically.

In this work, we present a comprehensive discussion and empirical evaluation of *how to specify product lines* implemented by means of *feature-oriented programming* (Prehofer, 1997). Feature-oriented programming extends object-oriented programming, where features are implemented in isolation and subsequently composed to form software products. In addition to introducing new classes, methods, and fields, features may refine pieces of functionality of other features. We concentrate on feature-oriented programming for product-line specification, as it contains only core variability mechanisms which can be encoded in many other implementation techniques. For instance, feature-oriented method refinement can be expressed using the around advice in aspect-oriented programming (Apel et al., 2008), using method modifications in delta-oriented programming (Schaefer et al., 2010), and using presence conditions in preprocessor-based product lines (Kästner et al., 2009a). Consequently, when identifying variability patterns for feature-oriented method refinements, we can transfer these patterns to these other implementation techniques, whereas the opposite would not be possible in general. Hence, we are confident that our considerations are applicable to other product-line implementation techniques following

* Corresponding author.
E-mail addresses: t.thuem@tu-bs.de (T. Thüm), a.knueppel@tu-bs.de (A. Knüppel), stefan.krueger@uni-paderborn.de (S. Krüger), s.bolle@tu-bs.de (S. Bolle), i.schaefer@tu-bs.de (I. Schaefer).

https://doi.org/10.1016/j.jss.2019.01.044
0164-1212/

feature-oriented software development. In Section 9, we sketch how to transfer our results to these other domains.

For specifying features in feature-oriented programs, we focus on the *design-by-contract* paradigm (Meyer, 1988). Contracts decorate procedures with preconditions, postconditions, and framing conditions to assert the correctness of a change in state of a program. Preconditions can be assumed by a procedure and must be provided by callers, whereas postconditions are guaranteed and provided by the procedure to callers. Framing conditions may additionally limit the procedure's access to memory (Beckert et al., 2007; Chalin et al., 2005; Leavens and Müller, 2007). Reasons to base our investigation of product-line specification on design by contract are manifold. First, contracts enable the formal specification of behavior and, thus, can be used for a wide range of verification techniques, such as theorem proving (Burdy et al., 2005; Beckert et al., 2007; Barnett et al., 2011; Hatcliff et al., 2012), model checking (Robby et al., 2006), static analysis (Burdy et al., 2005; Hatcliff et al., 2012), runtime assertion checking (Meyer, 1988; Burdy et al., 2005; Barnett et al., 2011; Hatcliff et al., 2012), and test-case generation (Burdy et al., 2005; Hatcliff et al., 2012). Consequently, our findings inherently have many applications. Second, contracts help to identify the location of defects by means of blame assignment (Meyer, 1988; Hatcliff et al., 2012). For example, the violation of a postcondition is the fault of the method itself, whereas the violation of a precondition is the fault of a caller. We expect defect localization to be especially helpful when developing product lines with large development teams, in which no developer knows the complete code base. Third, design by contract is a means for specifying detailed designs. Once we understand the variability mechanisms required for specifications at code-level, we can use this knowledge to guide the development of product-line specification techniques for more abstract specifications (e.g., transition systems) or even abstraction mechanisms, such as model-based refinement methods (e.g., ASM (Börger and Stark, 2003) and Event-B (Abrial, 2010)).

While both design by contract and feature-oriented programming have been hot research topics for more than two decades, their combination was rarely been explored. A key question is how to define and compose contracts when applying feature-oriented method refinements. For instance, inheritance in object-oriented programs often assumes to follow the Liskov principle (i.e., behavioral subtyping) (Meyer, 1988; America, 1991; Liskov and Wing, 1994; Dhara and Leavens, 1996; Hatcliff et al., 2012). That is, objects of type T may be replaced by objects of type T' in a program without imposing side-effects if T' is subtype of T. Applied to design by contract that means when redefining a method in a derived class, a precondition may only be replaced by a weaker one, and a postcondition may only be replaced by a stronger one (Meyer, 1992). An important question is whether we can also assume the Liskov principle for features when considering contract composition, or whether it is too restrictive for feature-oriented method refinements.

With this work, our goal is to identify mechanisms to adequately *specify* proper program behavior for feature-oriented software product lines following the design-by-contract paradigm. This broadens our scope of acceptable specifications. The contracts in some product lines we investigate are for documentation purposes only and thus rather weak. Others, however, have been fully proven. An exhausting discussion on employing the specification to reason about behavioral correctness of the products (e.g., using deductive verification, model checking, or automated test case generation) is out of scope for this work, but has been done for some of the presented techniques already (Thüm et al., 2011; Scholz et al., 2011; Thüm et al., 2012, 2013, 2014).

In prior work (Thüm et al., 2012), we proposed contract-composition mechanisms to specify feature-oriented programs using contracts and discussed their advantages and disadvantages. As proof-of-concept and to enable larger evaluations, we developed tool support for contract composition in FeatureHouse and FeatureIDE. This work completes our prior work with two additional approaches, a comprehensive discussion on properties of contract composition, the incorporation of framing conditions, and a significantly extended evaluation. In summary, we make the following contributions.

- A presentation of six mechanisms to specify method contracts in feature-oriented programming including framing conditions.
- A discussion of interesting and desired properties for contract composition and an analysis which contract-composition mechanisms establish which properties.
- Tool support for specifying feature-oriented contracts and automated composition in FeatureIDE.
- An empirical evaluation including a total of 14 product lines, where five product lines have been developed from scratch, six product lines have been developed by decomposing existing systems with contracts, and three product lines based on feature-oriented programming have been specified subsequently.

2. Background

In this section, we provide the necessary background on design by contract, software product lines, and feature-oriented programming.

2.1. Design by contract

A formal specification makes the intended behavior of source code explicit and allows to reason about a program's correctness. This idea is implemented by numerous programming languages by means of *assertions* (Hatcliff et al., 2012). Assertions are side-effect free logical formulas to ensure a correct program state at specific locations during the development phase. If an assertion fails, diagnostic information about failure and program location are provided. *Assertions* can be used to represent preconditions and postconditions of methods in object-oriented programming, which is commonly referred to as *design by contract* (Meyer, 1988; Hatcliff et al., 2012). For instance, a precondition of a method may require some input parameters to be non-null. The rationale is, whenever the precondition ϕ is satisfied, the method m guarantees the postcondition ψ. Callers of that method can thus reason about their own correctness when they are also specified with a contract. With contracts, programmers can make their implicit intention regarding the input and output behavior of procedures explicit (Meyer, 1988). Explicitly annotating methods can improve program understanding and reuse (Helm et al., 1990). Moreover, defensive programming can be omitted, as contracts make the allowed values of input parameters explicit (Meyer, 1992). Hatcliff et al. (2012) elaborate on further applications, such as verifying the conformance of implementation and specification or the generation of test cases including test oracles based on contracts.

Besides preconditions and postconditions, there also exists the *frame* condition. In postconditions, the old-keyword evaluates the given expression prior to the method execution. To specify that the value of a given field f remains unchanged when executing a method, one could therefore add old(f) == f to the postcondition of that method. However, to avoid unnecessarily long postconditions, one may also specify a frame condition α of a method (Hatcliff et al., 2012), instead of adding old(f) == f to the postcondition for each field f of a class that must not be changed by the method. The frame condition, also called *assignable clause* (Beckert et al., 2007; Chalin et al., 2005; Leavens and Müller, 2007), indicates which parts of the program state the respective method is

```
class Account {
  public int balance;
  public final static int DAILY_LIMIT = 1000;
  public int withdraw; // resetted at midnight
  /*@ requires withdraw <= DAILY_LIMIT && x !=0;
    @ ensures ((\result && x < 0) ==>
    @          withdraw == \old(withdraw) - x)
    @ && (\old(withdraw) - x > DAILY_LIMIT ==> !\result)
    @ && (\result ==> balance == \old(balance) + x);
    @ assignable withdraw, balance;
    @*/
  boolean update(int x) {
  int newWithdraw = withdraw;
    if (x < 0) {
      newWithdraw -= x;
      if (newWithdraw > DAILY_LIMIT)
        return false;
    }
    withdraw = newWithdraw;
    balance = balance + x;
    return true;
  }
}
```

Listing 1. An account implementation with contracts in JML.

allowed to modify. Implicitly, callers then know which parts of the program state remain unaffected after a call. This information is often essential for information hiding (Leavens and Müller, 2007) and formal verification (Beckert et al., 2007; Weiß, 2011). Formally, we define that the triple comprising precondition ϕ, postcondition ψ, and frame condition α of a method m establishes a contract $c = \{\phi\}m\{\psi\}[\alpha]$.

Examples of popular languages with support for contracts are Eiffel (Meyer, 1988), the Java Modeling Language, (JML) (Leavens and Cheon, 2006), which is an extension of Java with support for contracts, and Spec# (Barnett et al., 2011). Here, all examples are illustrated in Java and JML, as Java is one of the most prominent object-oriented programming languages and many case studies facilitating JML already exist. In principle, the examples and case studies are not restricted to Java and JML, but can be transferred to other contracting languages.

In Listing 1, we give an example of a contracts as specified in JML. The class Account stores the balance and the daily withdraw and provides method update to withdraw or debit money. Method update is specified with a contract. The precondition is denoted by requires and states that parameter x has to be unequal to zero and that field withdraw must be below a daily limit. The postcondition is denoted by ensures and states that anyone calling method update can assume that field balance is modified by the value of parameter amount afterwards and that withdraw is set to the new daily withdraw. Keyword old refers in postconditions to the state prior to method execution. The assignable clause states that only field balance and withdraw are allowed to be modified by the method.

2.2. Software product lines

A software product line comprises a set of software products as variations of a common code base (Pohl et al., 2005a). The goal of product lines is to reuse reliable and tested software artifacts in related software products of the same domain and to offer cost-effective mass customization of software. Software products are characterized by their *features*. A feature is an end-user-visible functionality that exhibits commonalities and differences of products in a product line (Pohl et al., 2005a). For instance, payment methods of online shops vary and, thus, can be seen as features. Software product lines do not only reduce development and maintenance costs, but lead to more robust and reliable software. Especially in related embedded systems, where software often has a critical responsibility, yet differs in its functionality, developing artifacts in isolation and subsequent composition mechanisms are more cost-effective.

As a first step, the reusable artifacts of the product line are identified in a process often being referred to as *domain engineering* (Pohl et al., 2005a). During *application engineering* (Pohl et al., 2005a), specific products are developed by reusing the identified artefacts. In this process, features are selected and composed to generate a software product.

2.3. Feature-oriented programming

A popular language-based implementation technique for product lines is *feature-oriented programming* that extends object-oriented programming (Prehofer, 1997; Batory et al., 2004). Classes are split up and distributed over feature modules to facilitate better reuse. Feature modules are ideally developed in isolation and introduce new classes, methods, and fields. In particular, feature modularity has many benefits over classical object-oriented programming. For instance, a simple mapping between features and their implementations eases fault localization. Once a set of features is selected, the corresponding implementation modules are composed automatically (Apel et al., 2013a).

Additionally, methods can be refined using the keyword original, similar to the keyword super when extending classes in object-oriented programs. In particular, methods are decomposed into parts to achieve separation of concerns (Harrison and Ossher, 1993; Kiczales et al., 1997; Tarr et al., 1999) or to enable the automatic composition based on requirements (Prehofer, 1997; Batory et al., 2004; Czarnecki and Eisenecker, 2000; Apel et al., 2013a). Depending on the selection and order of features that are composed, the resulting inheritance hierarchy may vary.

For brevity, we use the terms feature module and feature interchangeably. Subsequently, a set of features $F = \{f_1, \ldots, f_n\}$ is incrementally merged together to a particular software product p by a composition mechanism $\bullet$: $F \times F \to F$ (Apel and Hutchins, 2010):

$$p = f_n \bullet (f_{n-1} \bullet (\cdots \bullet (f_2 \bullet f_1)))$$

There exists a diverse set of tools for the incremental composition of features (e.g., FeatureHouse (Apel et al., 2013b) or AHEAD (Apel and Hutchins, 2010)). In FeatureHouse, software artifacts are represented as trees and can refine each other by merging their corresponding sub-trees respecting specific composition conditions. For example, packages, classes, fields, and methods form the hierarchical structure of object-oriented code. If two features modify the same class (e.g. by adding a new method each), their sub-trees overlap and the result is a new tree inheriting modifications of both features. In Listing 2, we exemplify the composition of two features a product line with the name *BankAccount*. Feature *DailyLimit* refines the update-method of feature *Base* by using the keyword original, which is replaced by the original content of the update-method in the resulting class.

3. Problem statement

Specifying feature-oriented software product lines is challenging. In feature-oriented programs, implementation artifacts, such as methods, are distributed over the set of feature modules and subsequently composed together when the respective features are selected. Similar to this idea, contracts could be modularized, too,

and composed subsequently together with their respective methods. With *contract composition*, we refer to the process of retrieving a contract for a method given a list of contracts as input. Contract composition is motivated by the decomposition of methods on code level. For both applications, the question arises how to decompose and compose contracts accordingly. That is, for each possible composition of methods, we are interested in the behavior in terms of a contract. While one may desire to compose an arbitrary number of contracts, our consideration is based on the composition of two contracts. This is sufficient, because a composition of more than two contracts can be simulated by several binary compositions. This simplification is in line with work on software composition, which is also often defined as a binary function (Meyer, 1988; Prehofer, 1997; Liskov and Wing, 1994; Dhara and Leavens, 1996; Batory et al., 2004; Höfner and Möller, 2009; Apel et al., 2010; Hatcliff et al., 2012).

When features refine methods, an important question is whether refinement of their contracts is inevitable or not. However, unlike method composition where only the order of features is relevant, it seems that contract composition has to be handled differently according to certain scenarios. To better understand the problem of refining contracts, consider the following example based on the aforementioned product line *BankAccount*.

Example 1. In Listing 3, we show another excerpt of the *BankAccount* product line comprising the features *Base* and *DailyLimit*. The product line *BankAccount* contains a method refinement of the specified method update. In feature module *Base*, the method adds value x to the current balance. In feature module *DailyLimit*, the method disallows withdrawals over a fixed daily limit. The resulting composition of both features is illustrated in Listing 2. The composed contract, depicted in Listing 1, shows the intended behavior of method update when both features are selected. In this specific case, the definition of the refining contract in feature module *DailyLimit* is currently unknown and needs to be identified. A solution for composing contracts can be simple. For instance, we may define that the refining contract simply overwrites the original one. In the general case, however, the composition of contracts is not obvious. We may add a new feature that refines method update and has a contradicting precondition (e.g., allows x to be 0). Hence, contract composition must be handled differently in such a case. The two most important challenges are therefore how to specify method update in feature *DailyLimit* and how to compose contracts for method refinement.

4. Contract-composition mechanisms

In this section, we introduce mechanisms to integrate contracts into feature-oriented programming as a particular technique for software composition. In particular, we propose six mechanisms for contract composition in feature-oriented programming, namely plain contracting, contract overriding, explicit contract refinement, conjunctive contract refinement, cumulative contract refinement, and consecutive contract refinement. These contract-composition mechanisms extend the composition of feature modules by support for contracts.

Given an *original contract* $c = \{\phi\}m\{\psi\}[\alpha]$ and a *refining contract* $c' = \{\phi'\}m'\{\psi'\}[\alpha']$, we denote the composed contract as $c'' = c' \bullet c = \{\phi'\}m'\{\psi'\}[\alpha'] \bullet \{\phi\}m\{\psi\}[\alpha] = \{\phi''\}m' \bullet m\{\psi''\}[\alpha'']$. A specific mechanism for contract composition defines how ϕ'', ψ'', and α'' are derived from the contracts c and c'. We consider a *contract-composition mechanism* M as a function $\bullet_M$: $C \times C \to C$ defined over the set C of all possible contracts. If it is clear from the context which contract-composition mechanism is meant, we write $\bullet$ instead of $\bullet_M$ and thus overload the composition operator for method implementations (cf. Section 2.3). Clearly, any given set

Base
```
class Account {
  public int balance;
  boolean update(int x) {
    balance = balance + x;
    return true;
  }
}
```

DailyLimit
```
class Account {
  public final static int DAILY_LIMIT = 1000;
  public int withdraw;
  boolean update(int x) {
    int newWithdraw = withdraw;
    if (x < 0) {
      newWithdraw −= x;
      if (newWithdraw > DAILY_LIMIT)
        return false;
    }
    withdraw = newWithdraw;
    original(x)
  }
}
```

DailyLimit • Base
```
class Account {
  public final static int DAILY_LIMIT = 1000;
  public int balance;
  public int withdraw;
  boolean update(int x) {
    int newWithdraw = withdraw;
    if (x < 0) {
      newWithdraw −= x;
      if (newWithdraw > DAILY_LIMIT)
        return false;
    }
    withdraw = newWithdraw;
    balance = balance + x;
    return true;
  }
}
```

Listing 2. *Feature composition* in product line *BankAccount* of features *Base* and *DailyLimit*.

Base
```
class Account {
  public int balance;
  /*@ requires x != 0;
    @ ensures \result ==> balance == \old(balance) + x ;
    @ assignable balance;
    @*/
  boolean update(int x) {...}
}
```

DailyLimit
```
class Account {
  public final static int DAILY_LIMIT = 1000;
  public int withdraw;
  /*@ requires ?
    @ ensures ?
    @ assignable ?
    @*/
  boolean update(int x) {...}
}
```

Listing 3. Decomposition of contracts in product line *BankAccount*: feature *Base* introduces a contract for method update and in feature *DailyLimit*, method update is refined. The question is how one formulates the contract of the refined method update to produce the result depicted in Listing 2?

Base

```
public class IntegerList {
  //@ invariant data != null;
  public int[] data;
  public IntegerList() { data = new int[0]; }
  /*@ assignable data;
    @ ensures (\exists int z; 0<=z && z<data.length
    @   && data[z]==newTop)
    @  && (\forall int k; 0<=k && k<\old(data).length ==>
    @  (\exists int z; 0<=z && z<data.length
    @ && data[z]==\old(data[k]))); @*/
  public void push(int newTop) {
    int[] tmp = new int[data.length+1];
    tmp[tmp.length−1] = newTop;
    for (int i = 0; i < data.length; i++) { tmp[i] = data[i]; }
    data = tmp;
  }
  [...]
}
```

Sorted

```
public class IntegerList {
  //@ invariant (\forall int k; 0<=k && k<data.length−1;
  //@ data[k]>=data[k+1]);
  public void push(int newTop) { original(newTop); sort(); }
  /*@ assignable data;
    @ ensures (\forall int k; 0<=k && k<data.length−1;
    @ data[k]>=data[k+1]);
    @ ensures (\exists int z; 0<=z && z<data.length
    @ && data[z]==newTop)
    @  && (\forall int k; 0<=k && k<\old(data).length ==>
    @  (\exists int z; 0<=z && z<data.length
    @ && data[z]==\old(data[k]))); @*/
  private /*@ helper @*/ void sort() {
    for (int i = 0; i < data.length; i++)
      for (int j = data.length−2; j >= 0; j−−)
        if (data[j] < data[j+1]) {
          int tmp = data[j]; data[j] = data[j+1]; data[j+1] = tmp;
        }
  }
}
```

Listing 4. *Plain contracting* in product line *IntegerList*: feature *Base* introduces a contract for method push, which is not refined in feature *Sorted*.

C is specific to a certain specification language, contract composition independently of a particular language. Still, we require that each contract $c \in C$ can be formulated as $c = \{\phi\}m\{\psi\}[\alpha]$, whereas we discuss further language constructs in Section 6.

All mechanisms are illustrated based on Java and JML, but are not restricted to them in principle. All examples used for illustration are excerpts of the product lines used in our evaluation.

4.1. Plain contracting

A simple mechanism to deal with contracts during feature-module composition is to apply the identity function to the original contract. That is, given an original contract c and a refining contract c', the result of contract composition is always the original contract c. We call this mechanism *plain contracting* and denote it as $\bullet_{pc}$ (i.e., $c' \bullet_{pc} c = c$). In plain contracting, the idea is to define a contract for each method, but to forbid their refinement. Instead of ignoring refining contracts during composition, in practice, no refinements are written. Nevertheless, feature modules may contain method refinements if they establish the original contract.

Example 2. In Listing 4, we show an excerpt of a product line that was developed by Wolfgang Scholz for feature-interaction detection (Scholz et al., 2011). The product line *IntegerList* contains a method refinement of method push. In feature module *Base*, the method is introduced with a contract and simply inserts a given element at the end of the list. In feature module *Sorted*, the method is refined such that the list is sorted after each insertion. The method refinement adheres to the contract defined in feature module *Base* and is not refined in the optional feature module *Sorted*.

A rather technical design decision of contract composition, in general, and plain contracting, in particular, is how to handle methods without a contract during composition. In design by contract, the absence of a precondition means that there are no assumptions that the caller has to fulfill (Meyer, 1988), which is semantically equivalent to requires true. Analogously, the absence of a postcondition indicates that there is nothing the caller can rely on (i.e., ensures true and assignable \everything). In the following, we call such a missing contract an *empty contract*, denoted as $c = \epsilon$. Given this semantics, one has decide how deal with such contracts during composition. For instance, assume we want to compose the method m with methods m' and m'', of which m has no contract and the contracts of m' and m'' are c' and c'', respectively. If we ignore empty contracts during composition, the result for our example is $c'' \bullet_{pc} c' = c'$. However, if we treat empty contracts as contracts that are subject to composition, the result is $c'' \bullet_{pc} c' \bullet_{pc} c = \epsilon$. While both options are possible, the discussion is largely orthogonal to our work. We choose to ignore empty contracts during composition in the following, because a programmer can enforce the latter behaviour by providing the trivial contract requires true, ensures true, and assignable \everything.

4.2. Contract overriding

Contract overriding is a contract-composition mechanism that is complementary to plain contracting. In contract overriding, all methods and method refinements may provide a contract. During composition, the refining contract completely overrides the original contract (i.e., $c' \bullet_{co} c = c'$). Similar as for plain contracting, we assume that empty contracts are ignored during composition. Otherwise, we would need to repeat contracts for each method refinement, even if they do not require any changes to original contracts.

Example 3. In Listing 5, we give an example for contract overriding. The product line *GPL-scratch* was developed by André Weigelt to illustrate the need of different contract composition techniques in a single product line (Weigelt, 2013). Feature module *Base* introduces a method addEdge that takes a given edge and inserts it into an existing graph. However, the edge needs to be non-null and the nodes it connects must already exist in the graph. The feature module *MaxEdges* refines the method implementation such that only a specified number of edges can be added to the graph. That is, an edge is only inserted if the maximum number of edges will not be exceeded. Consequently, the contract given in feature module *MaxEdges* is supposed to completely override the original contract.

4.3. Explicit contract refinement

Similar to contract overriding, *explicit contract refinement* permits contract refinements. That is, refining contracts override original contracts. However, refining contracts may refer to the original precondition and original postcondition in their precondition and postcondition, respectively. Similar to feature-oriented method refinement, we introduce keyword original in preconditions, postconditions, and assignable clauses, which are replaced by the

```
public class Graph {                                          Base
  private Collection<Node> nodes;
  private Collection<Edge> edges;
  /*@ requires edge != null && nodes.contains(edge.first)
    @   && nodes.contains(edge.second);
    @ ensures hasEdge(edge); @*/
  public void addEdge(Edge edge) {
    edges.add(edge);
  }
  [...]
}
```

```
public class Graph {                                          MaxEdges
    //@ invariant MAXEDGES != null;
  private final static int MAXEDGES = new int[10];
  /*@ requires edge != null && nodes.contains(edge.first)
    @   && nodes.contains(edge.second);
    @ ensures \old(edges.size()) < MAXEDGES ==>
    @ hasEdge(edge); @*/
  public void addEdge(Edge edge) {
    if(countEdges() < MAXEDGES)
      original(edge);
  }
}
```

Listing 5. *Contract overriding* in product line *GPL-scratch*: feature *MaxEdges* overrides the contract of feature *Base* (adapted from Weigelt, 2013).

```
public class Graph {                                          Base
  private Collection<Node> nodes;
  private Collection<Edge> edges;
  /*@ requires edge != null && nodes.contains(edge.first)
    @   && nodes.contains(edge.second);
    @ ensures hasEdge(edge); @*/
  public void addEdge(Edge edge) {
    edges.add(edge);
  }
  [...]
}
```

```
public class Graph {                                          MaxEdges
    //@ invariant MAXEDGES != null;
  private static int MAXEDGES = new int(10);
  /*@ ensures \old(edges.size()) < MAXEDGES ==>
    @ \original; @*/
  public void addEdge(Edge edge) {
    if(countEdges() < MAXEDGES)
      original(edge);
  }
}
```

Listing 6. *Explicit contract refinement* in product line *GPL-scratch*: feature *MaxEdges* refines a contract by referring to the original contract defined in feature *Base* (adapted from Weigelt, 2013).

original conditions during composition. We define explicit contract refinement based on the composition of predicates:

$$\{\phi'\}m'\{\psi'\}[\alpha'] \bullet_{ecr} \{\phi\}m\{\psi\}[\alpha] = \{\phi' \bullet \phi\}m' \bullet m\{\psi' \bullet \psi\}[\alpha' \bullet \alpha],$$

whereas $\phi' \bullet \phi$ is the result of replacing all occurrences of keyword original by ϕ in ϕ' ($\psi' \bullet \psi$ and $\alpha' \bullet \alpha$ are defined analogously). The keyword is neither mandatory in preconditions, postconditions, nor the frame and may even appear several times in the same precondition or postcondition, respectively. In fact, contract overriding is a special case of explicit contract refinement where the keyword original is never used.

Example 4. In Listing 6, we give an example for explicit contract refinement based on the previous example in Listing 5. The feature module *Base* is identical. However, instead of cloning precondition and postcondition of the original contract in feature module *MaxEdges*, we refer to them by means of keyword original. The result of composing both feature modules is exactly the same as for our example on contract overriding.

4.4. Conjunctive contract refinement

In contrast to explicit contract refinement, the next three contract-composition mechanisms that we discuss do not require any keywords to explicitly refer to previous contracts. We refer to these mechanisms as *implicit contract refinement*, as the composition of two contracts is not explicitly observable in the contracts, but only implicitly defined by the respective mechanism. One of these mechanisms for implicit contract refinement is *conjunctive contract refinement*. Given two contracts $c = \{\phi\}m\{\psi\}[\alpha]$ and $c' = \{\phi'\}m'\{\psi'\}[\alpha']$ their composition is determined by the conjunction of their preconditions and postconditions, respectively. All fields ignored in the frame can be encoded in the postcondition (i.e., $\psi \models \bigwedge_{f \notin \alpha} (\backslash\textbf{old}(f) == f)$). As the postconditions are conjoined, the resulting frame is semantically equal to the intersection of both frames. The reason is that the complement of frame α is expressed as the conjunction of $\backslash\textbf{old}(f) == f$ in the postcondition for all $f \notin \alpha$. We denote this intersection as the *frame cut*. To summarize, we define conjunctive contract refinement as

$$c' \bullet_{ConjCR} c = \{\phi' \wedge \phi\}m' \bullet m\{\psi' \wedge \psi\}[\alpha' \cap \alpha].$$

Example 5. In Listing 7, we again show an excerpt of product line *GPL-scratch* to exemplify conjunctive contract refinement. The keyword conjunctive_contract indicates that the contracts for method equals are composed using conjunctive contract refinement. A detailed discussion on this and other contract-composition keywords are postponed to Section 7.1. Each contract refinement contains a precondition and a postcondition that must be fulfilled in addition to all preconditions and postconditions defined in other feature modules. In particular, the optional feature module *Weighted* introduces a new field weight, for which method equals and its contract need to be refined accordingly, because edges are only considered equivalent if they have the same weight. A similar refinement is given in feature module *Directed*, which is considered alternative to feature module *Undirected* (not shown for brevity).

4.5. Cumulative contract refinement

There is a further mechanism for implicit contract refinement, to which we refer to as *cumulative contract refinement*. Compared to conjunctive contract refinement, the idea is to facilitate modular reasoning for callers similar to subcontracting in object orientation. Meyer (1988) states that composed preconditions must be weaker or equal to original preconditions and composed postconditions must be stronger or equal. He has proposed a simple language rule that avoids checking the conformance using theorem proving: preconditions are combined in a disjunction and postconditions in a conjunction. The framing adheres to the same thought process as for conjunctive contract refinement, because postconditions are composed by conjunction, too. Adopting this language rule to feature orientation leads us to the definition of cumulative contract refinement as $\{\phi'\}m'\{\psi'\}[\alpha'] \bullet_{CumCR} \{\phi\}m\{\psi\}[\alpha] = \{\phi' \vee \phi\}m' \bullet m\{\psi' \wedge \psi\}[\alpha' \cap \alpha]$. We omit an example, as the only difference between cumulative contract refinement and conjunctive contract refinement is the disjunction of preconditions.

```
public class Edge implements Comparable<Edge> {                    Base
  private Node first, second;
  /*@ \conjunctive_contract
    @ requires ob != null;
    @ ensures \result ==> ob instanceof Edge; @*/
  @Override
  public /*@ pure @*/ boolean equals(Object ob) {
    return (ob instanceof Edge) ? true : false;
  }
  [...]
}
```

```
public class Edge {                                                Weighted
  private Integer weight = 0;
  /*@ requires weight != null;
    @ ensures \result ==> weight == ((Edge)ob).weight; @*/
  public /*@ pure @*/ boolean equals(Object ob) {
    return original(ob) && weight.equals(((Edge)ob).weight);
  }
  [...]
}
```

```
public class Edge {                                                Directed
  /*@ requires first != null && second != null;
    @ ensures \result ==> first.equals(((Edge) ob).first) &&
    @   second.equals(((Edge) ob).second); @*/
  public /*@ pure @*/ boolean equals(Object ob) {
    return original(ob) && first.equals(((Edge) ob).first) &&
      second.equals(((Edge) ob).second);
  }
  [...]
}
```

Listing 7. *Conjunctive contract refinement* in product line *GPL-scratch*: features *Weighted* and *Directed* refine a contract by adding a precondition and a postcondition to the original contract defined in feature *Base* (adapted from Weigelt, 2013).

4.6. Consecutive contract refinement

In the third mechanism for implicit contract refinement, to which we refer to as *consecutive contract refinement*, we apply specification inheritance (Dhara and Leavens, 1996) known from object orientation to product lines and therefore we support behavioral subtyping (America, 1991; Dhara and Leavens, 1996) with this mechanism. Specification inheritance is an enhancement compared to subcontracting (Dhara and Leavens, 1996), and we aim to transfer this enhancement to product lines by the following definition. We define consecutive contract refinement as

$$\begin{aligned}&\{\phi'\}m'\{\psi'\}[\alpha'] \bullet_{ConsCR} \{\phi\}m\{\psi\}[\alpha]\\ &= \{\phi' \vee \phi\}m' \bullet m\{(old(\phi') \Rightarrow \psi') \wedge (old(\phi) \Rightarrow \psi)\\ &\wedge(\neg old(\phi) \Rightarrow \forall_{x\in\alpha\setminus\alpha'} old(x) = x)\wedge\\ &(\neg old(\phi') \Rightarrow \forall_{x'\in\alpha'\setminus\alpha} old(x') = x')\}[\alpha \cup \alpha']\end{aligned}$$

in which x, x' represent assignable locations. We extended the frame cut to also depend on the preconditions. To this end, we unify both frames and delegate the preserving of respective locations to the postcondition. The rationale is that the precondition depends on the input and must be evaluated to establish the frame.

Example 6. We give an example for consecutive contract refinement based on product line *GPL-scratch* in Listing 8. The original method sortEdges takes a list of edges as input and returns a sorted list. The feature module *UniqueEdges* extends the method by additionally supporting a set of edges as input, which cannot contain duplicate values. Thus, when calling method sortEdges with

```
public class Graph {                                               Base
  /*@ \consecutive_contract
    @ requires edges instanceof List<Edge>;
    @ ensures (\forall int i; 0 < i && i < \result.size();
    @   \result.toArray()[i−1].compareTo(
    @   \result.toArray()[i]) <= 0); @*/
  public Collection<Edge> sortEdges(
                        Collection<Edge> edges) {
    List<Edge> list = new ArrayList<Edge>(edges);
    Collections.sort(list);
    return list;
  }
  [...]
}
```

```
public class Graph {                                               UniqueEdges
  /*@ requires edges instanceof Set<Edge>;
    @ ensures (\forall int i; 0 < i && i < \result.size();
    @   \result.toArray()[i−1].compareTo(
    @   \result.toArray()[i]) < 0 ); @*/
  public Collection<Edge> sortEdges(
                        Collection<Edge> edges) {
    if (!(edges instanceof Set<Edge>))
      return original(edges);
    return new TreeSet<Edge>(edges);
  }
  [...]
}
```

Listing 8. *Consecutive contract refinement* in product line *GPL-scratch*: features *UniqueEdges* refines a contract by adding a new pair of precondition and postcondition to the original contract defined in feature *Base* (adapted from Weigelt, 2013; differences of contracts highlighted for convenience).

a set as input, the result is strictly sorted (i.e., is sorted and does not contain duplicates). Given that feature *UniqueEdges* is selected, the caller has the choice which precondition to fulfil (i.e., passing a list or a set) and can rely on the respective postcondition. This example could not have been specified by means of cumulative contract refinement, because it is impossible for the method to fulfil both postconditions if only the precondition of feature *Base* is established.

4.7. Summary

The six mechanisms we introduced can be distinguished by the supported contract refinements. Plain contracting completely forbids any contract refinement, whereas conjunctive and cumulative contract refinement enable the refinement in a limited way. Consecutive contract refinement subsumes cumulative contract refinement with respect to the supported refinements. Finally, contract overriding and explicit contract refinement facilitate arbitrary contract refinements and, consequently, also arbitrary method refinements. In the next section, we introduce desired properties of contract composition and discuss advantages and disadvantages of all six mechanisms. In the remainder of this work we investigate whether all mechanisms are required in particular scenarios or whether some of the mechanisms may completely replace others. The latter would certainly reduce the complexity of specifying features from a user's perspective.

5. A taxonomy for contract composition

In the previous section, we developed formal definitions for six mechanisms. In this section, we explore fundamental options for

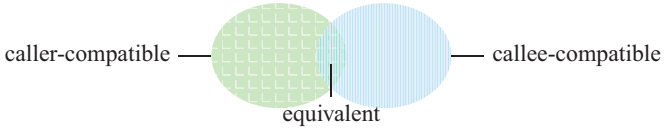

Fig. 1. Compatibility of changed contracts for callers and callees.

contract composition that are desirable and discuss all mechanisms in more detail with respect to their advantages and disadvantages. Whenever two methods are composed (cf. Section 3), the question arises which composition mechanism should be preferred over others and which properties are established by the composition. First, we discuss interesting properties of contract-composition mechanisms in Section 5.1. Second, we discuss four fundamental options for contract composition based on these properties in Section 5.2. Third, we provide a comparison of all six mechanisms in Section 5.3.

5.1. Properties of contract composition

A contract defines obligations and benefits for callers and the callee (i.e., the method itself), respectively (Meyer, 1988). We distinguish between two views, namely the *caller view* and the *callee view*. The caller has the obligation to fulfil the precondition of the method, but can rely on the postcondition and on the framing. The callee can rely on the precondition, but has to fulfil the postcondition and respect the framing. As the result of contract composition is a new contract for a particular method, a distinguishing property of contract composition is to which extent the new contract is compatible with the original and the refining contract. We define compatibility with respect to the callee view and the caller view:

Definition 7. Given two contracts $c_1 = \{\phi_1\}m_1\{\psi_1\}[\alpha_1]$ and $c_2 = \{\phi_2\}m_2\{\psi_2\}[\alpha_1]$.

- The contract c_2 is called *caller-compatible with respect to* c_1, if and only if $\phi_1 \vDash \phi_2$ and $\psi_2 \vDash \psi_1$ and $\alpha_2 \subseteq \alpha_1$, and *caller-incompatible* otherwise.
- The contract c_2 is called *callee-compatible with respect to* c_1, if and only if $\phi_2 \vDash \phi_1$ and $\psi_1 \vDash \psi_2$ and $\alpha_1 \subseteq \alpha_2$, and *callee-incompatible* otherwise.
- If and only if contract c_2 is both, callee-compatible and caller-compatible with respect to c_1, then c_2 is called *equivalent* to c_1.

We illustrate these definitions by means of a Venn diagram in Fig. 1. Assuming a fixed contract c_1, the Venn diagram illustrates the compatibility of all possible contracts c_2 with respect to c_1. Considering the caller view, it seems beneficial if the result of contract composition c_2 is caller-compatible, because all callers relying on c_1 can rely on c_2 instead. In contrast, callee-compatibility is desirable, because callees do not need to be aware of contract changes. As features are often optional and not part of the final software product, callee-compatibility removes the necessity to check whether a callee adheres to the specification during feature composition. However, requiring both properties enables only changes to contracts in which preconditions, postconditions, and frames remain equivalent (i.e., $\vDash \phi_1 \Leftrightarrow \phi_2$, $\vDash \psi_1 \Leftrightarrow \psi_2$, and $\alpha_1 = \alpha_2$).

Based on caller-compatibility and callee-compatibility, we define preservation properties for contract-composition mechanisms that indicate to which extent it maintains the compatibility with the original and refining contract. Compared to our previous definitions of compatibility, we quantify over all possible contracts as input and classify the mechanism rather than single contracts. We define the following four *preservation properties* (cf. Table 1) that facilitate some form of modular reasoning:

Table 1
Compatibility of contracts for different preservation properties.

Preservation property	Compatibility for	Compatibility to
Original-caller-preserving	Callers	Original contract
Refinement-caller-preserving	Callers	Refining contract
Original-callee-preserving	Callees	Original contract
Refinement-callee-preserving	Callees	Refining contract

Definition 8. Assume that a contract-composition mechanism m composes an original contract c with a refining contract c' to the resulting contract $c'' = c' \bullet_m c$ (cf. Section 4). The mechanism m is called *original-caller-preserving*, if the resulting contract c'' is caller-compatible with respect to c, and *refinement-caller-preserving*, if the resulting contract c'' is caller-compatible with respect to c' for all $c, c' \in C$. The mechanism m is called *original-callee-preserving*, if the resulting contract c'' is callee-compatible with respect to c, and *refinement-callee-preserving*, if the resulting contract c'' is callee-compatible with respect to c' for all $c, c' \in C$.

In object-oriented programming, original-caller-preserving contract composition for inheritance is already known as behavioral subtyping (Liskov and Wing, 1994; Dhara and Leavens, 1996; Hatcliff et al., 2012). Nevertheless, we introduce a new name for it as composition mechanisms are orthogonal to object-oriented inheritance for product lines. For example, aspect-oriented programming, delta-oriented programming, and feature-oriented programming can be seen as extensions of object-oriented programming and do not aim to completely replace inheritance (Kiczales et al., 1997; Prehofer, 1997; Schaefer et al., 2010).

Given the four preservation properties of contract composition, it seems useful to prefer mechanisms that fulfil as many as possible of these properties, because those mechanisms enable compositional reasoning. For instance, an original-caller-preserving mechanism allows verification tools and programmers to reason about a method call without knowing later contract refinements. That is, if we formally verify in a code review that a given method m conforms to a contract c of method m' and method m calls method m', this fact cannot be invalidated by refining the contract of method m' in a later refinement. This is because the contract c may only be changed by refining contracts c' in a way that the resulting contract c'' is caller-compatible to c. Similarly, refinement-caller-preserving mechanisms do not depend on changes to previous contracts. Analogously, the callee can be verified independently of earlier or later contract refinements, if the mechanism is original-callee-preserving or refinement-callee-preserving, respectively.

5.2. Fundamental options for contract composition

Not all four preservation properties of contract composition are compatible with each other.

Theorem 9. *There exists no contract-composition mechanism that is both, original-caller-preserving and refinement-callee-preserving.*

Proof. We prove the theorem with proof by contradiction. Assume there is a contract-composition mechanism m that is both, original-caller-preserving and refinement-callee-preserving. Because mechanism m is original-caller-preserving, for all original contracts $c \in C$ and refining contracts $c' \in C$ the resulting contract c'' is caller-compatible with respect to c. Without loss of generality, we assume that $c = \{\phi\}m\{\psi\}[\alpha]$, $c' = \{\phi'\}m'\{\psi'\}[\alpha']$, and $c'' = \{\phi''\}m''\{\psi''\}[\alpha'']$. Because contract c'' is caller-compatible with respect to c, we know that $\phi'' \vDash \phi$ and $\psi \vDash \psi''$. Because mechanism m is refinement-callee-preserving, we know that c'' is callee-compatible with c', and thus $\phi' \vDash \phi''$ and $\psi \vDash \psi''$. However, with $\phi' \vDash \phi''$ and $\phi'' \vDash \phi$ it follows that $\phi' \vDash \phi$, which is a restriction on

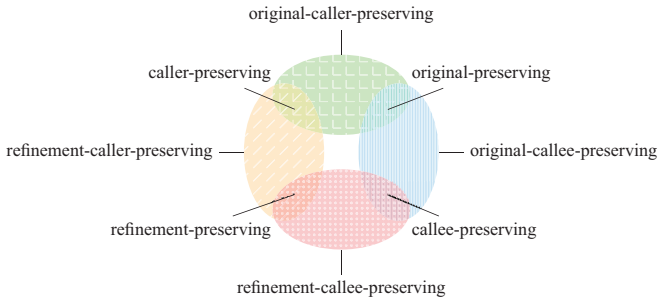

Fig. 2. Contract-preservation properties indicate compatibility for callers and callees of original and refining contracts.

the contracts that are composed. Thus the properties do not hold for all contracts c and c', which is a contradiction to our assumption that the mechanism is both, original-caller-preserving and refinement-callee-preserving. □

Theorem 10. *There exists no contract-composition mechanism that is both, original-callee-preserving and refinement-caller-preserving.*

Proof. Analogous to Theorem 9. □

By means of Theorem 9, we know that a contract-composition mechanism cannot return a contract c'' that is caller-compatible with respect to c and callee-compatible with respect to c' for all c, $c' \in C$. Nevertheless, the proof indicates that if we would restrict our contracts $c = \{\phi\}m\{\psi\}[\alpha]$ and $c' = \{\phi'\}m'\{\psi'\}[\alpha']$ to fulfil $\phi' \vDash \phi$, $\psi \vDash \psi'$, and $\alpha = \alpha'$, the resulting contract c'' could be caller-compatible with respect to c and callee-compatible with respect to c'. However, such restriction on the input contracts for contract composition are not in our scope and would hinder compositionality, because we would have to check that all possible compositions of contracts fulfil the restrictions on preconditions, postconditions, and the framing condition.

With Theorems 9 and 10, we conclude that at most two of the four preservation properties can be fulfilled by a contract-composition mechanism. We illustrate the possible combinations of the preservation properties also using an Euler diagram in Fig. 2. Overall, there are nine options for mechanisms with respect to the preservation properties. First, one option is to fulfil no preservation properties (white area in Fig. 2). Second, there are four options to fulfil exactly one of these four properties (red, orange, green, and blue). Finally, fulfilling two out of four properties give rise to further four options (mixed colors). Theorem 9 rules out overlapping between green and red, and Theorem 10 rules out overlapping between orange and blue. As explained above, fulfilling more of these preservation properties improves modular reasoning. Hence, of these nine options, the four options fulfilling two preservation properties seem to be most promising. For that reason, we refer to them as *fundamental options* for contract composition and introduce names for them:

Definition 11. A contract-composition mechanism m is called *caller-preserving* if and only if it is original-caller-preserving and refinement-caller-preserving. A mechanism m is called *callee-preserving* if and only if it is original-callee-preserving and refinement-callee-preserving. A mechanism m is called *original-preserving* if and only if it is original-caller-preserving and original-callee-preserving. A mechanism m is called *refinement-preserving* if and only if it is refinement-caller-preserving and refinement-callee-preserving.

The four fundamental options for contract composition have different properties, although they all support some kind of modular reasoning. A caller-preserving mechanism enables modular reasoning for callers. That is, if a method m calls a method n with contract c defined in the same module (e.g., aspect, feature module, or delta module), we can rely on the contract c without a need to consider other contracts defined for method n in other modules. Similarly, a callee-preserving mechanism enables modular reasoning for callees. That is, a method m has to fulfil only what is defined in the contract of the module. Contracts for method m defined in other modules can only strengthen preconditions or postconditions. In contrast, an original-preserving mechanism ensures that contracts may only be replaced by equivalent contracts (i.e., not refined at all). Hence, caller and callee can rely on this contract independent of later modules. A refinement-preserving mechanism basically allows to completely replace the contract, but callers and the callee do not need to consider previous modules.

Besides the four preservation properties, other interesting properties of contract composition are commutativity, associativity, and idempotence. Contract composition is *commutative*, if the order of contracts in composition does not matter (i.e., $c' \bullet c \equiv c \bullet c'$). Contract composition is *associative*, if different parentheses in the composition of more than two contracts result in equivalent contracts (i.e., $c'' \bullet (c' \bullet c) \equiv (c'' \bullet c') \bullet c$). Contract composition is *idempotent*, if the composition of two identical contracts yields an equivalent contract (i.e., $c \bullet c \equiv c$).

Contract composition is closely related to the composition of source code, because contracts are typically embedded in source code (Meyer, 1988). Commutativity can facilitate comprehension, even if the composition of source code is often not commutative, because method refinements may refer to previous method implementations (Höfner and Möller, 2009; Apel et al., 2010). Usually, there is a special keyword to do so; for example, keyword `super` or `Precursor` in object-oriented method overriding (Meyer, 1988; Bracha and Cook, 1990; Gosling et al., 2005), keyword `proceed` or `runNext` in aspect-oriented around advice (Kiczales et al., 2001), and keyword `original` in feature-oriented programming (Apel et al., 2013b) and delta-oriented programming (Schaefer et al., 2011). Hence, approaches enabling the composition of methods typically assume a partial order (i.e., a total order for each composition). For instance, delta modules in delta-oriented programming have to declare a partial order to all other modules refining the same methods (Schaefer et al., 2010). In feature-oriented programming, usually a total order on all features is assumed (Prehofer, 1997; Meinicke et al., 2017). In aspect-oriented programming, aspect precedence can be defined and if the order is not unique, the aspect compiler chooses an order (Kiczales et al., 2001). In object-oriented programming, an order is given by the inheritance hierarchy. Consequently, we can assume an order of composed contracts.

Our discussion of properties is independent of the composition of implementations. That is, contract composition can be commutative, even if the composition of source code is typically not commutative (Apel et al., 2010; Höfner et al., 2012). Commutativity, associativity, and idempotence are desirable properties for contract composition, because they may ease the understanding of contracts as the order, parentheses, and identical contracts do not influence resulting contracts.

5.3. Comparison of contract-composition mechanisms

In Section 4, we discussed six mechanisms for contract composition. We summarize the properties of these mechanisms in Table 2 and make several observations. All mechanisms are associative (i.e., $(c'' \bullet c') \bullet c$ is equivalent to $c'' \bullet (c' \bullet c)$) and, except for explicit contract refinement, also idempotent (i.e., $c \bullet c = c$). Only conjunctive, cumulative, and consecutive contract refinement are commutative (i.e., $c \bullet c' = c' \bullet c$). With respect to the previously defined preservation properties, plain contracting is original-

Table 2
Overview on contract-composition mechanisms and their properties.

Contract-composition mechanism*	Preservation property	Associativity	Idempotence	Commutativity
$c' \bullet_{PC} c = \{\phi\}m' \bullet m\{\psi\}[\alpha]$	Original-preserving	Yes	Yes	No
$c' \bullet_{CO} c = \{\phi'\}m' \bullet m\{\psi'\}[\alpha']$	Refinement-preserving	Yes	Yes	No
$c' \bullet_{ECR} c = \{\phi' \bullet \phi\}m' \bullet m\{\psi' \bullet \psi\}[\alpha' \bullet \alpha]$	None	Yes	No	No
$c' \bullet_{ConjCR} c = \{\phi' \wedge \phi\}m' \bullet m\{\psi' \wedge \psi\}[\alpha' \cap \alpha]$	None	Yes	Yes	Yes
$c' \bullet_{CumCR} c = \{\phi' \vee \phi\}m' \bullet m\{\psi' \wedge \psi\}[\alpha' \cap \alpha]$	Caller-preserving	Yes	Yes	Yes
$c' \bullet_{ConsCR} c = \{\phi' \vee \phi\}m' \bullet m\{\psi'' \wedge \psi'''\}[\alpha \cup \alpha']$	Caller-preserving	Yes	Yes	Yes

* $c = \{\phi\}m\{\psi\}$, $c' = \{\phi'\}m'\{\psi'\}$, $\psi'' = \big(old(\phi') \Rightarrow \psi'\big) \wedge \big(old(\phi) \Rightarrow \psi\big)$, and $\psi''' = (\neg old(\phi) \Rightarrow \forall_{x \in \alpha \setminus \alpha'} old(x) = x) \wedge (\neg old(\phi') \Rightarrow \forall_{x' \in \alpha' \setminus \alpha} old(x') = x')$.

preserving, because the caller view and callee view of the original contract are both maintained. Contrary, contract overriding is refinement-preserving, because caller and callee view of the refining contract are maintained. Explicit contract refinement and conjunctive contract refinement do not fulfill any preservation properties. The remaining two mechanisms, cumulative and consecutive contract refinement, are both caller-preserving. Proofs of all these properties can be found elsewhere (Thüm, 2015).

In the following, we discuss all contract-composition mechanisms with respect to four important criteria, namely *modular reasoning, allowed contract refinements, specification effort including specification clones and derivative contracts*, and *dangling references*.

Modular Reasoning. A property that is conflicting with support for arbitrary contract refinements is modular reasoning for callers. Modular reasoning is possible with plain contracting as no refinement is available. Furthermore, programmers and verification tools can easily reason about method calls, because the same contract holds for every possible combination of features.[1] The only other mechanisms enabling modular reasoning for callers are cumulative and consecutive contract refinement, but they only support contract refinement in a limited way (i.e., for callers as discussed in Section 5.2).

Allowed Contract Refinements. Plain contracting is the only mechanism that prohibits contract refinement when refining methods. For this mechanism, method refinements may change the behavior only such that the original contract is maintained. As a consequence, callers cannot rely on the changed behavior. For instance, if we replace the instable sorting algorithms heap sort and quick sort by a stable algorithm, such as merge sort, we may want to express that callers can rely on stability, which is impossible with plain contracting. cumulative and consecutive contract refinement enable contract refinement by combining original contracts with new preconditions and postconditions.

For contract overriding and explicit contract refinement, when a method refinement provides some new guarantees, we can actually specify them in a refining contract and callers can rely on it. In addition, because we can arbitrarily refine contracts, also all method refinements are possible and do not have to adhere to the original contract. That all contract refinements and all method refinements are possible, provides flexibility particularly with respect to unanticipated changes.

For conjunctive and cumulative contract refinement, to be compatible with the conjunction of postconditions, the frame is only allowed to become smaller. Introducing new fields to the frames of refining contracts is thus impossible. However,it is possible to abstract away from specific fields by creating and allowing groups in frames, where new fields can be inserted into (Leino, 1998; Weiß, 2011). A further disadvantage of conjunctive contract refinement is that only a limited form of contract refinements can be expressed. In principle, contract refinements can only add formulas to preconditions and postconditions in conjunction to existing ones. As a result, we might have to remove some contracts to enable certain method refinements. In the worst case, we may only be able to specify a small portion of the product-line behavior, and thus only detect some errors of the product line. For example, conjunctive contract refinement is too restrictive to specify the contract refinement shown in Listing 5. We evaluate the practical implications of this limitation in Section 8.

Using cumulative or consecutive contract refinement, we can easily create contracts that are hard to fulfill for callees, or there might not even be a single implementation as the contract refinement is contradictory. In the end, this restrictiveness with respect to the addition of preconditions and postconditions may lead to the fact that many interesting properties of a given product-line implementation cannot be specified, which we empirically investigate in Section 8. The examples given in 5 and 7 cannot be expressed by means of cumulative contract refinement.

Specification Effort. The simplicity of plain contracting may facilitate creation and maintenance of contracts for programmers. A programmer only needs to specify a method once, even if it is refined by several other feature modules, which reduces the effort for specification (i.e., writing contracts).

For contract overriding, any contract may be subject to later refinement. This is challenging, as *callers need to be aware of any contract* for a given method in order to determine the contract they can rely on. Consequently, it may require to clone and adapt previous contracts. For example, the refined contract in Listing 5 repeats the complete precondition and postcondition of the original contract. We refer to such cloned contracts as *specification clones* (Farrell et al., 2017). Cloning contracts is necessary here because contract overriding only supports completely replacing contracts without any mechanism to reuse existing contracts. Furthermore, if two or more features refine the same contract using contract overriding, we may get undesired contracts if both features are selected. A solution is to introduce *derivative contracts* (i.e., a contract that is only included if two or more features are selected). However, derivative contracts may not scale for many contract refinements of the same method. A solution is to introduce *derivative contracts* (i.e., a contract that is only included if two or more features are selected). However, derivative contracts may not scale for many contract refinements of the same method.

Explicit contract refinement improves over contract overriding by providing a means to explicitly refer to previous preconditions and postconditions and, thus, a means to avoid some specification clones and derivative contracts. Furthermore, explicit contract refinement overcomes some drawbacks of contract overriding. First, programmers can avoid most *specification clones*, because they have a linguistic means to refer to original preconditions, postcondi-

[1] Although contract refinement is not possible with this approach, there can be different contracts for the same method when alternative features introduce the same method with a different contract. However, such cases can be forbidden and their absence could be automatically verified by means of a static analysis.

tions, and frames individually and do not have to clone and adapt contracts for all method refinements. Second, the need for *derivative contracts* is reduced as we can refer to a previous contract from whatever feature module this contract may be provided, and thus supporting compositional flexibility. Nevertheless, in some cases specification clones and derivative contracts may still be necessary (e.g., if we want to change just a small part within a larger precondition). However, in explicit contract refinement, specifications may get more complex than in contract overriding if several refinements for the same method contract exist and some, but not all refinements refer to the previous contracts. Furthermore, a new dimension of complexity arises due to the possibility to independently refer to preconditions, postconditions, and frames, and that it is even possible to negate preconditions or postconditions or to use them in some new logical context. Again, it may be hard for a programmer to retrieve the contract for a certain context, which may be mitigated by means of tool support.

Compared to explicit contract refinement, there is no specification effort for conjunctive contract refinement with respect to providing the keyword original. Furthermore, resulting contracts are easy to understand as all preconditions and all postconditions are simply concatenated. That is, a later or potentially unknown feature module can only change the contract in this limited way. Using conjunctive contract refinement, it is possible to avoid some specification clones and derivative contracts due to the fact that previous preconditions and postconditions are assumed to hold in all cases.

Cumulative and consecutive contract refinement may lead to specification clones, as adapting the postcondition also requires to specify the precondition. due to the disjunction of the preconditions. We need to evaluate this further.

Dangling References. References to previous contracts introduce the possibility of dangling references. That is, we may use the keyword original in a contract, but during composition we detect that there is no original contract to which the keyword can point to. For example, we may define a method introduction including a contract in an optional feature and use the keyword original to refer to this contract from another optional method refinement. This is a particular instance of an unwanted feature interaction (Calder et al., 2003). As typical for feature interactions, this dangling reference may only occur in some feature combinations and thus stay unnoticed until feature modules are composed for one of these combinations. However, inserting true for non-existent preconditions and postconditions might be reasonable. In particular, explicit contract refinement relies on similar linguistic means as feature modules and may therefore lead to an additional source of errors in contracts due to dangling references for keyword original.

5.4. Summary

Based on our discussions, we cannot yet rule out mechanisms or even favor a single mechanism. There are some indicators that consecutive contract refinement is superior to cumulative contract refinement, as the advantages and disadvantages of both are almost the same, except that callers on consecutive contract refinement cannot only provide one of the preconditions and rely on all postconditions. Consequently, it is not the burden of callees to fulfil all postconditions for any given precondition. Moreover, there are some indicators that explicit contract refinement is superior to contract overriding. Nevertheless, an empirical evaluation in Section 8 is required to judge the practical relevance of these six mechanisms.

6. Composition beyond preconditions, postconditions, and framing conditions

For brevity, our considerations in Sections 4 and 5 only focused on contracts consisting of a precondition, a postcondition, and a framing condition. However, there are numerous advanced specification concepts based on the notion of contracts (Meyer, 1988; Chalin et al., 2005; Hatcliff et al., 2012). In this section, we discuss some of the most relevant concepts and how our previous discussions relate to these concepts. In particular, we discuss specification cases, multiple preconditions and postconditions, class invariants, as well as pure methods.

6.1. Specification cases

In the principle of design by contract, every contract consists of exactly one precondition and one postcondition (Meyer, 1988). However, specification languages such as JML support the use of multiple *specification cases* for one method, by connecting them with keyword also (Chalin et al., 2005). Roughly speaking, a method is specified with multiple contracts, which are connected by contract composition. In fact, keyword also is just syntactic sugar and can be desugared whenever required, as described by Chalin et al. (2005): given k specification cases combined with keyword also and their respective precondition ϕ_i, postcondition ψ_i, and framing condition α_i with $i = 1, \ldots, k$, the combined contract, where keyword also is desugared, has precondition $\bigvee_i \phi_i$, postcondition $\bigwedge_i \backslash\mathbf{old}(\phi_i) \Rightarrow \psi_i$, and frame condition $\alpha = \bigcup_i \alpha_i \,\mathrm{if}\, \phi_i$.

In theory, we can easily apply all contract-composition mechanisms as defined above by desugaring all specification cases prior to composition. In practice, the resulting contracts may turn out to be hard-to-read, especially if there are more than two specification cases involved. The problem arises when the result of contract composition is presented to the user. One application that requires programmers to read these contracts is if we generate a documentation for a single product of our product line. Another application is when the contracts are used for runtime assertion checking or verification and the programmer is supposed to understand the contract to locate the faulty feature module or combination of feature modules. Consequently, desugaring during composition could be considered undesirable.

To avoid desugaring of keyword also, we can extend contract-composition mechanisms with support for specification cases. A trivial extension exists for plain contracting and contract overriding; instead of copying the original and refining precondition and postcondition, respectively, one can simply copy the complete original or refining contract during composition. Extending consecutive contract refinement is straightforward as it is equivalent to specification cases anyway. That is, one may combine contracts from different feature modules by means of keyword also, too. In other words, consecutive contract refinement can be implemented through specification cases.

Supporting specification cases in explicit contract refinement is slightly more complex. This is because keyword original may also be used in specification cases or refer to a contract with specification cases. There are several options what keyword original refers to. First, it may refer to the desugared precondition or postcondition of the original contract. This would require some desugaring only during the replacement of keyword original. Second, the keyword may refer to the precondition or postcondition of a particular specification case. As specification cases do not have an identifying name, a simple strategy is to assume that keyword original defined in the ith specification case of a refining contract refers to the ith specification case of the original contract. We refer to a bachelor's thesis introducing new keywords that enable to choose between one of these two semantics (Benduhn, 2012).

In contrast to these four contract-composition mechanisms, it seems that desugaring cannot be avoided for conjunctive contract refinement and cumulative contract refinement. However, specification cases are an elegant way for implementation of consecutive contract refinement, because it avoids the cloning of preconditions into postconditions during composition (cf. Section 4.6).

6.2. Multiple preconditions and postconditions

Besides the definition of multiple specification cases each consisting of a precondition, a postcondition, and a framing condition, one may also define multiple preconditions, postconditions, and framing conditions for a single specification case. Similar to specification cases, multiple preconditions and postconditions are just syntactic sugar and can be conjoined into one (Beckert et al., 2007; Benduhn, 2012). The framing condition is only a set of locations. Multiple framing conditions $\alpha_1, \alpha_2, \ldots, \alpha_n$ are replaced by a single framing condition comprising their union (i.e., $\bigcup_{1 \leq i \leq n} \alpha_i$).

As for specification cases, we may apply desugaring with respect to multiple preconditions and postconditions before composition to reuse contract-composition mechanisms as presented above. However, desugaring is, again, not required for plain contracting and contract overriding, as copying the contracts is sufficient. The same can be said about consecutive contract refinement when implemented by means of specification cases (as discussed in Section 6.1). For conjunctive contract refinement, preconditions and postconditions are composed by conjunction, which can be implemented with multiple preconditions and postconditions. For cumulative contract refinement, the same is true for postconditions, but preconditions need to be desugared as described above prior to composition. Finally, explicit contract refinement poses similar challenges as specification cases do, because keyword original can either refer to the desugared precondition or postcondition, or refer to the *j*th precondition or postcondition of the *i*th specification case. For simplicity, we assume the former option.

In summary, most contract-composition mechanisms can directly be extended to multiple preconditions and postcondition for input contracts. Some of them can even take advantage of this language construct to avoid lengthy preconditions and postconditions.

6.3. Pure methods and model methods

Contracts may contain calls to pure methods, i.e., methods that terminate and are side-effect free (Beckert et al., 2007). Pure methods require attention during composition, as their refinement implicitly refines all those contracts calling this method. That is, by refining one pure method, we can refine several contracts indirectly at the same time. In the following, we refer to the refinement of pure methods as *pure-method refinement*. Interestingly, pure-method refinement enables contract composition by means of traditional feature-oriented method composition and does not require new language concepts nor new mechanisms for contract composition.

Example 12. In Listing 9, we give an example of pure-method refinement in a feature-oriented database implementation for student exams. Class ExamDatabase stores the results of student exams in array students, 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. While our example just shows one method contract with one pure-method call, in principle, a pure method could be called from several contracts and a contract may contain calls to several pure methods.

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

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

Listing 9. *Pure-method refinement* in product line *ExamDB*: the contract of method consistent contains a call to the pure method validStudent. Feature module *BackOut* refines the contract of method consistent indirectly by refining method validStudent (Thüm et al., 2012).

Model methods represent an alternative to pure methods. A *model method* is a method introduced only for specification purposes, can only be called within the specification, and is invisible for the implementation (Chalin et al., 2005; Hatcliff et al., 2012). With refining a model method, one also indirectly refines all contracts calling this it. Model methods come with the advantage of not requiring any specifications to be encoded into the underlying programming language. In contrast, pure methods allow for the same refinement to be made use of in specification *and* implementation. Consequently, it depends on the situation whether pure methods or model methods should be refined. Without loss of generability, we focus exemplarily focus on pure-method refinement in the following.

All mechanisms discussed in Section 4 may or may not be enriched with pure-method refinement, resulting in twelve mechanisms overall. However, pure-method refinement unfortunately breaks almost all aforementioned properties: preservation, idempotence, and commutativity. This is due to method refinement (e.g., in feature-oriented programming with keyword original) being neither idempotent nor commutative (Apel et al., 2010). In addition, contracts containing method calls may also be arbitrarily changed by means of refining pure methods. One necessarily buys into these issues when opting for pure-method refinement and, therefore, has to carefully weigh the consequences. If the refinement of pure-methods that are is forbidden, tools could check for possible violations.

6.4. Class invariants

Apart from method contracts, the design-by-contract paradigm also allows for defining conditions for classes by so-called *class invariants*. Depending on the visibility (i.e., public in JML) of a given class invariant, it applies to all methods with the same or higher visibility (Leavens and Müller, 2007). A class invariant is equivalent to adding the same condition to all preconditions and postconditions of methods that it applies to. Except for inheritance hierarchies with unknown subclasses (i.e., all open, object-oriented systems), class invariants are often just syntactic sugar. For example, an invariant defined for the class AbstractSet in the Java platform cannot be expressed by means of preconditions and postconditions, because we simply do not know all subclasses this invariant applies to. Nevertheless, for our discussion of the class-invariant refinements, it is sufficient to consider them as syntactic sugar.

For each of our six contract-composition mechanisms, we derive a strategy for dealing with the refinement of class invariants. In plain contracting, class invariants desugared into pre- and postconditions cannot be refined as because plain contracting prohibits any such refinement. In contrast, there are no concerns about arbitrarily refinements of class invariants in contract overriding or explicit contract refinement, as preconditions and postconditions can be arbitrarily refined, too. Similar to preconditions and postconditions in explicit contract refinement, we can facilitate class-invariant refinement by introducing keyword original in invariants. For conjunctive contract refinement and cumulative contract refinement, invariants may also be changed arbitrarily. However, for these two cases, refining a class invariant from i to i' requires that the postcondition $i' \wedge i$ is satisfiable. For example, we cannot refine an invariant by its negation. Otherwise, the postcondition cannot be fulfilled by any method implementation. In summary, refinement of class invariants is possible for all mechanisms except for plain contracting.

A rather technical problem for the refinement of class invariants is their lack of identifiers. We may define several invariants for one class without providing a unique name for them. Furthermore, invariants are not assigned to a particular member of a class. In contrast, a method contract is always assigned to a particular method that can be uniquely identified by its fully qualified name. The identification problem of class invariants results from most contract languages not being *feature-ready*: Apel et al. (2010) define properties a language must fulfil to apply feature-oriented composition. They discuss a similar problem with XML and the solution is to simply introduce names by either extending the contemporary language or by adding an overlaying module structure.

However, a restrictive refinement of class invariants does not require a unique identification. That is, if an existing invariant is refined by adding further terms in a conjunction, introducing a new invariant instead solves the issue. This solution works as multiple invariants in a single class are syntactic sugar for a single invariant being the conjunction of all of them. Hence, by adding new invariants in feature modules, one can implicitly refine invariants in this restrictive way, which is similar to conjunctive contract refinement. By introducing invariants in feature modules, we achieve variable class invariants customizable by the feature selection. In particular, an invariant defined in a certain feature module does only need to be established if the feature module is selected. We refer again to Listings 4 and 7, in which class invariants are introduced in core features as well as optional features.

7. Tool support for specifying feature-oriented contracts

So far, we have discussed numerous options for feature-oriented contract composition. This discussion included six mechanisms for contract composition, as well as design questions about how to deal with pure-method calls and class invariants. We aim to answer remaining questions by means of an empirical investigation of product-line specifications in Section 8. However, a comprehensive empirical investigation requires tool support for specifying feature-oriented contracts and for their composition. In this section, we describe the tool support that has been developed since 2012. In Section 7.1, we discuss our extensions to FeatureHouse, a tool for feature-oriented composition with support for JML. Based on our FeatureHouse extension, we implemented tool support in Eclipse by extending FeatureIDE, which we present in Section 7.2.

7.1. Automating contract composition with FeatureHouse

The goal of feature-oriented contract composition is to define feature-oriented contracts, which can be *composed automatically* for a given configuration. Hence, tool support for feature-oriented contract composition must provide a language to define feature-oriented contracts and a composer taking feature-oriented contracts as input and returning the resulting contract. The composition of feature-oriented contracts should be automated, because manual composition would need to be done whenever one of the feature modules evolves. Manual composition, however, is a laborious and error-prone task that may even lead to wrong results in a subsequent verification of products.

As we strive to build tool support for feature-oriented contracts based on *existing tools*, we aim to extend an existing language for method contracts. We have chosen to extend JML, because many verification tools are available for JML (Burdy et al., 2005). Furthermore, instead of building a new feature-module composer from scratch, we rather extend an existing one; available composers are FeatureC++ (Apel et al., 2005), AHEAD (Batory, 2006), and FeatureHouse (Apel et al., 2013b). While FeatureC++ only supports the composition of feature modules written in a feature-oriented extension of C++, AHEAD and FeatureHouse support a variety of programming languages. We decided to extend FeatureHouse as it already supports Java 1.5 and a new language can be integrated by providing a new grammar annotated with information about composition rules (Apel et al., 2013b).

We have developed a JML grammar according to the JML language levels 0–3 (Leavens et al., 2013) (i.e., from fundamental features to exotic and often not so well understood features), based on the existing FeatureHouse grammar for Java 1.5. The six contract-composition mechanisms discussed in Section 4 are implemented by means of a new command-line parameter of FeatureHouse. That is, a user can choose *one mechanism for a product line* and then compose feature modules of selected features automatically. The main part of this tool support was developed in the course of a bachelor's and a master's thesis (Benduhn, 2012; Bolle, 2017).

An interesting question to address in the evaluation is whether several mechanisms could be combined in a one hierarchy when specifying a single product line. Otherwise, a single method of a product line may rule out certain composition mechanisms and certain preservation properties. By means of keywords, we can choose a *contract-composition mechanism for each method* individually.

During implementation, we have had to make two design decisions, we briefly touch on in the following. First, we argue that there is no need for separate keywords for contract overriding and explicit contract refinement. The difference of both is that explicit contract refinement allows for the usage of keyword original, but there is no reason to forbid this keyword. Hence, explicit contract refinement subsumes contract overriding as the developer may decide to not use keyword original at his choice. Second, we decided to set *one mechanism as default* to save effort when specifying keywords. We argue that this should be explicit contract refinement,

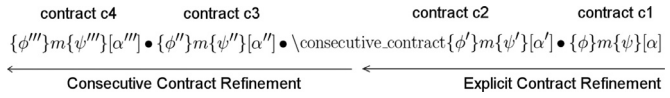

Fig. 3. Overriding a contract-composition mechanism with another mechanism (Weigelt, 2013).

as it does not pose any restrictions on the possible method refinements in feature-oriented programming. In particular, it would be counter-intuitive if the default in a specification technique in feature-oriented programming is restrictive, while feature-oriented programming is not.

We propose the following *contract-composition keywords* for the remaining mechanisms. The keyword for plain contracting is final_contract, because the contract cannot be refined at all. Similarly, we can forbid the refinement of a pure method with the keyword final_method. The latter keyword may actually also be applied to non-pure methods to indicate the developer can rely on the method's implementation independent of subsequent feature modules. For all remaining mechanisms we propose keywords based on their name: conjunctive_contract, cumulative_contract, and consecutive_contract. We refer interested readers to a bachelor's thesis (Weigelt, 2013) providing more details on the choice of the keywords.

We propose to define contract-composition keywords at the beginning of contracts. However, the ability of feature-oriented programming for method refinement naturally raises the question whether it should be possible to *override contract-composition keywords*. With overriding, we could use different contract-composition mechanisms for different refinements of the same method. We illustrate the overriding of keywords in Fig. 3. For brevity, we assume an empty frame. Contract c_1 and c_2 are composed using explicit contract refinement as no keyword is specified in contract c_1. Keyword consecutive_contract defined in contract c_2 overrides the contract composition with consecutive contract refinement for later refinements (e.g., when applying the contract refinements c_3 and c_4).

The overriding of contract-composition keywords provides more flexibility for refinement, but may *break preservation properties*. For instance, consider a contract using consecutive contract refinement that is then refined by one with explicit contract refinement. This overriding drops modular reasoning for method callers, as later feature modules may completely replace existing preconditions and postconditions. To prevent keyword overriding from breaking preservation properties, we only permit overriding if guarantees are preserved.

We refer to the contract composition with the aforementioned keywords and possible overridings as *feature-oriented contract composition*. With our extension to FeatureHouse, refinement is supported through any of the contract-composition mechanisms discussed in Section 4 and feature-oriented contract composition.

7.2. Supporting feature-oriented contracts in FeatureIDE

The command-line tool FeatureHouse has been integrated into Eclipse in the FeatureIDE project (Meinicke et al., 2017; Kästner et al., 2009b), together with several other product-line implementation tools. FeatureIDE is a framework for product-line implementation providing editors with syntax highlighting and content assistants as well as views specific to feature-oriented programming supporting the whole development process (Meinicke et al., 2017). We have integrated our extension of FeatureHouse into FeatureIDE (a) to provide a convenient use of contracts for student in lectures on product lines, (b) to ease the transfer of research results to practice, and (c) to implement error reporting for wrong usage of contracts based on the FeatureIDE infrastructure. Our extensions are open-source and available as part of the FeatureHouse and FeatureIDE projects.[2]

In our FeatureIDE extension, the contract-composition mechanisms can be chosen in the properties of Eclipse projects. More specifically, the Eclipse project needs to be a regular FeatureIDE project with FeatureHouse selected as the product-line generation tool. A developer can change the contract-composition mechanism for each project at any time of development, which is especially useful to compare different approaches. In addition, several views in FeatureIDE are extended to support contracts (Proksch and Krüger, 2014). The outline view for feature-oriented Java files indicates which methods are specified by means of a contract. The collaboration diagram, a view giving an overview on all feature modules and their mapping to classes, has options to show contracts or filter methods with contracts. Finally, the statistics view showing metrics on FeatureIDE projects is enriched with several metrics on contracts.

8. Empirical evaluation of feature-oriented contracts

The above sections raise several research questions that we aim to answer by means of an empirical evaluation.

RQ-1 To what extent do feature modules require variability in contracts and contract refinement?
RQ-2 Are refinements of feature-oriented contracts typically original-preserving, refinement-preserving, caller-preserving, or callee-preserving?
RQ-3 Which of the six contract-composition mechanism discussed in Section 4 is superior to others in practice?
RQ-4 What can we say about the granularity of contract refinements, which describes whether only the precondition, the postcondition, the framing condition, or a combination thereof are refined?

We give on overview on the subject product lines in Section 8.1. In Section 8.2, we share the insights gained with our case studies. Finally, we discuss possible threats to validity in Section 8.2.

8.1. Case studies

As we propose the concept of feature-oriented contracts, we cannot expect to find and study real product lines specified with feature-oriented contracts. We used three *strategies to create product lines* with feature-oriented contracts. First, we implemented product lines and feature-oriented contracts from scratch. Second, we decomposed existing, object-oriented programs, which were formally verified before, including their contracts into a product line. That is, we identified features of the program and separated them into feature modules. Third, we specified existing product lines with feature-oriented contracts. Each of these creation strategies is a typical application scenario of employing feature-oriented contracts and may impose different requirements for contract-composition mechanisms.

The rationale behind *developing product lines from scratch* is that a given language for contracts may impose restrictions on the design. However, when developing a feature-oriented program from scratch, we can try to come up with a design that fits the language. In the end, we know whether a certain contract-composition mechanism is feasible when building modules and their contracts from scratch. Overall, we have implemented and specified five feature-oriented product lines, which consist of data structures (*IntegerList*), algorithms (*UnionFind, StringMatcher*), and combinations thereof (*BankAccount, GPL-scratch*).

[2] https://featureide.github.io/.

When *decomposing existing programs* with contracts into feature modules, we benefit already available rich specifications from existing software. Thus, it is possible to evaluate whether such specifications can be decomposed into features by means of a given contract-composition mechanism. We have used this approach for variational data structures (*IntegerSet, Numbers*), libraries (*DiGraph, ExamDB*), and stand-alone systems (*Paycard, Poker*).

Existing feature-oriented systems, in which contracts have not been specified during development, may require certain kinds of variability that are not supported by some contract-composition mechanisms. With the other two approaches, we might miss such situations, because specifications are considered from early on. By developing contracts for existing modules, we evaluate how contract-composition mechanisms can be used to specify systems with typical variability characteristics. We have enriched existing feature-oriented systems with contracts, such as a library (*GPL*) and stand-alone systems (*Elevator, Email*). For product lines *Elevator* and *Email*, we formalized existing informal specifications in JML. For product line *GPL*, we specified contracts based on the existing design and domain knowledge.

In total, we conducted 14 case studies with a different product line each. According to our above mentioned tool support, all studied product lines are implemented in feature modules based on Java and feature-oriented contracts based on JML, but we expect similar results for other object-oriented languages and contract-based specification languages. As our work focuses on *specification*, the primary goal was not to make contracts strong enough to facilitate a full verification. Nevertheless, two product lines have been fully verified through static verification. For most of the other case studies, feature-oriented contracts serve documentation purposes. All 14 subject product lines are publicly available through the example wizard in FeatureIDE.[3]

8.2. Results and insights

In the following, we share our results of our case studies and gained insights. First, we aim to answer to which extent feature-oriented contracts are variable and whether contract refinement is actually needed (**RQ-1**). Then, we discuss which contract preservation properties are established by the contract refinements in our subject product lines (**RQ-2** and **RQ-3**). Finally, we discuss how fine-grained typical contract refinements are and the consequences of granularity that we experienced (**RQ-4**).

Variability in Contracts. We proposed feature-oriented contracts as a means to define variable product-line specifications. However, it is unclear to what extent typical feature-oriented contracts are variable. In particular, while our tool support makes possible the automatic generation of contracts for each product, the question arises whether these product specifications actually differ from each other. No differences would occur if all contracts and invariants are defined in feature modules that do belong to core features (for short *core contracts* and *core invariants*). After creating the subject product lines, we counted the number of method contracts and class invariants defined in core features and the overall number of contracts.

In Fig. 4, we display the percentages of core contracts and core invariants compared to all contracts and invariants, respectively. Missing numbers for some product lines indicate that invariants were not existent and thus the percentage is undefined (due to division by zero). Every product line contains contracts that are *not* defined in core features. Except for product line *DiGraph*, every product line with invariants also contains invariants not defined in core features. That is, the generated specification is typically similar between products, but not identical. In particular, for the product lines *Elevator* and *Email* there are products that do not share any specifications, because they do neither contain core contracts nor core invariants. The average over all product lines is that 29% are core contracts and 54% are core invariants, but we expect smaller percentages for larger product lines.[4]

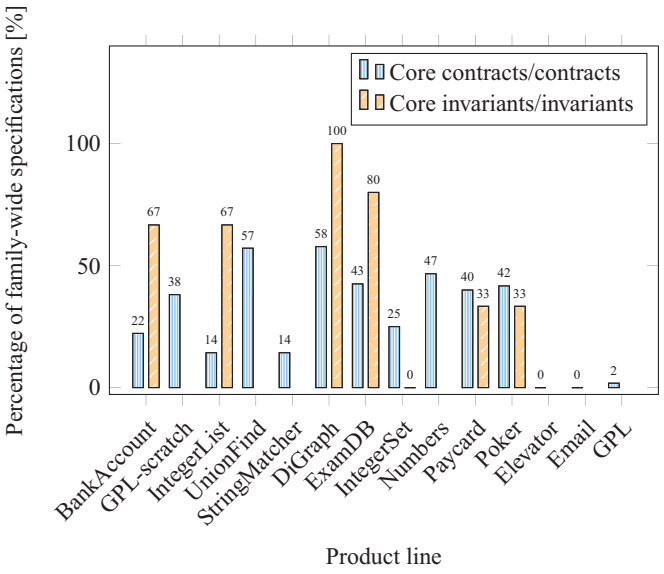

Fig. 4. Percentage of family-wide specifications compared to all specifications.

We can define a specification for a particular product line that is assumed to hold for all products, called family-wide specification (Thüm et al., 2014). Our case studies, however, indicate that family-wide specifications are not feasible, at least when specifying product lines by means of contracts. Hence, it seems that the code-level specification of variable code requires variable specifications, which are supported by all contract-composition mechanisms discussed above. Even with plain contracting we can introduce contracts in optional feature modules to create variation in specifications.

Nevertheless, we may define a family-wide specification by means of feature-oriented contracts. That is, we can decompose a family-wide specification into several feature modules belonging to core features. Indeed, such a decomposition has been done in product line *Poker*. In Fig. 5, we show an excerpt of the collaboration diagram, in which product-line specifications common to all products are split up to three core features. Reasons for such a decomposition are manifold. For example, the decomposition separates concerns that different developers need to consider during evolution. In this sense, feature-oriented contracts are more general than defining contracts that all products must establish.

We may raise a similar research question, namely whether we need to explicitly define the behavior of feature combinations or whether we can define the intended behavior of a product line by specifying each feature in isolation. Our results are in line with previous research (Apel et al., 2013c) and indicate that the former approach (i.e., family-wide specification) is not enough, whereas the latter approach (i.e., feature-based specification) indeed suffices. This question resembles the question whether product lines can be implemented by means of feature modules.

[3] https://featureide.github.io/.

[4] We compute all average values by computing the percentage of each product line and then calculating the average. That is, we do not sum up the values of all product lines and then calculate the average, because then larger product lines would have more influence on the result.

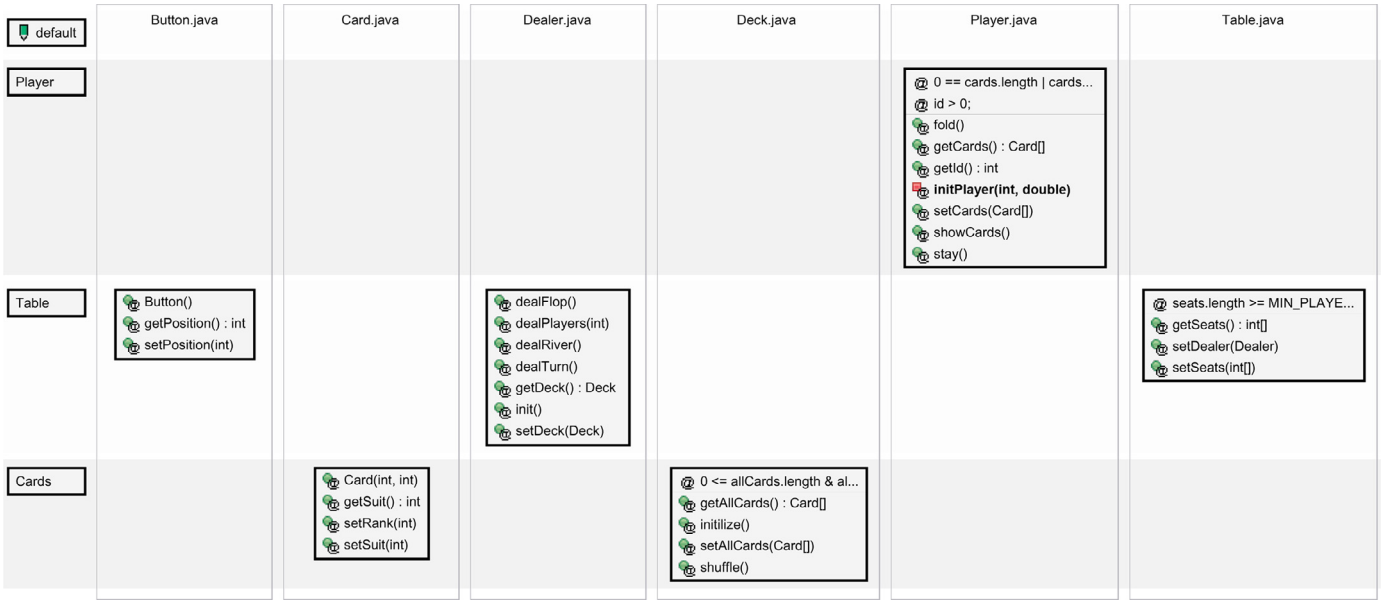

Fig. 5. Collaboration diagram showing all core contracts and core invariants of product line *Poker*.

Liu et al. (2006) experienced that implementing a feature module for each feature has not been sufficient when they decomposed a legacy system into a product line, but they can easily be enriched by additional derivative modules (a.k.a. lifters (Prehofer, 1997)). A derivative module is a feature module that is added whenever two or more specific features are chosen.

In our case studies, we made similar experiences when decomposing contracts into feature-oriented contracts. For almost all programs that we migrated to a product line, we were able to specify all contract refinements by means of contract-composition mechanisms as discussed in Section 4 – without a need for derivative modules.

As an exception, massive use of derivative modules turned out to be necessary for product line *ExamDB*. In fact, we had to create all theoretically possible derivative modules. For one core feature and three independent-optional features, we had to create three second-order derivatives and one third-order derivative (cf. Fig. 6). We discuss the reason and possible solutions below. For creation strategies beyond migration (i.e., development from scratch and specification of existing product lines), we only needed derivatives in *GPL*. However, *GPL* already contained ten derivatives for implementation purposes and specifying the product line with feature-oriented contracts did not require further derivatives. Overall, our experience is that feature-based specification is sufficient, because derivative modules could be completely avoided in our case studies by choosing a feasible contract-composition mechanism for each product line.

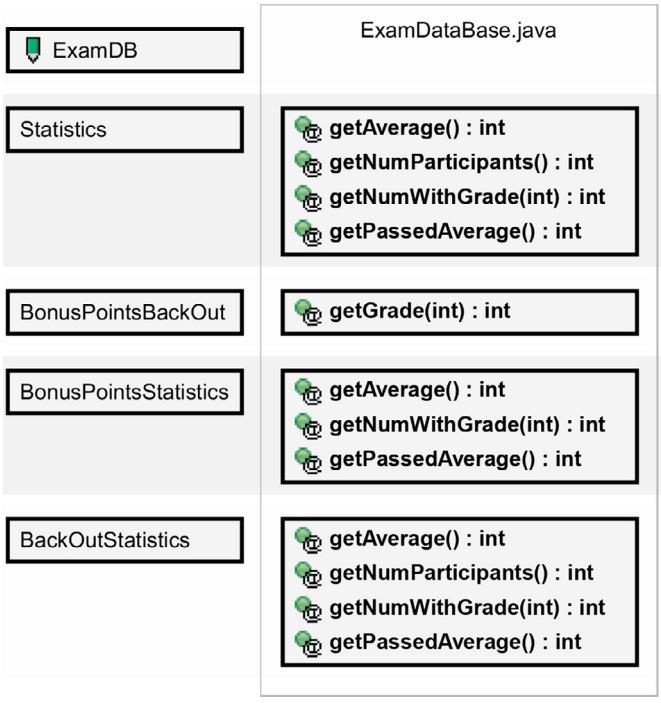

Fig. 6. The derivative modules of product line *ExamDB* cover all combinations of the optional features *BonusPoints*, *BackOut*, and *Statistics*.

The Need for Contract Refinement. Although we have seen that feature-oriented contracts are variable to a large extent, this does not necessarily mean that there is a need for contract refinement. A method with a contract that is introduced in an optional feature is variable (i.e., not part of all products), even if never refined. Thus, it is questionable whether there is a need for contract-composition mechanisms that allow programmers to refine existing contracts (i.e., all mechanisms except plain contracting).

In Fig. 7, we show the percentage of contract refinements with respect to the overall number of contracts. We discover that between 0% and 86% of all contracts refine another contract in at least one product. No contract refinements were necessary only for *DiGraph* and *IntegerSet*. Product line *DiGraph* does not contain a single method refinement that could have required contract refinement. In other words, the features chosen for decomposition do not cross-cut method implementations. Nevertheless, method and contract refinement could be necessary when extracting further features or extending *DiGraph* with new features. The product line *IntegerSet* contains several method refinements that all adhere to the initially introduced contract. Leaving this exception aside, all other product lines contain contract refinements. Hence, plain contracting is only applicable to all contracts in product lines *DiGraph* and *IntegerSet*.

In our subject product lines, several methods do have different contracts in different products. However, contract refinement is not the only option to achieve different contracts for the very same method; we experienced alternative introductions of con-

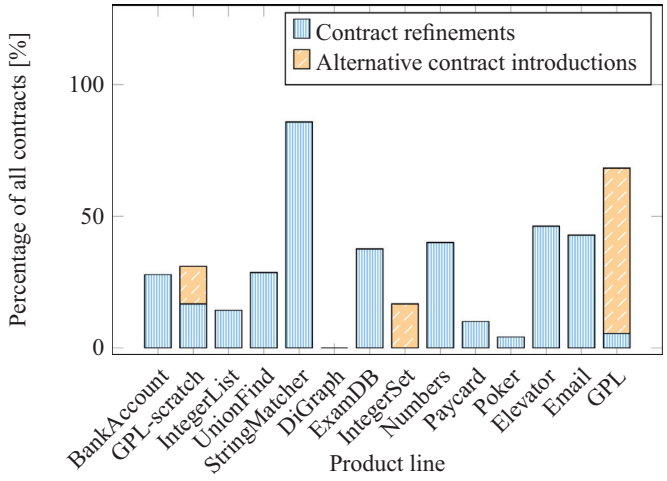

Fig. 7. Percentage of contract refinements and alternative contract introductions compared to all contracts.

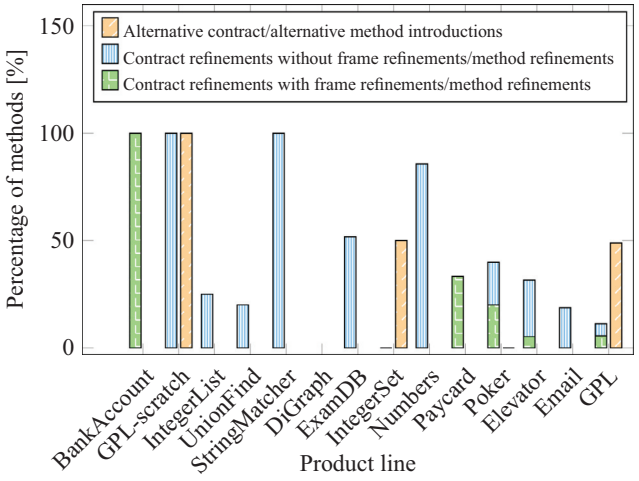

Fig. 8. Percentage of contract refinements (with and without frame refinements) and alternative contract introductions compared to method refinements and alternative method introductions.

tracts in three product lines, namely *GPL-scratch, IntegerSet*, and *GPL*. Interestingly, the three product lines have been created by means of a different strategy each (i.e., product line developed from scratch, specified program migrated to a product line, and specification of an existing product line). Hence, alternative contract introductions are not only caused by an existing program or product-line design. Based on Fig. 7, we make two observations. First, in product line *IntegerSet* all differing contracts are due to alternative contract introductions (i.e., there are no contract refinements). Second, in product line *GPL* more than half of the contracts are introduced in alternative features. Consequently, each contract-composition mechanism should also come with a strategy to deal with alternative contract introductions. Disallowing alternative contract introductions completely does not seem to be a valid option according to our experience with these case studies.

As previously discussed, not all method refinements require contract refinements. That is, even though the implementation of a method is refined, the resulting method still adheres to the original contract (cf. plain contracting in Section 4.1). In Fig. 8, we illustrate the percentage of method refinements that require the refinement of contracts and also that additionally require the refinement of the frame. Similarly, we illustrate the percentage of alternative method introductions that require alternative contract introductions. Missing bars indicate that the product line does not contain method refinements or alternative method introductions (i.e., the percentage is undefined). The percentages range from 0% to 100% over all product lines. On the one hand, as indicated above, product line *IntegerSet* has method refinements, but no contract refinements. On the other hand, all method refinements of product lines *BankAccount, GPL-scratch*, and *StringMatcher* require contract refinements. Similarly, product line *Poker* has alternative method introductions but no alternative contract introductions, whereas in product line *GPL-scratch* every alternative method introductions is specified by means of an alternative contract. In average over all product lines, 47% of the method refinements require contract refinements (i.e., plain contracting is sufficient for 53% of them), and 50% of the alternative method introductions require alternative contracts. Only product lines *BankAccount, Paycard, Poker, Elevator*, and *GPL* require *frame refinements*, ranging from 17% (*Elevator*) to 100% (*BankAccount, Paycard*). In summary, only 11 out of 60 contract refinements in total modified the framing condition.

In contrast to method contracts, alternative introductions and refinements of class invariants were not needed in our case studies. However, there was an interesting case when decomposing *ExamDB* into a product line, which is shown in Listing 10. The original source code contained a large invariant that we had to decompose to the feature modules *ExamDB* and *BonusPoints*. We decided to decompose them into two because the field `bonusPoints` was moved to feature module *BonusPoints*, and thus the reference to the field within the invariant had to be moved accordingly. Otherwise, the absence of feature *BonusPoints* would have resulted in a dangling method reference in JML. In this case, fortunately, we could decompose the original invariant into two invariants. Hence, the refinement of invariants was not necessary. Nevertheless, other product lines may actually require invariant refinements.

Contract Preservation. In Section 5.1, we discussed several preservation properties that contract composition may establish. In the first diagram of Fig. 9, we show to which extent contract refinements preserve preservation properties.[5] Each bar may reach 100% for each property. On average, the majority of contract refinements are original-caller-preserving (47%) or refinement-caller-preserving (32%). Only a small portion of all contract refinements are refinement-callee-preserving (25%) or original-callee-preserving (13%). Many contract refinements do not preserve any of these four properties (40%). In total, seven product lines contain contract refinements without preservation properties, namely *BankAccount, GPL-scratch, ExamDB, Paycard, Poker*, and *Numbers*. Interestingly, except for *GPL-scratch* and *ExamDB*, all contract refinements in the remaining product lines without preservation properties were due to the framing conditions. For an example of such contract refinements, we refer again to Listing 7. In this example, precondition and postcondition are strengthened at the same time using conjunctive contract refinement.

In the second diagram of Fig. 9, we show the percentage of contract refinements that adhere to the fundamental preservation options discussed in Section 5.2. The majority of all contract refinements do not establish any of the four fundamental options (62%). With respect to the discussed properties, the majority of contract refinements are caller-preserving (27%). Callee-preserving (12%), refinement-preserving (8%), and original-preserving (3%) contract refinements are rather rare. Original-preserving contract refinements only occur in product line *GPL*. The product line requires one as a method with its contract is introduced in an optional

[5] Our definitions of preservation properties are based on the composition of two contracts. However, in some case the same contract refinement established different properties in different products. We count only preservation properties if they are established by a contract refinement in all products.

Original source code

```
public abstract class ExamDataBase {
/*@ public invariant students!=null &&
  @ 0<threshold && threshold<=maxPoints &&
  @ 0<step && step<=(maxPoints−threshold)/10 &&
  @ (\forall int i; 0<=i && i<students.length && students[i]!=null;
  @   −1<=students[i].points && students[i].points<=maxPoints
  @   && 0<=students[i].bonusPoints
  @   && students[i].bonusPoints<=maxPoints
  @   && (\forall int j; 0<=j && j<students.length
  @   && students[j]!=null && i!=j;
  @   students[i].matrNr!=students[j].matrNr)
  @   && (\forall ExamDataBase ex; ex!=null && ex!=this;
  @     (\forall int k; 0<=k && k<ex.students.length;
  @       ex.students[k]!=students[i])));
  @*/
  [...]
}
public class Student { public int bonusPoints = 0; [...] }
```

ExamDB

```
public abstract class ExamDataBase {
/*@ public invariant students!=null &&
  @ 0<threshold && threshold<=maxPoints &&
  @ 0<step && step<=(maxPoints−threshold)/10 &&
  @ (\forall int i; 0<=i && i<students.length && students[i]!=null;
  @   −1<=students[i].points && students[i].points<=maxPoints
  @   && (\forall int j; 0<=j && j<students.length
  @    && students[j]!=null && i!=j;
  @    students[i].matrNr!=students[j].matrNr)
  @   && (\forall ExamDataBase ex; ex!=null && ex!=this;
  @     (\forall int k; 0<=k && k<ex.students.length;
  @       ex.students[k]!=students[i])));
  @*/
  [...]
}
public class Student { [...] }
```

BonusPoints

```
public abstract class ExamDataBase {
/*@ public invariant
  @ (\forall int i; 0<=i && i<students.length
  @  && students[i]!=null;
  @   0<=students[i].bonusPoints
  @  && students[i].bonusPoints<=maxPoints);
  @*/
  [...]
}
public class Student { public int bonusPoints = 0; }
```

Listing 10. Decomposition of a class invariant for product line *ExamDB*.

feature. Even though the contract refinement establishes the same contract, the contract has to be cloned. Otherwise the method would have no contract in those products that do not contain the optional contract introduction. Those original-preserving contract refinements could be avoided by changing the design of the product line (i.e., introducing a new feature module for the contract introduction). This insight justifies our strategy to specify an existing product line, because an existing product-line design was given, and it could have remained unnoticed with the other two strategies.

Overall, we observe that caller-preserving mechanisms, such as consecutive contract refinement and cumulative contract refinement, are suitable for a quarter of all contract refinements. For all other contract refinements, explicit contract refinement can be used, which does not establish any preservation properties. One may also use a callee-preserving mechanism for some contract refinements, which, however, was not part of our previous discussion. In Fig. 10, we show the percentage of contract refinements that can be expressed with each contract-composition mechanism. We counted contract overriding as applicable to contract refinements if explicit contract refinement was applicable and keyword `original` was not used. Furthermore, we counted mechanisms only as applicable if there was no other mechanism that could be used and avoided more specification clones.

Most contract refinements can be expressed with explicit contract refinement (87% on average), followed by consecutive contract refinement (58%). In comparison, conjunctive contract refinement (32%) and cumulative contract refinement (23%) are seldom

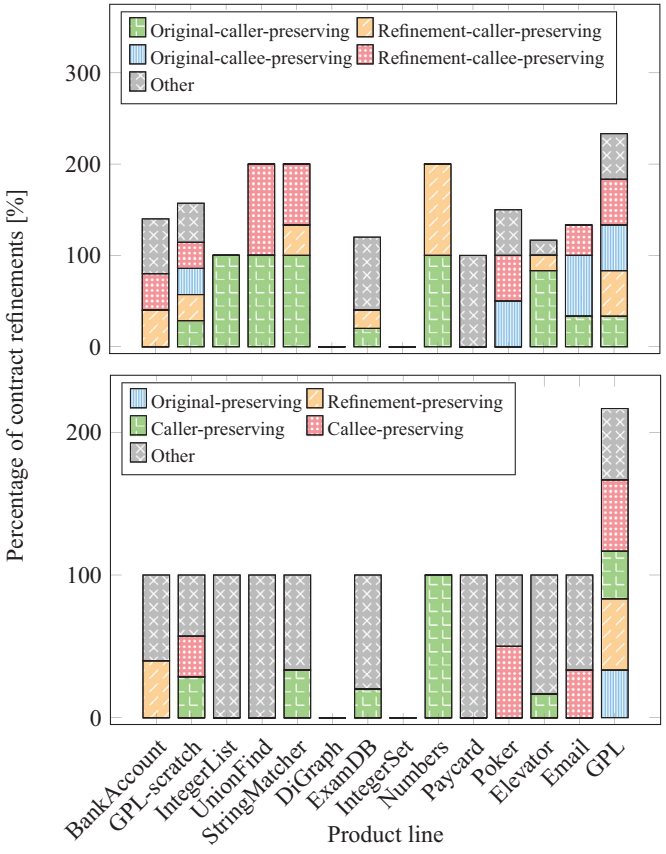

Fig. 9. Preservation properties of contract refinements in all product lines.

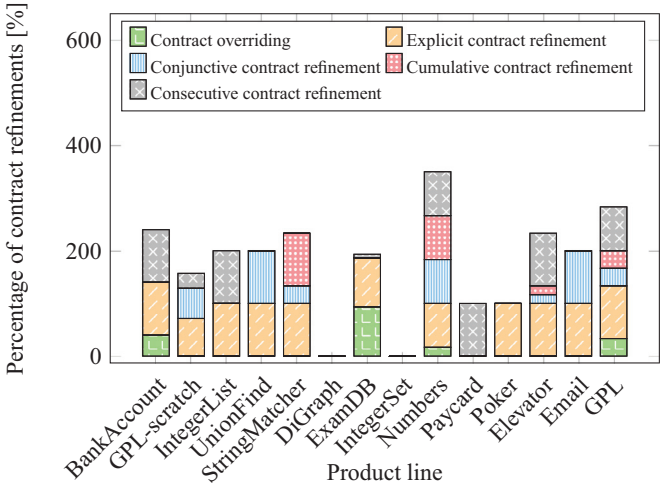

Fig. 10. Applicability of contract-composition mechanisms.

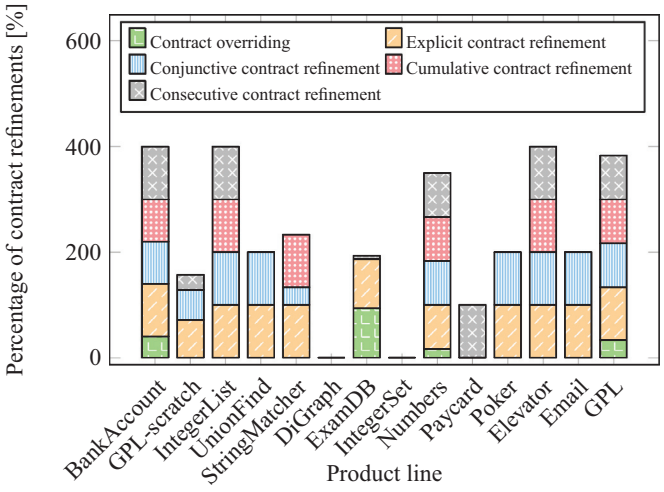

Fig. 11. Applicability of contract-composition mechanisms using *data groups* instead of the frame cut.

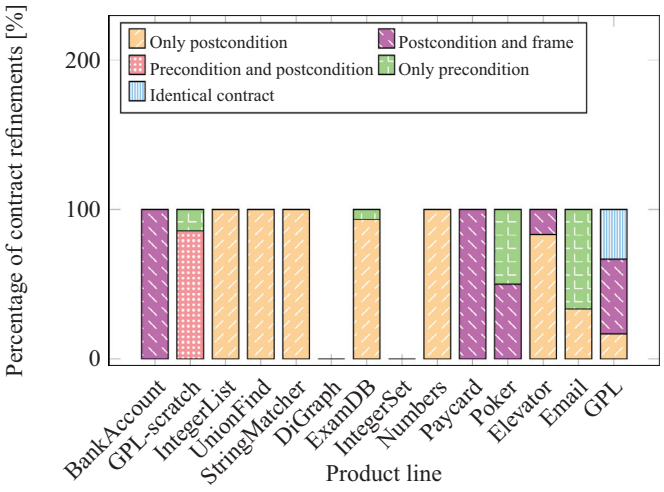

Fig. 12. The granularity of contract refinements in all product lines.

applicable because their frame-cut definitions are seemingly restrictive. In Fig. 11, we depict an alternative approach to the frame cut by relying on *data groups* (Leino, 1998). A data group is a set of locations and has a name that can be used as an alias in the framing condition. Classes can then add new locations to the data group (i.e., the data group hides location details), whereas the framing condition is identical in original and refining contract and, as such, does not need to be refined. With data groups, conjunctive contract refinement is applicable in 57% and cumulative contract refinement is applicable in 45% of all contract refinements.

Only every third contract refinement can be expressed with contract overriding (32%). In particular, explicit contract refinement is applicable in 55% of the contract refinements by taking advantage of keyword `original` and 21% without keyword `original` (i.e., contract overriding). The experience with our case studies confirms our suspicion that contract overriding is superseded by explicit contract overriding. Similarly, there was not a single contract refinement to which cumulative contract refinement is applicable, while consecutive contract refinement is not applicable. Hence, our case studies suggest that contract overriding and cumulative contract refinement are not needed. Each case in which a method refinement does not require a contract refinement can be considered as original-preserving (cf. Fig. 8), and, thus, plain contracting can be applied.

Granularity of Contract Refinement. Addressing **RQ-4**, a further interesting property of contract refinement is granularity. *Granularity* describes whether only a precondition, only a postcondition, only a framing condition, or a combination thereof are refined. In Fig. 12, we give an overview on the granularity for the contract refinements in our subject product lines.[6] Most contract refinements only changed the postcondition while the precondition remained unchanged (60% of contract refinements on average over all product lines). In contrast, sole refinement of a precondition was rather rare (11%). Contract refinements that refine precondition *and* postcondition only occurred in product line *GPL-scratch* and seem to be rather atypical. The likeliest explanation for this anomaly is *GPL-scratch*'s explicit design goal to show the power of contract-composition mechanisms. Interestingly, frame refinement only occurs together with the refinement of the postcondition and, when it occurs, the frame never shrinks. We assume the cause to be that postconditions typically describe what happens to fields newly added to the frame.

The rare occurrence of simultaneous refinement of precondition and postcondition is rather surprising to us and does have consequences that we have not anticipated in our previous discussion: First, when refining only the postcondition conjunctive contract refinement, cumulative contract refinement, and consecutive contract refinement collapse to the same semantics (i.e., postconditions are connected in a conjunction). Second, when only refining a precondition cumulative contract refinement and consecutive contract refinement collapse to the same semantics (i.e., preconditions are connected in a disjunction). Finally, it is relevant how contract-composition mechanisms deal with contract refinements in which either precondition or postcondition is refined. In particular, we found that conjunctive contract refinement can avoid cloning unchanged preconditions compared to cumulative contract refinement or consecutive contract refinement. We give an example of such cloning in Listing 11. The precondition has to be cloned, because non-existent preconditions are treated as `requires true` and the composition with the original precondition by means of a disjunction would invalidate all existing preconditions. However, it seems more reasonable to treat non-existent preconditions and postconditions in contract refinements of all contract-composition mechanisms as `requires original` and `ensures original` instead, which is similar to the treatment of empty contracts.

We noticed that specification clones were especially a problem in those product lines, in which only a part of a precondition or postcondition was refined. These product lines include *GPL-scratch, StringMatcher, ExamDB*, and *Numbers* (cf. Fig. 12). For instance, in product line *ExamDB*, we had to clone and adapt contracts for all contract refinements, because they required fine-granular changes. All discussed mechanisms for contract composition only enable the reuse of complete preconditions and postconditions. Even worse, we had to create many derivative contracts due to the cloning; for four contract refinements in regular feature modules, we had to create 11 additional contract refinements in four derivatives modules. These derivative contracts massively aggravated the cloning. However, we found an elegant solution that avoids all clones and all derivatives based on pure-method refinement. We refer again to Listing 9, which is an excerpt of product line *ExamDB* illustrating the solution. Overall, we introduced two pure methods called from several contracts and refined the pure methods instead of refining each contract explicitly. Alternatively, we could have used model methods, but opted for pure methods to reduce code clones, too. Hence, pure methods and model methods appear to be especially

[6] As mentioned above, product line *GPL* contains identical contracts in contract refinements due to optional contract introductions.

Base

```
public class Client {
  static void mail(Client client, Email msg) { [...] }
  [...]
}
```

Encrypt

```
public class Client {
  /*@ \consecutive_contract
    @ requires msg.isEncrypted() ==>
    @   !unEncryptedMails.contains(msg);
    @ requires !msg.isEncrypted() ==>
    @   !encryptedMails.contains(msg);
    @ requires encryptedMails.contains(msg) ==>
    @   msg.isEncrypted();
    @ ensures msg.isEncrypted() ==>
    @   encryptedMails.contains(msg);
    @ ensures !msg.isEncrypted() ==>
    @   unEncryptedMails.contains(msg); @*/
  static void mail(Client client, Email msg) { [...]
    original(client, msg);
  }
  [...]
}
```

Sign

```
public class Client {
  /*@ requires msg.isEncrypted() ==>
    @   !unEncryptedMails.contains(msg);
    @ requires !msg.isEncrypted() ==>
    @   !encryptedMails.contains(msg);
    @ requires encryptedMails.contains(msg) ==>
    @   msg.isEncrypted();
    @ ensures msg.isSigned() ==>
    @   signedMails.contains(msg); @*/
  static void mail(Client client, Email msg) { [...]
    original(client, msg);
  }
  [...]
}
```

Listing 11. Consecutive contract refinement in product line *Email* requires cloning of preconditions.

suited for fine-grained contract refinements. At the same time, it is possible to reduce accidental contract refinements by using keyword final_method for all pure methods that are not supposed to be refined.

Furthermore, we experienced several specification clones in alternative contract introductions. However, only a small percentage of them actually contain fine-granular differences, which could be avoided by means of pure methods and model methods. The majority of these alternative contract introductions were identical clones. We experienced the relevance of this problem especially in product line *GPL*, in which more than half of the contracts are introduced in alternative features and cloned (cf. Fig. 7). To resolve this issue, one may specify affected contracts in another feature module, which is available whenever one of the alternative features is available. This solution could, of course, require to create a new feature module only for contracts. One may also more systematically investigate how contracts for alternative features should be specified in general, something we only superficially touched on in our previous discussion.

8.3. Threats to Validity

Our case studies and results depend on several threats to validity that we aim to discuss in the following.

The subject product lines we analyzed may not be representative of real-world product lines. First, most of our subjects are significantly smaller than industrial product lines. We tried to overcome this threat by including also larger product lines such as *GPL* with 27 features and 110 contracts. Second, our product lines may not be representative even for small, industrial product lines. To reduce this threat, our subject product lines include several domains, variable algorithms as well as variable data structures, and used three strategies to create product lines with feature-oriented contracts, which should cover all typical application scenarios for the creation of product-line specifications. Furthermore, the subject product lines have been created by several graduate and undergraduate students to reduce subjectivity (cf. Chapter A in Thüm, 2015).

The results of our case studies may depend on the purpose of specifying a product line with contracts. Typically, the main purpose of specifying contracts is static verification. Nevertheless, only the product lines *BankAccount* and *StringMatcher* have been fully verified. All programs with contracts that we decomposed into a product line were said to be verified prior to our decomposition. Hence, it is conceivable they still contain defects with respect to variability, but we do not expect their fixing to heavily impact our statistics on contracts and their refinements. For all remaining product lines, contracts were created for documentation only. However, developing contracts for an existing product-line design is not straightforward, as we had to recover the implicit assumptions that the programmer could have had in mind. Nevertheless, with all three strategies for the creation of subject product lines, we achieved similar results.

A further influence on our results might be that the case studies have been applied between 2010 and 2017, some even before we started to systematically discuss contract-composition mechanisms. That is, the product lines *IntegerList* and an initial version of *BankAccount* have been developed while creating a new contract-composition mechanism that fits the needs of the product line (Scholz et al., 2011; Thüm et al., 2011). However, both product lines have led to similar statistics as other product lines. In particular, by means of manual inspections and validations by several authors, we have retrieved statistics on all product lines at the same time to mitigate threats due to different knowledge on contract composition.

9. Related work

In the following, we discuss the roots of feature-oriented contracts and differences to prior work on contract composition.

Contracts in Feature-Oriented Programming. We proposed a combination of design by contract and feature-oriented programming. Nevertheless, initial ideas have been discussed in the literature long ago. Helm et al. (1990) proposed contracts to model collaborations of classes, although their understanding of contracts has not much in common with contracts as proposed by Meyer (1988). Mezini and Lieberherr (1998) compare adaptive plug and play components, which are similar to feature modules, with those collaborative contracts and rather see both as alternative approaches. Batory et al. (2000) were the first proposing to combine feature modules and contracts, but they have not investigated the combination. Agostinho et al. (2008) and Smaragdakis and Batory (2002) argue that behavioral subtyping does not apply to mixin-like refinements. Our experience with case studies generally supports their claim, but, in most cases, behavioral subtyping is still applicable.

In earlier work, we started to investigate contracts for feature modules with the purpose of product-line verification (Thüm et al., 2011; Scholz et al., 2011). However, the verification techniques

required a contract-composition mechanism to compose feature-oriented contracts into product specifications. Regarding the subject product lines, we relied on cumulative contract refinement (Scholz et al., 2011) and contract overriding (Thüm et al., 2011). With the insights of the previous section, we know that consecutive contract refinement and explicit contract refinement should have been used instead.

Contracts for Mixins. Mulet et al. (1995) were the first to discuss contracts in the context of metaobjects and mixins. They seem to assume that contracts of each metaobject must be established by all mixins, which is similar to plain contracting. Findler and Felleisen (2002) apply contracts to higher-order functions. One such a higher-order function is a mixin, which consumes a class and produces a new class by adding and refining methods. They do not discuss whether mixins may introduce contracts or refine existing ones. Yet, they propose dependent contracts which may solve some problems with specification clones similar to pure-method refinement and model-method refinement. Strickland and Felleisen (2010) extend this work by proposing contract composition for first-class classes (a generalization of mixins and traits) in the functional language Racket. The main difference to our work is a reverse control of contracts: a mixin can decide whether it establishes behavioral subtyping with respect to its superclass, whereas our keywords in feature-oriented contracts specify the expected behavior of later feature modules. Their mechanism for contract composition is similar to explicit contract refinement. Instead of referring to the original precondition and postcondition, they enable the reuse of complete contracts (i.e., even those defined for other methods). Hence, as almost all of our contract refinements only refine the postcondition, their approach would have required to completely clone the precondition in each case. Strickland et al. (2013) provide a formal model, soundness proofs, and an evaluation of contracts for first-class classes. Takikawa et al. (2012) use contracts to implement a static type system for the dynamically-typed language Racket, but do not discuss how contracts are composed for mixins.

Contracts in Delta-Oriented Programming. Contracts have also been proposed for delta-oriented programming, which is similar to feature-oriented programming; delta modules are basically feature modules that can also remove classes and members (Schaefer et al., 2010). Bruns et al. (2011) propose a mechanism analog to contract overriding in feature modules, but also permit the removal of contracts in delta modules. Damiani et al. (2012) define contracts similar to explicit contract refinement. However, their contracts can reference preconditions and postconditions of any other contract by means of uninterpreted assertions. These uninterpreted assertions can be seen as a generalization of keyword `original`, with which only the contract being subject to refinement can be referenced. With the abstract behavioral specification (ABS) language, Hähnle and Schaefer (2012) propose a language that amounts to cumulative contract refinement for delta modules, whereas the composition is technically implemented by means of a restricted form of explicit contract refinement. Hähnle et al. (2013) propose a form of explicit contract refinement that supports to add, modify, and remove preconditions, postconditions, assignable clauses, and specification cases. According to our evaluation results, a single contract-composition mechanism is usually not sufficient and instead several mechanisms should be combined when specifying a product line.

Contracts in Aspect-Oriented Programming. Aspect-oriented programming aims to modularize homogeneous cross-cutting concerns, whereas feature-oriented programming focuses on heterogeneous cross-cutting concerns (Apel et al., 2013a). Nevertheless, the aspect-oriented around advice can be considered equivalent to feature-oriented method refinement (Apel et al., 2008). Aspects have been specified by means of contracts similar to plain contracting (Clifton and Leavens, 2002; Clifton, 2005; Lorenz and Skotiniotis, 2005; Shinotsuka et al., 2006; Molderez and Janssens, 2015), contract overriding (Zhao and Rinard, 2003; Lorenz and Skotiniotis, 2005; Wampler, 2007; Agostinho et al., 2008), explicit contract refinement (Molderez and Janssens, 2015), and conjunctive contract refinement (Klaeren et al., 2001; Clifton and Leavens, 2002; Clifton, 2005; Rebêlo et al., 2013a). Unlike us, some of these approaches (Zhao and Rinard, 2003; Lorenz and Skotiniotis, 2005; Agostinho et al., 2008; Molderez and Janssens, 2015) analyze the contracts of each method in a refinement chain and not only the resulting contract for the last method refinement. In aspect-oriented programming, the considered variability is typically limited to two products: the base application without aspects and the base application with all aspects. As discussed above, all approaches applying contract overriding (Zhao and Rinard, 2003; Lorenz and Skotiniotis, 2005; Wampler, 2007; Agostinho et al., 2008) suffer from the problem of specification clones.

Clifton and Leavens (2002) classify aspects into *observers* and *assistants*. An observer or spectator (Clifton, 2005) is supposed to adhere to contracts of methods it advises, and thus is similar to plain contracting. Specifications of assistants are composed similar to conjunctive contract refinement. In contrast to our definition, the effective assignable clause is the union of all assignable clauses defined for that method and its pieces of advice.

Zhao and Rinard (2003) specify AspectJ programs in Pipa and translate them into Java programs with JML annotations. With Pipa, they introduce two new keywords to JML; keyword `proceeds` indicates whether the advised method is executed or not, and keyword `then` separates the contract of before advice from that of after advice. A difference with respect to invariants is that their invariants defined in an aspect specify the state of this aspect only, whereas invariants in feature modules are added directly to classes.

Lorenz and Skotiniotis (2005) propose to check advice contracts by means of runtime assertion checking. Similar to our mechanisms, they propose three categories of aspects with an according runtime assertion strategy each: *agnostic* and *obedient* similar to plain contracting and *rebellious* similar to contract overriding. The difference between agnostic and obedient is that in an agnostic advice the precondition and postcondition of the original contract are checked before and after the proceed call, respectively. Lorenz and Skotiniotis (2005) discuss blame assignment not only for callers and callees, but for the original method, the advice, and the aspect, whereas for obedient pieces of advice the original method cannot be blamed. In rebellious pieces of advice, preconditions may only be weakened and postconditions only be strengthened, which is also known as the advice substitution principle (Wampler, 2007; Agostinho et al., 2008; Molderez and Janssens, 2012). Wampler (2007) argues that most aspects adhere to the advice substitution principle, which is supported by our empirical investigation on feature-oriented contracts.

Molderez and Janssens (2012) propose ContractAJ, in which aspects have to adhere to the advice substitution principle. However, in follow-up work (Molderez and Janssens, 2015), they adapted ContractAJ to support two contract-composition mechanisms, which are similar to plain contracting and explicit contract refinement. To maintain modular reasoning, they propose to explicitly mention all pieces of advice that do not adhere to the advice substitution principle with keyword `@advisedBy`. Their keyword `proc` is similar to keyword `original` and is actually the first time that previous contracts can be referenced explicitly in aspect-oriented programming. However, all pieces of advice not mentioned in `@advisedBy` clause are oblivious to each other, and thus

they may define a new contract but no other aspect can rely on it. In contrast, when composing contracts with any of our mechanisms, callers can rely on changed contracts.

There are several further approaches combining contracts with aspect-oriented programming (Griswold et al., 2006; Rebêlo et al., 2008a, 2008b; Rebêlo et al., 2011; Rebêlo et al., 2013a; 2013b; Rebêlo et al., 2014). However, these approaches consider contracts as crosscutting concerns that must be modularized by means of aspects, which initiated a controversial discussion on whether aspects are suitable for implementing contracts (Balzer et al., 2006; Rebêlo et al., 2014).

Besides contracts, Katz (2006) proposes to distinguish categories of aspects that have different temporal properties. First, *spectative* pieces of advice establish all temporal properties of the base application except for next state properties. Second, *regulative* pieces of advice establish only safety properties except for next state properties. Roughly speaking, spectative corresponds to plain contracting and regulative is orthogonal to our contract-composition mechanisms. That is, while we focus on input-output behavior only, one could extend our taxonomy and mechanisms to also preserve temporal properties.

Contracts in Object-Oriented Programming. For object-oriented programming, several versions of behavioral subtyping and specification inheritance have been proposed (Meyer, 1988; America, 1991; Findler et al., 2001; Liskov and Wing, 1994; Dhara and Leavens, 1996; Hatcliff et al., 2012), whereas newer versions are usually supposed to completely replace older ones. With cumulative contract refinement and consecutive contract refinement, we discussed two mechanisms being similar to subcontracting as proposed by Meyer (1988) and specification inheritance as proposed by Dhara and Leavens (1996), respectively. For feature-oriented contracts, our case studies revealed that consecutive contract refinement supersedes cumulative contract refinement, which could indicate that specification inheritance supersedes subcontracting, too. The main difference to our work is that we allow programmers to use several mechanisms, because consecutive contract refinement does not apply to all feature-oriented contracts. Similarly, not all subclasses are necessarily behavioral subtypes, which is especially challenging when specifying existing designs posterior; either we cannot specify all properties of superclasses or we must change the design. Thus, having the option to use several contract-composition mechanisms could also be beneficial for object orientation. Interestingly, America (1991) already proposed to distinguish between inheritance for code reuse and actual behavioral subtypes, but unfortunately his idea is not implemented in today's behavioral interface specification languages such as Eiffel, JML, and Spec#. Even if not establishing any of the preservation properties discussed above, conjunctive contract refinement is also applied to object orientation in the runtime assertion checker JMSAssert.[7]

The *open-closed principle* in object-oriented programming states that classes and methods should be open for extension, but closed for modification (Meyer, 1988). As feature-oriented programming is an extension of object-oriented programming, inheritance can be used in feature modules to implement the open-closed principle. However, in our experience, feature-oriented method refinement requires modification rather than extension in some cases. The discussed contract-composition mechanisms are all open for extension using inheritance, but differ to the extent to which modifications using method refinement are possible. While feature-oriented programming is permissive regarding modification, we found contract-composition mechanisms such as consecutive contract refinement are useful to restrict possible modifications. Contract-composition mechanisms can be used to adjust the degree of openness at a fine grain, such that some methods are open for some modifications.

Product-Line Specification with Contracts. Several other researchers proposed product-line specification by means of contracts for various applications, such as feature-model analysis (Bubel et al., 2010; Rhanoui and Asri, 2014), analysis of service-oriented product lines (Lee et al., 2008) and multi product lines (Schröter et al., 2013), as well as test generation for product lines (Bashardoust-Tajali and Corriveau, 2008) and specification of non-functional properties (Rhanoui and Asri, 2014). However, they do not systematically discuss advantages and disadvantages of mechanisms for specifying product lines by means of contracts.

For product-line development, Kästner et al. (2011) distinguish between *open-world view* and *closed-world view*. In a closed-world view, we know all feature modules when reasoning about them, whereas, in an open-world view, feature modules may be added that are unknown. In an open-world view, we need to reason about a set of feature modules without knowing all possible feature modules. In particular, Smaragdakis and Batory (2002) argue that feature-oriented programming is difficult to use in open, collaborative developments. However, the contract-composition mechanisms we have presented (e.g., consecutive contract refinement) may be used to facilitate reasoning for feature-oriented programming even in an open-world view. That is, we can assign contract-composition keywords to certain methods to enable modular reasoning.

Product-Line Verification with Contracts. Several researchers investigated contracts for product-line verification, whereas all approaches are based on theorem proving and model checking (Thüm et al., 2014). Bruns et al. (2011) and Hähnle et al. (2013) propose optimized product-based theorem proving based on KeY. Bruns et al. (2011) reduce the effort for products by means of slicing techniques, whereas Hähnle et al. (2013) split product verification into a first phase with abstract contracts and a second phase with concrete contracts. With proof composition, we presented another technique for proof reuse (Thüm et al., 2011). Instead of searching for reuse potential after product generation, we systematically compose partial proofs defined for each feature. Damiani et al. (2012) discuss feature-product-based theorem proving. The difference to proof composition is that features are verified by means of uninterpreted assertions. Proofs that could not be closed in the feature-based phase, are proven for each product separately. In contrast to proof composition, the product-based phase is likely to require user interaction. Hähnle and Schaefer (2012) propose feature-family-based theorem proving, in which features are verified in isolation and then their valid combinations are verified in a family-based fashion. The advantage of their approach over variability encoding is that part of the verification problem is already solved at feature level.

10. Conclusion

The goal of this work was to systematically investigate how to specify product lines, whereas the focus of our discussion lies on method contracts and feature-oriented programming. We argue that this focus enables the transfer of our results to more abstract specification techniques and other techniques for product-line implementation. In particular, key questions were how to specify contracts in feature modules and how to compose contracts for method refinements.

We introduced and discussed six mechanisms for the composition of preconditions, postconditions, and framing conditions. We

[7] http://www.mmsindia.com/DBCForJava.html.

proposed a taxonomy for contract composition based on the caller view and callee view as well as the original contract and the refining contract. Subsequently, we compared our six mechanisms based on their preservation properties. We further outlined how to extend each contract-composition mechanism for advanced specification concepts. We proved that, in general, not all properties of two contracts can be preserved during composition.

We presented our tool support for contract composition and an empirical evaluation, in which we specified 14 product lines by means of contracts and gained several insights: First, the majority of contracts defined for product lines are not contained in all products (i.e., family-wide specification is not sufficient). Second, product-line specifications can be given by specifying each feature module and usually even without derivative modules (i.e., feature-based specification is sufficient). Third, most but not all method refinements establish behavioral subtyping. Fourth, we identified that four of our six mechanisms were superior to all other mechanisms for certain contract refinements, and thus these four mechanisms should be used in concert. Fifth, fine-granular contract refinements and alternative method introductions often cause specification clones. Finally, most contract refinements only refine the postcondition while the precondition and framing condition remain unchanged. In particular, only eleven out of sixty contract refinements modified the frame.

Acknowledgements

We gratefully acknowledge fruitful discussions on feature-oriented contracts with Fabian Benduhn, Jens Meinicke, André Weigelt, Matthias Praast, Florian Proksch, Gunter Saake, Sven Apel, Don Batory, Reiner Hähnle, and Martin Kuhlemann. Work on this paper was partially supported by the DFG Research Unit *Controlling Concurrent Change*, funding number FOR 1800, project FE407/17-2. This work was additionally partially supported by the DFG through its Collaborative Research Center CROSSING.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.jss.2019.01.044.

References

Abrial, J.-R., 2010. Modeling in Event-B: System and Software Engineering, 1st. ISBN 0521895561, 9780521895569.

Agostinho, S., Moreira, A., Guerreiro, P., 2008. Contracts for Aspect-Oriented Design, pp. 1:1–1:6. doi:10.1145/1408647.1408648, ISBN 978-1-60558-144-6.

America, P., 1991. Designing an Object-Oriented Programming Language with Behavioural Subtyping, pp. 60–90. ISBN 3-540-53931-X.

Apel, S., Batory, D., Kästner, C., Saake, G., 2013a. Feature-Oriented Software Product Lines: Concepts and Implementation.

Apel, S., Hutchins, D., 2010. A Calculus for Uniform Feature Composition. ACM Transactions on Programming Languages and Systems (TOPLAS) 32 (5), 19:1–19:33. doi:10.1145/1745312.1745316, ISSN 0164-0925.

Apel, S., Kästner, C., Lengauer, C., 2013b. Language-Independent and Automated Software Composition: The FeatureHouse Experience 39 (1), 63–79. doi:10.1109/TSE.2011.120, ISSN 0098-5589.

Apel, S., Leich, T., Rosenmüller, M., Saake, G., 2005. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In: Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE), pp. 125–140.

Apel, S., Leich, T., Saake, G., 2008. Aspectual Feature Modules 34 (2), 162–180. http://doi.ieeecomputersociety.org/10.1109/TSE.2007.70770.

Apel, S., Lengauer, C., Möller, B., Kästner, C., 2010. An Algebraic Foundation for Automatic Feature-Based Program Synthesis. Sci. Comput. Program. 75 (11), 1022–1047.

Apel, S., von Rhein, A., Thüm, T., Kästner, C., 2013c. Feature-Interaction Detection Based on Feature-Based Specifications 57 (12), 2399–2409. doi:10.1016/j.comnet.2013.02.025, ISSN 1389-1286.

Balzer, S., Eugster, P.T., Meyer, B., 2006. Can Aspects Implement Contracts?, pp. 145–157. doi:10.1007/11751113_11, ISBN 3-540-34063-7, 978-3-540-34063-8.

Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H., 2011. Specification and Verification: The Spec# Experience. Communications of the ACM 54, 81–91. doi:10.1145/1953122.1953145, ISSN 0001-0782.

Bashardoust-Tajali, S., Corriveau, J.-P., 2008. On Extracting Tests from a Testable Model in the Context of Domain Engineering. In: Proceedings of the International Conference on Engineering of Complex Computer Systems (ICECCS), pp. 98–107. doi:10.1109/ICECCS.2008.17.

Batory, D., 2006. A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite. In: Proceedings of the summer school on Generative and Transformational Techniques in Software Engineering (GTTSE), pp. 3–35.

Batory, D., Cardone, R., Smaragdakis, Y., 2000. Object-Oriented Frameworks and Product Lines. In: Proceedings of the International Software Product Line Conference (SPLC). Kluwer Academic Publishers, Norwell, MA, USA, pp. 227–247. ISBN 0-79237-940-3.

Batory, D., Sarvela, J.N., Rauschmayer, A., 2004. Scaling Step-Wise Refinement 30 (6), 355–371.

Beckert, B., Hähnle, R., Schmitt, P., 2007. Verification of Object-Oriented Software: The KeY Approach.

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

Benduhn, F., Thüm, T., Lochau, M., Leich, T., Saake, G., 2015. A survey on modeling techniques for formal behavioral verification of software product lines. In: Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems. ACM, p. 80.

Beohar, H., Varshosaz, M., Mousavi, M.R., 2016. Basic Behavioral Models for Software Product Lines. Sci. Comput. Program. 123 (C), 42–60. doi:10.1016/j.scico.2015.06.005, ISSN 0167-6423.

Bolle, S., 2017. Feature-orientiertes Framing für die Verifikation von Software-Produktlinien. Braunschweig Master's thesis doi:10.24355/dbbs.084-201711280920. In German.

Börger, E., Stark, R.F., 2003. Abstract State Machines: A Method for High-Level System Design and Analysis ISBN 3540007024.

Bracha, G., Cook, W., 1990. Mixin-Based Inheritance. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), pp. 303–311. doi:10.1145/97945.97982, ISBN 0-89791-411-2.

Bruns, D., Klebanov, V., Schaefer, I., 2011. Verification of Software Product Lines with Delta-Oriented Slicing, pp. 61–75. ISBN 3-642-18069-8, 978-3-642-18069-9.

Bubel, R., Din, C., Hähnle, R., 2010. Verification of Variable Software: An Experience Report. Technical Report 2010-13, Department of Informatics, Karlsruhe Institute of Technology, Karlsruhe, Germany.

Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J., Leavens, G.T., Leino, K.R.M., Poll, E., 2005. An Overview of JML Tools and Applications 7 (3), 212–232. doi:10.1007/s10009-004-0167-4, ISSN 1433-2779.

Calder, M., Kolberg, M., Magill, E.H., Reiff-Marganiec, S., 2003. Feature Interaction: A Critical Review and Considered Forecast 41 (1), 115–141.

Carmo Machado, I.D., McGregor, J.D., Cavalcanti, Y.a. C., De Almeida, E.S., 2014. On Strategies for Testing Software Product Lines: A Systematic Literature Review 56 (10), 1183–1199. doi:10.1016/j.infsof.2014.04.002, ISSN 0950-5849.

Chalin, P., Kiniry, J., Leavens, G.T., Poll, E., 2005. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2, pp. 342–363.

Clements, P., Northrop, L., 2001. Software Product Lines: Practices and Patterns. Addison-Wesley, Boston, MA, USA ISBN 0-201-70332-7.

Clifton, C., 2005. A Design Discipline and Language Features for Modular Reasoning in Aspect-Oriented Programs. Iowa State University, Ames, IA, USA Ph.D. thesis.

Clifton, C., Leavens, G.T., 2002. Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning. In: Proceedings of the Workshop on Foundations of Aspect-Oriented Languages (FOAL). Iowa State University, Ames, IA, USA, pp. 33–44.

Czarnecki, K., Eisenecker, U., 2000. Generative Programming: Methods, Tools, and Applications.

Damiani, F., Owe, O., Dovland, J., Schaefer, I., Johnsen, E.B., Yu, I.C., 2012. A Transformational Proof System for Delta-Oriented Programming, pp. 53–60. doi:10.1145/2364412.2364422, ISBN 978-1-4503-1095-6.

Dhara, K.K., Leavens, G.T., 1996. Forcing Behavioral Subtyping through Specification Inheritance. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 258–267. ISBN 0-8186-7246-3.

Farrell, M., Monahan, R., Power, J.F., 2017. Specification clones: An empirical study of the structure of event-b specifications. In: International Conference on Software Engineering and Formal Methods. Springer, pp. 152–167. ISBN 1-58113-487-8.

Findler, R.B., Felleisen, M., 2002. Contracts for Higher-Order Functions. In: Proceedings of the International Conference on Functional Programming (ICFP), pp. 48–59. doi:10.1145/581478.581484, ISBN 1-58113-487-8.

Findler, R.B., Latendresse, M., Felleisen, M., 2001. Behavioral Contracts and Behavioral Subtyping. In: Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESECFSE), pp. 229–236. doi:10.1145/503209.503240, ISBN 1-58113-390-1.

Gosling, J., Joy, B., Steele, G., Bracha, G., 2005. The Java Language Specification, Third Edition. Addison-Wesley, Amsterdam.

Griswold, W.G., Sullivan, K., Song, Y., Shonle, M., Tewari, N., Cai, Y., Rajan, H., 2006. Modular Software Design with Crosscutting Interfaces 23 (1), 51–60. doi:10.1109/MS.2006.24, ISSN 0740-7459.

Hähnle, R., Schaefer, I., 2012. A Liskov Principle for Delta-Oriented Programming, pp. 32–46. doi:10.1007/978-3-642-34026-0_4, ISBN 978-3-642-34025-3 .

Hähnle, R., Schaefer, I., Bubel, R., 2013. Reuse in Software Verification by Abstract Method Calls, pp. 300–314. doi:10.1007/978-3-642-38574-2_21, ISBN 978-3-642-38573-5.

Harrison, W., Ossher, H., 1993. Subject-Oriented Programming: A Critique of Pure Objects. In: Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), pp. 411–428. doi:10.1145/165854.165932, ISBN 0-89791-587-9.

Hatcliff, J., Leavens, G.T., Leino, K.R.M., Müller, P., Parkinson, M., 2012. Behavioral Interface Specification Languages 44 (3), 16:1–16:58. doi:10.1145/2187671.2187678, ISSN 0360-0300.

Helm, R., Holland, I.M., Gangopadhyay, D., 1990. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), pp. 169–180. doi:10.1145/97945.97967, ISBN 0-89791-411-2.

Höfner, P., Möller, B., 2009. An Extension for Feature Algebra, pp. 75–80. doi:10.1145/1629716.1629731, ISBN 978-1-60558-567-3.

Höfner, P., Möller, B., Zelend, A., 2012. Foundations of Coloring Algebra with Consequences for Feature-Oriented Programming, pp. 33–49. doi:10.1007/978-3-642-33314-9_3, ISBN 978-3-642-33313-2.

Kästner, C., Apel, S., Kuhlemann, M., 2009a. A Model of Refactoring Physically and Virtually Separated Features. In: Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE), pp. 157–166. doi:10.1145/1621607.1621632, ISBN 978-1-60558-494-2.

Kästner, C., Apel, S., Ostermann, K., 2011. The Road to Feature Modularity?, pp. 5:1–5:8. doi:10.1145/2019136.2019142, ISBN 978-1-4503-0789-5.

Kästner, C., Thüm, T., Saake, G., Feigenspan, J., Leich, T., Wielgorz, F., Apel, S., 2009b. FeatureIDE: A Tool Framework for Feature-Oriented Software Development. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 611–614. Formal demonstration paper

Katz, S., 2006. Aspect Categories and Classes of Temporal Properties 1, 106–134. doi:10.1007/11687061_4.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G., 2001. An Overview of AspectJ. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), pp. 327–354. doi:10.1007/3-540-45337-7_18, ISBN 3-540-42206-4.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J., 1997. Aspect-Oriented Programming. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), pp. 220–242.

Klaeren, H., Pulvermueller, E., Rashid, A., Speck, A., 2001. Aspect Composition Applying the Design by Contract Principle. In: Proceedings of the International Conference on Generative and Component-Based Software Engineering (GCSE), pp. 57–69. ISBN 3-540-42578-0.

Leavens, G. T., Cheon, Y., 2006. Design by Contract with JML.

Leavens, G.T., Müller, P., 2007. Information Hiding and Visibility in Interface Specifications. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 385–395. doi:10.1109/ICSE.2007.44, ISBN 0-7695-2828-7.

Leavens, G. T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D. M., Dietl, W., 2013. JML Reference Manual.

Lee, J., Muthig, D., Naab, M., 2008. An Approach for Developing Service Oriented Product Lines. In: Proceedings of the International Software Product Line Conference (SPLC), pp. 275–284. doi:10.1109/SPLC.2008.34.

Leino, K.R.M., 1998. Data Groups: Specifying the Modification of Extended State. In: Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), pp. 144–153. doi:10.1145/286936.286953, ISBN 1-58113-005-8.

Liskov, B.H., Wing, J.M., 1994. A Behavioral Notion of Subtyping. ACM Transactions on Programming Languages and Systems (TOPLAS) 16 (6), 1811–1841. doi:10.1145/197320.197383, ISSN 0164-0925.

Liu, J., Batory, D., Lengauer, C., 2006. Feature Oriented Refactoring of Legacy Applications. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 112–121. doi:10.1145/1134285.1134303, ISBN 1-59593-375-1.

Lorenz, D.H., Skotiniotis, T., 2005. Extending Design by Contract for Aspect-Oriented Programming abs/cs/0501070.

Meinicke, J., Thüm, T., Schröter, R., Benduhn, F., Leich, T., Saake, G., 2017. Mastering software variability with featureide.

Meyer, B., 1988. Object-Oriented Software Construction, 1st Prentice-Hall, Inc., Upper Saddle River, NJ, USA ISBN 0136290493.

Meyer, B., 1992. Applying Design by Contract 25 (10), 40–51.

Mezini, M., Lieberherr, K., 1998. Adaptive Plug-and-Play Components for Evolutionary Software Development. In: Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), pp. 97–116. doi:10.1145/286936.286950, ISBN 1-58113-005-8.

Molderez, T., Janssens, D., 2012. Design by Contract for Aspects, by Aspects. In: Proceedings of the Workshop on Foundations of Aspect-Oriented Languages (FOAL), pp. 9–14. doi:10.1145/2162010.2162015, ISBN 978-1-4503-1099-4.

Molderez, T., Janssens, D., 2015. Modular Reasoning in Aspect-Oriented Languages from a Substitution Perspective, pp. 3–59. doi:10.1007/978-3-662-46734-3_1.

Mulet, P., Malenfant, J., Cointe, P., 1995. Towards a Methodology for Explicit Composition of MetaObjects. In: Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), pp. 316–330. doi:10.1145/217838.217870, ISBN 0-89791-703-0.

Pohl, K., Böckle, G., van Der Linden, F.J., 2005a. Software product line engineering: foundations, principles and techniques. Springer Science & Business Media.

Pohl, K., Böckle, G., van der Linden, F.J., 2005b. Software Product Line Engineering: Foundations, Principles and Techniques.

Prehofer, C., 1997. Feature-Oriented Programming: A Fresh Look at Objects. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), pp. 419–443.

Proksch, F., Krüger, S., 2014. Tool Support for Contracts in FeatureIDE. Technical Report FIN-001-2014. University of Magdeburg, Germany.

Rebêlo, H., Coelho, R., Lima, R., Leavens, G.T., Huisman, M., Mota, A., Castor Filho, F., 2011. On the Interplay of Exception Handling and Design by Contract: An Aspect-Oriented Recovery Approach, pp. 7:1–7:6. doi:10.1145/2076674.2076681, ISBN 978-1-4503-0893-9.

Rebêlo, H., Leavens, G.T., Bagherzadeh, M., Rajan, H., Lima, R., Zimmerman, D.M., Cornélio, M., Thüm, T., 2014. AspectJML: Modular Specification and Runtime Checking for Crosscutting Contracts. In: Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD), pp. 157–168. doi:10.1145/2577080.2577084, ISBN 978-1-4503-2772-5.

Rebêlo, H., Leavens, G.T., Lima, R., Borba, P., Ribeiro, M., 2013a. Modular Aspect-Oriented Design Rule Enforcement with XPIDRs. In: Proceedings of the Workshop on Foundations of Aspect-Oriented Languages (FOAL), pp. 13–18. doi:10.1145/2451598.2451603, ISBN 978-1-4503-1865-5.

Rebêlo, H., Lima, R., Cornélio, M., Soares, S., 2008. A JML Compiler Based on AspectJ, pp. 541–544. doi:10.1109/ICST.2008.14, ISBN 978-0-7695-3127-4.

Rebêlo, H., Lima, R., Kulesza, U., Ribeiro, M., Cai, Y., Coelho, R., Sant'Anna, C., Mota, A., 2013b. Quantifying the Effects of Aspectual Decompositions on Design by Contract Modularization: A Maintenance Study 23 (7), 913–942. doi:10.1142/S0218194013500265.

Rebêlo, H., Soares, S., Lima, R., Borba, P., Cornélio, M., 2008. JML and Aspects: The Benefits of Instrumenting JML Features with AspectJ. University of Central Florida, Orlando, Florida, USA, pp. 11–18.

Rhanoui, M., Asri, B.E., 2014. A Contractual Specification of Functional and Non–Functional Requirements of Domain-Specific Components 11 (2), 172–181 ISSN 1694-0814.

Robby, Rodríguez, E., Dwyer, M.B., Hatcliff, J., 2006. Checking JML Specifications Using an Extensible Software Model Checking Framework 8 (3), 280–299. doi:10.1007/s10009-005-0218-5, ISSN 1433-2779.

Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N., 2010. Delta-Oriented Programming of Software Product Lines. In: Proceedings of the International Software Product Line Conference (SPLC), pp. 77–91.

Schaefer, I., Bettini, L., Damiani, F., 2011. Compositional Type-Checking for Delta-Oriented Programming. In: Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD), pp. 43–56. doi:10.1145/1960275.1960283, ISBN 978-1-4503-0605-8.

Scholz, W., Thüm, T., Apel, S., Lengauer, C., 2011. Automatic Detection of Feature Interactions Using the Java Modeling Language: An Experience Report, pp. 7:1–7:8. doi:10.1145/2019136.2019144, ISBN 978-1-4503-0789-5.

Schröter, R., Siegmund, N., Thüm, T., 2013. Towards Modular Analysis of Multi Product Lines, pp. 96–99. doi:10.1145/2499777.2500719, ISBN 978-1-4503-2325-3.

Shinotsuka, S., Ubayashi, N., Shinomi, H., Tamai, T., 2006. An Extensible Contract Verifier for AspectJ, pp. 1:1–1:6.

Smaragdakis, Y., Batory, D., 2002. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs 11 (2), 215–255. doi:10.1145/505145.505148, ISSN 1049-331X.

Strickland, T.S., Dimoulas, C., Takikawa, A., Felleisen, M., 2013. Contracts for First-Class Classes. ACM Transactions on Programming Languages and Systems (TOPLAS) 35 (3), 11:1–11:58. doi:10.1145/2518189, ISSN 0164-0925.

Strickland, T.S., Felleisen, M., 2010. Contracts for First-Class Classes, pp. 97–112. doi:10.1145/1869631.1869642, ISBN 978-1-4503-0405-4.

Takikawa, A., Strickland, T.S., Dimoulas, C., Tobin-Hochstadt, S., Felleisen, M., 2012. Gradual Typing for First-Class Classes. In: Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), pp. 793–810. doi:10.1145/2384616.2384674, ISBN 978-1-4503-1561-6.

Tarr, P., Ossher, H., Harrison, W., Sutton Jr., S.M., 1999. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 107–119. doi:10.1145/302405.302457, ISBN 1-58113-074-0.

Thüm, T., 2015. Product-Line Specification and Verification with Feature-Oriented Contracts. Otto von Guericke University Magdeburg Ph.D. thesis.

Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G., 2014. A Classification and Survey of Analysis Strategies for Software Product Lines 47 (1), 6:1–6:45. doi:10.1145/2580950, ISSN 0360-0300.

Thüm, T., Apel, S., Zelend, A., Schröter, R., Möller, B., 2013. Subclack: Feature-Oriented Programming with Behavioral Feature Interfaces, pp. 1–8. doi:10.1145/2489828.2489829, ISBN 978-1-4503-2046-7.

Thüm, T., Meinicke, J., Benduhn, F., Hentschel, M., von Rhein, A., Saake, G., 2014. Potential Synergies of Theorem Proving and Model Checking for Software Product Lines. In: Proceedings of the International Software Product Line Conference (SPLC), pp. 177–186. doi:10.1145/2648511.2648530, ISBN 978-1-4503-2740-4.

Thüm, T., Schaefer, I., Apel, S., Hentschel, M., 2012. Family-Based Deductive Verification of Software Product Lines. In: Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE), pp. 11–20. doi:10.1145/2371401.2371404, ISBN 978-1-4503-1129-8.

Thüm, T., Schaefer, I., Kuhlemann, M., Apel, S., 2011. Proof Composition for Deductive Verification of Software Product Lines, pp. 270–277. doi:10.1109/ICSTW.2011.48.

Thüm, T., Schaefer, I., Kuhlemann, M., Apel, S., Saake, G., 2012. Applying Design by Contract to Feature-Oriented Programming. In: Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE), pp. 255–269. doi:10.1007/978-3-642-28872-2_18, ISBN 978-3-642-28871-5.

Wampler, D., 2007. Aspect-Oriented Design Principles: Lessons from Object-Oriented Design. In: Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD), pp. I6:1–I6:10.
Weigelt, A., 2013. Methoden-basierte Komposition von Kontrakten in Feature-orientierter Programmierung. University of Magdeburg. Germany Bachelor's thesis. In German
Weiß, B., 2011. Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic, and Predicate Abstraction. Karlsruhe Institute of Technology, Germany Ph.D. thesis.
Zhao, J., Rinard, M.C., 2003. Pipa: A Behavioral Interface Specification Language for AspectJ. In: Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE), pp. 150–165. ISBN 3-540-00899-3.