University of Magdeburg

School of Computer Science

Bachelor's Thesis

Contract-Aware Feature Composition

Author:

Fabian Benduhn

October 01, 2012

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake
Dipl.-Inform. Thomas Thüm

Department of Technical & Business Information Systems

Benduhn, Fabian:
Contract-Aware Feature Composition
Bachelor’s Thesis, University of Magdeburg, 2012.

Abstract

Feature-oriented programming is a paradigm to develop software product lines, in which products can be generated by composing feature modules. Design by contract is a development methodology, in which methods are specified by annotating them with contracts. Researchers propose to use such contracts to specify feature modules in order to perform different product-line analysis-techniques such as verification and automated detection of feature interactions.

We have found six approaches to specify feature modules with contracts in literature. These approaches differ in the way specifications can be refined. Each of them relies on a certain way in which contracts are composed. So far, these contract composition mechanisms must be applied manually for each product. Thus, the evaluation of applications that rely on contracts in feature-oriented programming is laborious, error-prone, or not feasible at all.

We formalize the underlying composition mechanism for each existing approach considering possible interpretations, variations, and combinations. Based on our formalization, we provide tool support that can be used to specify feature modules with contracts expressed in the Java Modeling Language. Our tool is integrated into an integrated development environment that provides support for the whole product-line engineering process. Furthermore, it allows the developer to choose the desired contract composition mechanism for each product line. We performed four case studies to evaluate the applicability of different contract composition mechanisms.

Acknowledgments

I would like to thank my advisors Thomas Thüm and Prof. Gunter Saake for giving me the opportunity to write this thesis. Without the outstanding support of Thomas, this thesis would not have been possible for many reasons. I thank for all his advice, for all his effort, for endless hours of discussion, for all the patience when listening to my ideas and questions, and especially for all his trust in me and my capabilities.

I thank Dr. Ina Schaefer for valuable discussion and everyone else who commented on my thesis, such as the participants of the FOSD-Meeting, 2012 in Braunschweig.

For their support with FeatureHouse, I would like to thank Dr. Sven Apel, Dr. Christian Kästner, and Jörg Liebig.

Furthermore, I thank Martin Hentschel for discussion and insights about semantics of the Java Modeling Language.

Finally, I thank my family and friends, especially my girlfriend Katharina, for the moral support.

Contents

List of Figures

List of Tables

1. Introduction

How can we build software that works? According to Parnas [64] there are two approaches we can follow in order to achieve this goal.

The first is to use mathematical methods to reason about the correctness of software. This approach is being researched in the field of formal methods. The idea to use the computer to reason about its own programs was originally stated by McCarthy in 1962 [53]. In 1967, Floyd refined this idea by proposing a verifying compiler that automatically checks the correctness of a program [25]. There has been a lot of progress in formal methods and they have successfully been used in domains such as traffic control [33], microprocessors [26] [57], electronic cash systems [72] [83], flight control [13], and for other safety-critical systems such as the Maeslant Kering (a movable barrier protecting the port of Rotterdam from flooding) [78]. However, the use of formal methods in industry is still more exception than rule [82]. The task of constructing a verifying compiler has turned out to be very challenging and has been restated by Hoare in 2003 as a grand challenge for the software engineering community [31].

Formal methods require developers to write a formal specification of the software to be build. Developers have to consider every aspect of the software intensively in order to express requirements of the software precisely. Experience in industrial projects suggest that specifying a software system formally provides valuable insights that prevent errors from being propagated into late development phases [41] [82] [41].

The second approach to develop correct software is to structure it in a manner that allows programmers to build and understand software systems as an aggregation of smaller program units. The systematic development of programs as a set of modules allows the developer to understand each module in isolation [62] [21]. Progress in this area of software engineering

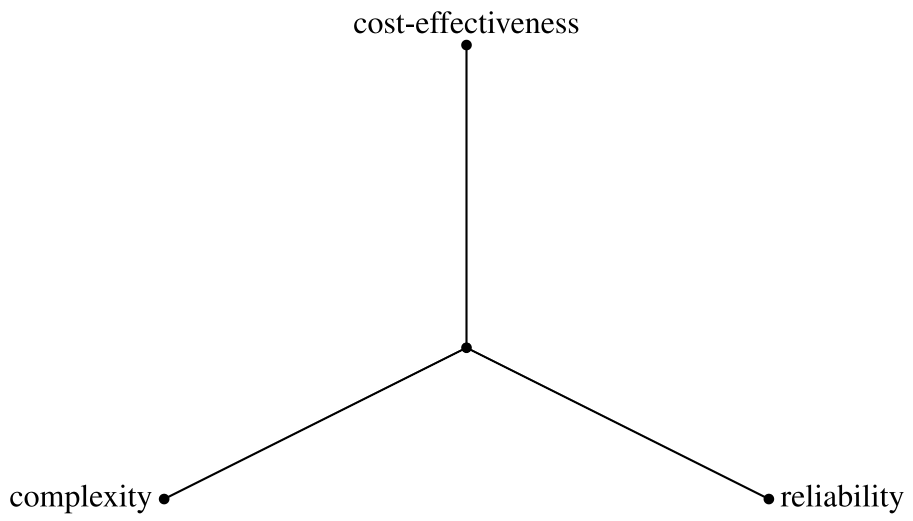


Figure 1.1: The interdependent relationship of cost-effectiveness, complexity and reliability in software development

has manifested in several programming paradigms, methodologies and languages that provide different kinds of modules and composition. Object-oriented programming has emerged as a wide-spread paradigm in which software is designed and built as systems of communicating objects [55].

Researchers in the area of software engineering are not only concerned with correctness but also need to find ways to build complex software systems cost-efficiently. It is especially challenging to achieve all of these desired properties simultaneously because they are interdependent. Figure 1.1 illustrates the relationship between complexity, reliability and cost-effectiveness. When software developers want to improve one of these properties, at least one of the other properties is effected negatively. Increasing the complexity of software usually results in increased costs and/or a reduced reliability. Similarly, the decision to increase reliability results in increased costs and/or a reduced complexity (e.g., by reducing functionality).

One of the fundamental ideas in software engineering to reduce development costs is to take advantage of commonalities between software artifacts by reusing certain parts of software. A significant part of the success of object-oriented programming can be traced back to the means of reuse it provides [47]. A method can be called from multiple callers, and is therefore a simple means of reuse. Inheritance provides a mechanism of reuse inside class hierarchies. The idea to build software systems from a set of reusable components is as old as software engineering as a field [54]. However, there is a trade-off between usability and re-usability of such software components. General components such as software libraries and off-the-shelf tools (e.g., database management systems) provide high reusability and are commonly used ways to achieve a form of reuse. However, the price for their re-usability is a rather low usability (i.e., components usually need to be adapted to solve a given problem).

Software product lines (also called program families) aim to solve this trade-off by taking advantage of commonalities between a set of products inside a specific domain [20] [66]. This provides a high rate of reuse for highly specialized components. The vision of software product line engineering is to achieve a level of mass-customization as known from other industries, such as automobiles. Different variants of a product can be assembled from a set of core assets providing a high grade of variability. Each variant can be distinguished in terms of features, where a feature describes a property of interest for a stakeholder that also serves as a configuration option [67]. Software product lines have been successfully applied in practice and proven to be a cost-effective way to produce complex software systems—by developing multiple related products simultaneously [81].

Feature-oriented programming (FOP) is a paradigm in which programs are assembled from a set of feature modules [67]. A feature module encapsulates all classes and class fragments that belong to a particular feature. While feature-oriented programming emerges as a promising paradigm to develop software product lines efficiently, the aspect of reliability moves into the focus of attention by academia and industry [3] [10] [23] [35]. Mission-critical and safety-critical applications as they are needed in domains such as automobiles, avionics or embedded systems in general often have hard limits on resources which makes mass-customization of software even more desirable. Furthermore, these applications require a high grade of reliability.

Meyer proposes design by contract as a design principle for the development of object-oriented programs, which aims to increase reliability of software. [56]. He introduces the notion of method contracts as a means to specify the behavior of programs. The callers of a method guarantees that a certain precondition holds when calling the method. In return, the method guarantees to fulfill a certain postcondition after execution. The specification in common design by contract languages is expressed in the style of the implementation language, which allows developers without rigorous training in formal methods to achieve a high reliability. Meyer argues that the notion of contracts, which are a form of formal specification, allow modular reasoning and can also be used as a design guideline (i.e., methods do not have to check the preconditions), which leads to a more efficient implementation. Furthermore, contracts are executable instructions that can be used as input for several verification tools [16].

As we have stated above, feature-oriented programming can be used for efficient development of software product lines and design by contract can be used to achieve a high grade of reliability. Recently, Thüm et al. suggested to apply design by contract to feature-oriented programming [77]. We believe that the combination of these two approaches may result in a software development methodology that helps to develop reliable software systems cost-effectively. Furthermore, we expect that the combination of design by contract and feature-oriented programming can be fruitful for both research communities. On the one hand, the acceptance of formal methods might be increased because their costs are reduced when they are applied in a product-line fashion. It is possible to specify each feature in a product line separately and to compose the specification of each product automatically. On the other hand, several FOP-specific problems such as automated detection of feature interactions can be solved using design by contract [69]. Furthermore, verification approaches that take the variability in the product line into account can be investigated [73].

Traditional design by contract as it was introduced for object-oriented programming cannot be applied in feature-oriented programming directly, because in feature-oriented programming contracts may be refined [77]. Existing tools for feature-oriented programming allow features to introduce classes, methods, and fields as well as to refine existing ones [11] [5] [6]. When introducing methods, they can be enriched with contracts, but it is not clear how contracts should be defined when methods are refined. Thüm et al. propose five different approaches to specify features with contracts expressed in the Java Modeling Language (JML): plain contracting, contract overriding, explicit contract refinement, consecutive contract refinement, and pure method refinement [77]. Another approach, cumulative contract refinement, is used for the automated detection of feature interactions [69]. Each approach relies on a specific way in which contracts are composed. So far the composition of contracts had to be performed manually, due to lack of tool support.

Goal of this Thesis

The goal of this thesis is to provide tool support for design by contract in feature-oriented programming that supports the necessary contract composition mechanisms for these approaches. As a first step, we identify and formalize the required contract composition mechanisms that must be used during feature composition for each existing approach. We also consider possible variations and combinations of these approaches.

Based on the results of our formalization, we provide a tool that supports design by contract in feature-oriented programming. Our tool is based on FeatureHouse [5], a framework for the composition of software artifacts. Furthermore, it is integrated into FeatureIDE [74], an integrated development environment for feature-oriented software development. Our tool allows the developer to decide which specification approach to use, by choosing the appropriate composition mechanism. We have validated our definitions of contract composition mechanism by using our tool on existing case studies, in which contract composition had to be performed manually. Furthermore, we have used our tool support to perform four case studies, in which we have compared the applicability of the different contract composition mechanisms in practice.

Structure of this Thesis

We give an overview of the necessary background to understand this thesis and its context in Chapter 2. Chapter 3 includes our definitions of contract specification approaches including a formalization of the underlying contract composition mechanisms and implementation decisions specific to the Java Modeling Language. In Chapter 4, we present our tool support for design by contract in feature-oriented programming that is based on the results of the previous chapters. Chapter 5 covers the evaluation of our tool support and contract composition mechanisms. We give an overview of related work in Chapter 6 and conclude in Chapter 7. In Chapter 8, we sketch future work.

2. Background

This chapter introduces the underlying concepts used in this thesis. Section 2.1 gives a definition of software product lines and Section 2.2 introduces feature-oriented programming as an implementation technique for software product lines. We give an overview about method contracts in Section 2.3.

2.1 Software Product Lines

Software product lines are families of related software products [20] [66]. Northrop defines a software product line as "a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [20]. A feature can be any property of interest for stakeholders. The members of a software product line are called products (variants) and can be distinguished in terms of features. Each product can be described by a set of features [34].

Software product lines can be used in any application area of software development that requires the development of similar products sharing certain features. A typical example for a domain in which software product lines can be applied are embedded systems [14]. To exemplify the notion of software product lines we will consider the domain of franking machines.[1] Franking machines are devices that are used to process mail automatically in order to weigh, frank, and seal it. In order to fit different customer requirements franking

[1]While franking machines are real devices, the example domain presented in this thesis is a simplified, merely fictional notion to exemplify the discussed concepts.

machines are typically offered in many different variants: From simple machines that can only process one envelope at a time, to machines that are able to process large stacks of mail in different sizes and weights. Some machines offer extended features such as Internet connection to download postal tariff changes and software updates. Furthermore, franking machines are usually sold to customers from different countries, which results in different user-interface languages, currencies, stamps, and tariff systems.

Software for embedded devices often has to be developed in different variants to suit different customer needs [14]. In theory, it is possible to develop one program that can be used for all different franking machine variants. However, resources like memory and processing power are limited and relative expensive for embedded devices such as the franking machines. Therefore, we want to create a software product that contains exactly the necessary functionality for each type of franking machine. It is possible to develop and maintain each variant of the software independently. Traditionally, version control systems can be used to keep track of variants, and a new variant can be created by creating a new branch. An existing variant is copied to this branch and the required changes are made. However, this approach is very limited in the number of variants that can be managed efficiently. Consider the situation of finding and fixing a bug in one of these variants. Does the bug-fix need to be applied in other variants? And if yes, in which variants? Another problem is that creating a new variant is complex and expensive, which means that it is usually not feasible to develop variants in advance. The idea of software product lines is to consider variability of software systems explicitly. Instead of developing a single product first and thinking about creating variants afterwards a whole set of products is planned from the beginning.

The software product line development process starts with the domain engineering phase [66]. The domain engineering phase consists of an initial domain analysis, followed by domain implementation.

During domain analysis, a domain (e.g, franking machine software) is chosen and possible variants are identified. The results of domain analysis are typically expressed in a feature model [34]. A feature model can be represented graphically by a feature diagram. In a feature diagram, features are arranged hierarchically [34]. It describes valid combinations of features that can be used to build variants. Figure 2.1 shows a feature model for our franking machine software product line. The feature model defines rules that must be followed in order to build a variant of the product line. Combinations of features that are allowed by a feature model are called valid configurations.

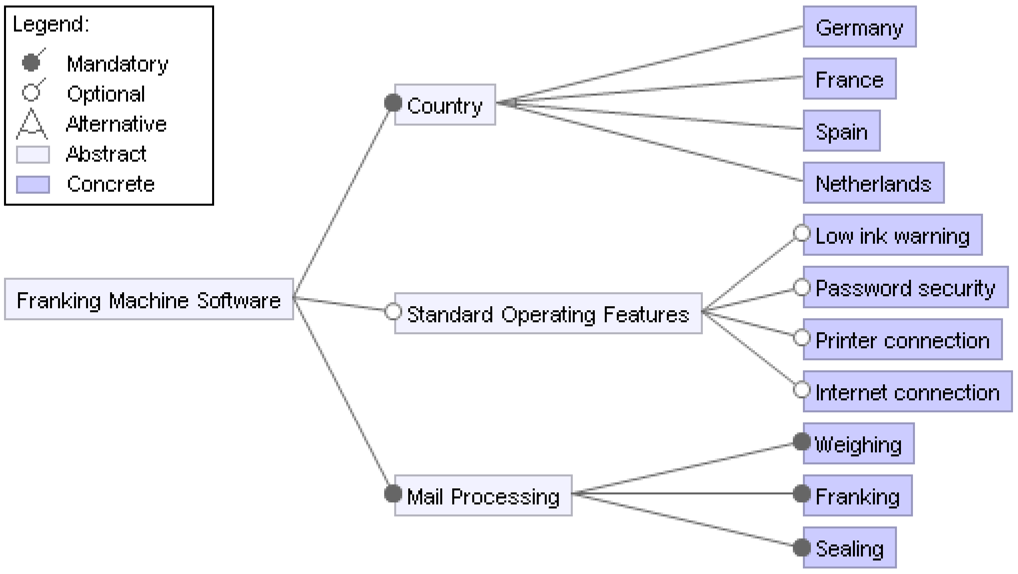


Figure 2.1: Feature model for franking machine product line

The hierarchical structure of a feature diagram implies the following meaning: If a feature is selected, the parent feature must be selected, too (e.g., if Spain is selected to be included in a variant, Country must be also be selected in order to get a valid variant). Mandatory features must be selected if their parent feature is selected. (e.g., Weighing, Franking, and Sealing must be selected if their parent is selected). Features that are not marked as mandatory are denoted as optional features and can either be selected or not selected (e.g., Printer connection is optional and can be selected if Standard Operating Features is selected). Alternatively, features can have one of several group types. The group type *Alternative* implies that exactly one of the child features must be selected (e.g., exactly one of the countries must be chosen). Group type *Or* demands that at least one of the child features must be selected. Abstract features are not mapped to implementation artifacts. They are used to structure the feature diagram. Features that are not abstract are denoted as concrete features.

2.2 Feature-Oriented Programming

After domain analysis the domain implementation can be performed. There are several implementation techniques for software product lines. Assuming the use of object-oriented programming the core assets could be a set of classes, templates, libraries, or a framework, from which the other variants can be derived. A possible approach for domain implementation is to use feature-oriented software development. In feature-oriented software development the mapping between code-artifacts and features is made explicit and features can be generated

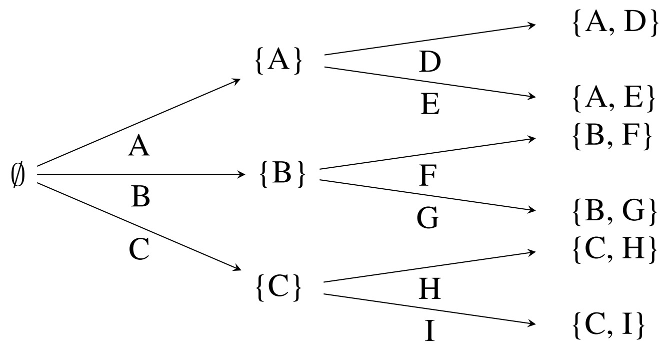


Figure 2.2: In stepwise development, program variants are derived by successive refinement of the program, starting from the empty program. In traditional stepwise development each edge represents an implementation decision, in feature oriented programming each edge represents a feature. Technically, a product is defined as an ordered set of implementation decisions/features.

automatically based on a configuration. Several techniques for the feature oriented implementation of code artifacts have been suggested [80]. Kästner et al. distinguish between compositional and annotative feature-oriented implementation techniques [36]. In compositional approaches, features are represented as distinct modules that can be composed automatically. Annotative approaches require to create a mapping between features and source code fragments by explicitly marking source code fragments.

This work focuses on feature-oriented programming, a compositional approach, in which the core assets are feature modules. Each feature module encapsulates an increment in functionality implementing a feature. Feature-oriented programming has been suggested as an extension of object-oriented programming [67], but can be generalized to support different languages and non-code artifacts [4]. Parnas proposes stepwise refinement to develop software product lines [63]. Figure 2.2 illustrates the principle of stepwise refinement. The developer starts with the empty program (denoted by the symbol $\emptyset$) and makes an implementation decision (e.g., introducing a method) resulting in a different program variant. Thus, a program variant can be described as an ordered set of implementation decisions. Batory proposes to apply step wise refinement on the level of features, in order to develop software product lines to achieve a better scalability [12]. In case of object oriented languages, a feature module can introduce classes, methods or fields or refine classes and methods. Figure 2.3 shows the composition of feature Processing and feature Low Ink Warning. Feature Processing contains basic functionality that is needed for our franking machine software. Feature

Low Ink Warning contains the code that is used to display a warning if the ink level is below a certain level.

Keyword `original` is used to reference the body of the original method, similar to the keyword `super()` to reference the implementation of a super class, as used in object-oriented programming.

Superimposition

In feature-oriented programming a variant is build by composing a set of features. A possible approach for the composition of features is superimposition of feature structure trees [5].

A feature-structure tree (FST) is a hierarchical representation that organizes the structural elements of a feature such as packages, classes, fields, or methods (or any other non-code or code artifact) [5]. Figure 2.4 shows a feature structure tree from our franking machine example. Each node in a feature structure tree has a name and a type (e.g., class `FrankingMachine` is represented by a node with the name *FrankingMachine* and the type *class*). We distinguish between terminal nodes (i.e., nodes without children) and non-terminal nodes (i.e., nodes with children). Feature structure trees can represent any kind of software artifact that can be expressed as an hierarchically structure.

Superimposition is a language independent composition method based on merging of tree structures. When superimposing two FSTs, their nodes are merged recursively based on their names, types, and relative positions. If two nodes have the same name, type and relative position they are merged. Nodes that cannot be matched this way are added to the FST at the current position. Two non-terminal nodes are merged by merging their children recursively. Terminal nodes are merged following language- and type specific composition rules.

When we refer to feature composition in this thesis, we always assume superimposition as the underlying composition mechanism. Formally, feature composition is described by the feature composition operator • defined over the set of features F [8].

$$\bullet : F \times F \to F$$

$$v = F_n \bullet F_{n-1} \bullet \dots \bullet F_1$$

A variant v is the result of a feature composition. Technically, it is not necessary to distinguish between variants and features because a variant is also a valid feature. The result of superimposition is a new FST, that can again be used as input for superimposition with another FST. Thus, superimposition can be used to compose an arbitrary number of FSTs

```
class FrankingMachine {                                          {Base}
 void process(Letter letter){
  weigh(letter);
  frank(letter);
  seal(letter);
  }
}
```

$\bullet$

```
class FrankingMachine {                               {Low Ink Warning}
 void process(Letter letter){
  if(inkLevel < inkLimit ){
      displayLowInkWarning();
      return;
  }
  original();
 }
}
```

$=$

```
class FrankingMachine {                         {Base, Low Ink Warning}
 void process(Letter letter){
  if(this.inkLevel < INK_LIMIT )
      displayLowInkWarning();
  weigh(letter);
  frank(letter);
  seal(letter);
  }
}
```

Figure 2.3: Feature Low Ink Warning refines class `FrankingMachine` by overriding method *process*. Keyword `original` is used to reference the original method body.

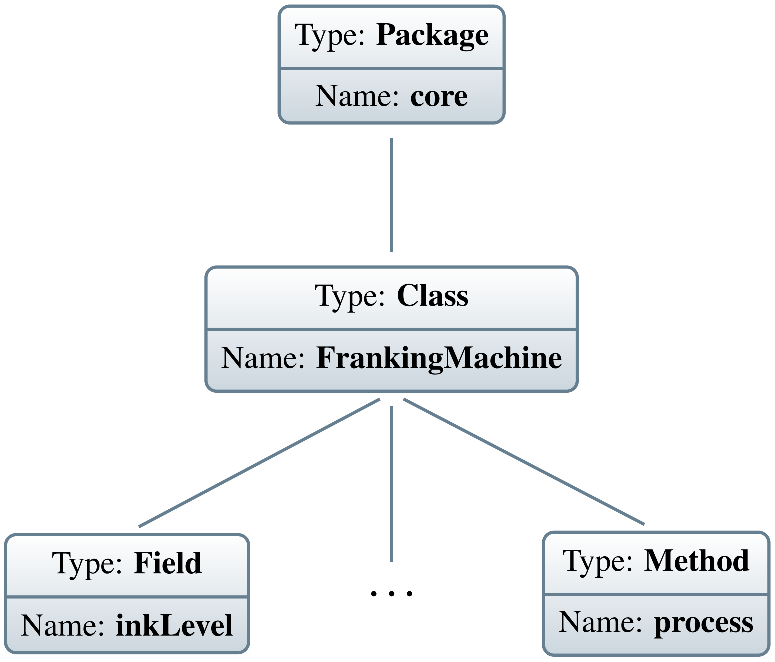


Figure 2.4: Feature structure tree representing a feature from the franking machine product line.

subsequently. Feature composition is associative, but not commutative (i.e., the result of the operation depends on the order in which two features are composed). Associativity, is desired for feature-oriented programming because it provides more freedom for the composition process [8].

2.3 Method Contracts

The notion of method contracts plays a central role in this thesis. Method contracts are a special form of assertion used to specify the behavior of methods. Section 2.3.1 introduces the concept of assertions and summarizes its main benefits. We give an overview of design by contract, a software development methodology, which coined the term of method contracts as used in this thesis.

2.3.1 Assertions

An assertion, in its general sense, is a property that must hold at a certain point of program execution. Initially, assertions were proposed as a means to simplify the understanding of a complex function. In 1945, Alan Turing proposed to use assertions that allow a human checker to reason about a large function step-by-step [79] [58]. Assertions state properties that can be checked in independently. If all assertions are considered to be fulfilled, one can conclude that the whole routine is correctly implemented.

For the rest of the thesis, we focus on such assertions that can be interpreted automatically, i.e. assertions that are stated in a formal language and usually embedded into program code to define the state of execution in which they apply. Such assertions can be used to verify that a given program fulfills properties of interest(stated as assertions). A straight-forward use of assertions is to include the assertions in an executable form in the compiled program, which will report an error if one of them is not fulfilled. This technique is known as runtime assertion checking [19].

The concept of assertions has proven to be adaptable to new programming concepts such as object-oriented programming and is still a key principle in formal methods. Hoare reports that about 250,000 lines of code of Microsoft Office are assertions [31].

According to Turing, assertions provide two key benefits [79]: modularity of reasoning (i.e., smaller implementation units can be analyzed in isolation) and blame assignment (i.e., it is easier to locate the source of errors).

2.3.2 Design by Contract

Design by contract is a method for software construction [56] [32]. It can be seen as an extension of assertions to object-oriented programming. In design by contract, the developer assumes a contract between a method and its caller. This contract is expressed formally in terms of pre- and postconditions. A method can rely on the properties stated by its precondition, while the caller can rely on the precondition to be fulfilled by the method. Furthermore, it is possible to define class invariants to state properties that must hold in every publicly visible state of execution (i.e., for and after calls to public methods).[2]

Some languages (e.g., Eiffel and Spec#) have native support for design by contract. For other languages there are extensions available (e.g., Java Modeling Language as an extension of Java).

We use the Hoare-style notation $\{\phi\}m\{\psi\}$ to express a method m with precondition ϕ and postcondition ψ. The precondition states properties that must be fulfilled when the method is called and the postcondition expresses properties that must be fulfilled after method execution. A contract $c = \{\phi\}m\{\psi\}$ states two properties: Precondition ϕ must be fulfilled and

[2]There is no final consensus about the semantics of invariants. Possible definitions may also include private method calls

$\phi \implies \psi$ must be fulfilled (i.e., if the precondition is fulfilled, the postcondition must be fulfilled, too).

We use the Java Modeling Language to exemplify the use of design by contract. The Java Modeling Language is a behavioral interface specification language. It allows developers to specify classes and methods on a semantic level [18] [43]. One of the design goals of the Java Modeling Language was to be easily understandable by Java developers without formal training in specification methods [45]. Therefore, it uses regular Java syntax enriched by a set of new keywords. Java Modeling Language specifications are embedded inside Java comments (i.e., they are ignored by Java compilers). A subset of the Java Modeling Language can be used to apply design by contract, i.e. class invariants and method contracts can be expressed in JML [44]. Since, Java Modeling Language is a formal language it can be processed automatically to reason about the program. There are tools for Java Modeling Language that support run-time assertion checking, static analysis, and the generation of documentation or test cases [16].

Design by contract transfers the key benefits of assertions to object-oriented programming. Leavens et al. list the following benefits [56]:

- Modularity of Reasoning - Contracts define an interface for methods. The developer can reason about a given method by considering only the contracts of called methods instead of their implementation. Thus, a method can be understood modularly.
- Blame Assignment - When an error is found contracts can be used to identify the source of this error. When a precondition is not satisfied the blame can be put on the caller. Analogously, the method implementation can be blamed in the case of a broken post-condition.
- Efficiency - Methods can rely on their precondition to be fulfilled when they are executed. Thus, methods can perform less checks on the validity of arguments to avoid redundant calculations.
- Documentation - Contracts express the intention of the developer. By only examining the implementation of a method, a developer can not distinguish between essential and arbitrary implementation details. Furthermore, he can often not decide whether an implementation is correct or not. By expressing all important properties of a method in its contract, the developer can distinguish between the intended behavior and the actual implementation.

3. Design by Contract in Feature-Oriented Programming

In the last chapter, we have introduced superimposition as a possible implementation technique for feature-oriented development of software product lines. In this thesis, we use superimposition to compose source code annotated with contracts. Superimposition can be used to compose all types of code artifacts that fulfill the following requirements [5]:

- The code artifact must be convertable into a feature-structure tree.
- The nodes of this tree must be identifiable by its relative position, its name, and its type.
- A node can only contain one node with the same name and type.
- Terminal nodes must provide rules for composition or cannot be composed.

Object-oriented languages usually fulfill these requirements and can be used in feature-oriented programming [4]. Design by contract is a development method that uses contracts to specify object-oriented languages [56] [32]. Languages that support design by contract are object-oriented languages that provide the possibility to specify methods with contracts [19].

We conclude that it is generally possible to use superimposition as an implementation technique for design by contract in feature-oriented programming. Contracts can be represented in a feature-structure tree as child nodes of methods. Each method can only have one contract. Thus, each node with the type `method` can only have one child node with the type *contract*. A unique name is not necessary to identify the node representing a contract. We can use a default name, as long as it is the same name for every contract. When two methods are composed, the corresponding contracts are also composed following a language-specific rule.

We do not need to know the composition rules to conclude that object-oriented languages with support for method contracts can be used for feature-oriented programming using superimposition. The details of these rules become important when we consider the purpose of the contracts. The specification of feature modules with contracts recently gained attention of researchers [77] [76] [69] [75]. They have been used as a means to specify software product lines and for specific applications such as the automated detection of feature interactions [69].

There are multiple ways to specify a software product line with contracts. One possible approach to specify a software product line is to specify each product separately [73]. However, the number of products in a software product line might be large (up-to exponential in the number of features). Thus, such a product-based specification approach is only applicable for small product lines.

The specification effort can be reduced by specifying each feature module separately, and generating the specification for each product automatically [77]. This can be done by enriching feature modules with contracts [77].

In feature-oriented programming, feature modules can introduce classes, methods, and fields. Furthermore, methods can be refined (i.e. they are overridden during feature composition). When applying design by contract to feature-oriented programming, method introductions can be enriched with contracts. However, it is not clear how the corresponding composition of two contracts should be performed. Generally, it is not obvious which approach developers should use to specify features.

Thüm et al. propose five approaches to specify contracts in feature-oriented programming: plain contracting, contract overriding, consecutive contract refinement, explicit contract refinement and pure method refinement [77]. Another approach is used for the detection of feature interactions [69]. We refer to this approach as cumulative contract refinement. These approaches differ in their expressiveness (i.e., on the set of programs that can be specified using a particular approach). More expressive approaches give the developer more freedom when specifying methods, but can lead to less understandable specifications. Less expressive approaches usually lead to simpler specifications, but may not be applicable in certain situations [77].

We propose to distinguish two aspects of each specification approach. First, each specification approach describes how developers should specify features with contracts. We call these descriptions contract specification strategies. Furthermore, each approach relies on a certain way in which contracts are composed during feature composition. We refer to these rules as contract composition mechanisms.

Consider the example of contract overriding, which is discussed in more detail later. On the one hand, it describes a contract specification strategy: Developers specify each method refinement with a complete specification of its behavior. On the other hand, contract overriding relies on a contract composition mechanism that composes two contracts by returning the original contract as a result.

We found ambiguities in the informal descriptions of contract specification approaches for feature-oriented programming. We provide more precise definitions of the specification approaches to provide a foundation for further research and tool development. Thus, we discuss the intended contract specification strategy for each specification approach, and formalize the underlying contract composition mechanisms in Section 3.1.

We define the specification strategies informally because they are intended to be applied by developers rather than machines. The contract composition mechanisms are supposed to be applied automatically by a tool. Therefore, we specify them formally. Our formalization is independent of a certain specification language such as the Java Modeling Language. Section 3.2 discusses the application of the theoretical concepts in the Java Modeling Language and language-specific properties that are not covered by the general definitions.

3.1 Definition of Contract Specification Approaches

The goal of this section is to define existing contract specification approaches. We define each contract specification approaches by distinguishing between contract specification strategies (ways to formulate contracts) and contract composition mechanisms (ways to compose two given contracts). The contract specification strategy can be seen as a guideline for the developer that must be followed in order to apply a certain contract specification approach. Each of these approaches also relies on a certain contract composition mechanism, that defines how the automatic composition of contracts is performed.

In the following, we discuss the existing contract specification approaches by giving an informal definition of the intended contract specification strategy and give formal definitions for each contract composition mechanism. The derivation of some of these definitions required us to make decisions on how to interpret existing descriptions of contract specification approaches. In these cases, we discuss the alternative interpretations.

The notion of a method contract is essential for both contract specification strategies and contract composition mechanisms. We define the term method contract as follows:

Definition 1 (Method Contract)**.** *Let P be the set of all propositional predicates and M the set of all methods. A* **method contract** *c is a triple (ϕ,m,ψ), where $\phi, \psi \in P$ and $m \in M$. ϕ is called the precondition of m, and ψ is called the postcondition of m. We denote the set of all possible method contracts with C.*

Based on our definition of method contracts, we formalize each contract composition mechanism by defining a function $\bullet\colon C \times C \to C$ that performs the desired operation. This function is similar to the feature composition operator, but describes feature composition on the level of methods and contracts. We define such a function $\bullet$ for each contract composition mechanism that we formalize.

An interesting property of any feature composition operator is associativity ($(A \bullet B) \bullet C = A \bullet (B \bullet C)$). This attribute is generally desired because it provides more freedom for the composition process [8]. Associativity allows tools to perform the composition in more than only one order. Thus, we analyze for each contract composition mechanism whether it is associative.

Thüm et al. distinguish between contract composition mechanisms that allow or prohibit the following types of contract refinement [73]:

- Weakening of contracts describes the ability to refine a contract in such a way that the contract is weaker than before. Given a method contract $c = \{\phi\}m\{\psi\}$, a refining contract $c' = \{\phi'\}m\{\psi'\}$ and their composition $c_{res} = \{\phi_{res}\}(m \bullet m)\{\psi_{res}\}$ the implications $\phi_{res} \implies \phi$ and $\psi_{res} \implies \psi$ are always true if weakening of contracts is not supported. If weakening of contracts is allowed these implications may be broken.

- Strengthening of contracts describes the ability to refine a contract in such a way that the contract is stronger, or more strict than before. Given a method contract $c = \{\phi\}m\{\psi\}$, a refining contract $c' = \{\phi'\}m\{\psi'\}$ and their composition $c_{res} = \{\phi_{res}\}(m \bullet m)\{\psi_{res}\}$ the implications $\phi_{res} \implies \phi$ and $\psi_{res} \implies \psi$ are always true if strengthening of contracts is not supported. If strengthening of contracts is not allowed allowed these implications may be broken.

- Behavioral subtyping states that the composition of two classes will always be a behavioral subtype of the base class. Technically, this means that strengthening of contracts is allowed and weakening of contracts is not allowed. Behavior subtyping is term that usually refers to inheritance in object-oriented programming. We use it in the context of feature composition. The relationship between these two notions is that if feature composition ensures behavioral subtyping, this means that the resulting class is a behavioral subtype of the class subject to refinement.

We analyze for each contract composition mechanism whether it supports weakening of contracts, strengthening of contracts, behavioral subtyping, or none of these properties.

Furthermore, a method contract $\{\phi\}m' \bullet m\{\psi\}$ states two properties. The first property is that the precondition must be fulfilled (i.e, ϕ must be ensured). The second property is that if the precondition is fulfilled, then the postcondition must be fulfilled, too (i.e., $\phi \implies \psi$ must be ensured). The result of contract composition can be classified, by which of these properties are fulfilled for a given contract composition mechanism. We classify each contract composition mechanism according to this classification.

3.1.1 Plain Contracting

Thüm et al. propose plain contracting as a simple contract specification approach for methods in feature-oriented programming [77]. The idea of plain contracting is that there is only one contract for any given method in the product line. The developer may specify a method with a contract, when the method is initially introduced. A feature may refine the implementation of an existing method, but not the contract. A goal of plain contracting is that the developer can rely on contracts not to be changed by feature refinements. Figure 3.1 shows an example in which method `process` is refined. Feature *Base* introduces method `process` and the corresponding contract. The precondition states that parameter `letter` must not be null. The postcondition states that after execution the letter can be expected to be weighted, franked, and sealed. Feature *Low Ink Warning* refines method `process`. It adds an additional check whether there is enough ink in the device. If the ink level is too low, a warning is displayed. The implementation is refined, but the contract is not changed.

The advantage of this approach is its simplicity [73]. The specification effort is relative low, because method refinements do not need to be specified. However, the applicability of plain contracting is rather limited [77]. The developer is able to refine the implementation of methods, but cannot refine the contract. Thus, the initial contract has to be defined in a way that conforms to all variants that include the same method. From the opposite perspective, feature-specific method refinements, such as the additional ink check inside method

```
class FrankingMachine {                                        {Base}
/*@requires letter != null;
  @ensures letter.isWeighted() && letter.isFranked()
  @ && letter.isSealed();
  @*/
 void process(Letter letter){
  weigh(letter);
  frank(letter);
  seal(letter);
  }
}
```

•

```
class FrankingMachine {                             {Low Ink Warning}
 void process(Letter letter){
  if(inkLevel < INK_LIMIT )
      displayLowInkWarning();
  original(letter);
  }
}
```

=

```
class FrankingMachine {                       {Base, Low Ink Warning}
/*@requires letter != null;
  @ensures letter.isWeighted() && letter.isFranked()
  @ && letter.isSealed();
  @*/
 void process(Letter letter){
  if(this.inkLevel < INK_LIMIT )
      displayLowInkWarning();
  weigh(letter);
  frank(letter);
  seal(letter);
  }
}
```

Figure 3.1: Plain contracting allows developers to define methods when introducing contracts. It does not allow to refine contracts.

`process`, cannot be specified with plain contracting. We give the following informal definition of the contract specification strategy for plain contracting as proposed by Thüm et al [77].

Definition 2 (Plain Contracting: Contract Specification Strategy)**.** *In plain contracting any method introduction may be specified with a contract. Method refinements can only change the implementation of a method, but not its contract. Thus, method refinements are not specified with contracts. The developer must ensure that any method inside a given feature module is either a method refinement in all products or a method introduction in all products.*

The idea of plain contracting is that contracts cannot be refined. However, we need to define a contract composition mechanism because our goal is to give definitions for contract-aware feature composition. Thus, it is necessary to specify the handling of contracts during feature composition. The contract composition mechanism that is required for plain contracting must ensure that the result of the composition of two contracts is identical to the original contract. The developer can just assume method refinements not to include contracts. However, our definition subsumes the more general case in which method refinements include contracts by treating a method without contract as having an empty contract.

Definition 3 (Plain Contracting: Contract Composition Mechanism)**.** *We define the contract composition mechanism for plain contracting as follows:*

$$\bullet\colon C \times C \to C$$

$$\{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\} = \{\phi\}m' \bullet m\{\psi\}$$

Following this definition, the resulting contract for $m' \bullet m$ is always identical to the contract of the method subject to refinement m. This definition is directly based on the informal description of plain contracting.

Proposition. *The contract composition mechanism for plain contracting is associative.*

Proof. Let C be the set of all method contracts. Given a contract composition operator for plain contracting with $\bullet\colon C \times C \to C, \{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\} = \{\phi\}m' \bullet m\{\psi\}$.

Given three contracts $c_1 = \{\phi\}m\{\psi\}, c_2 = \{\phi'\}m'\{\psi'\}, c_3 = \{\phi''\}m''\{\psi''\}, c \in C$.

Then

$(\{\phi''\}m''\{\psi''\} \bullet \{\phi'\}m'\{\psi'\}) \bullet \{\phi\}m\{\psi\}$

Contract Composition Mechanism	Plain Contracting
Definition:	$\{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\} = \{\phi\}m' \bullet m\{\psi\}$
Associative	yes
Ensured preconditions:	
ϕ	yes
ϕ'	no
Ensured postconditions:	
$\phi \implies \psi$	yes
$\phi' \implies \psi'$	no
Expressiveness:	
Weakening	no
Strengthening	no
Behavioral subtyping	no

Table 3.1: Properties of plain contracting.

= $(\{\phi'\}m'' \bullet m'\{\psi'\}) \bullet \{\phi\}m\{\psi\}$

= $\{\phi\}m'' \bullet m' \bullet m\{\psi\}$

= $\{\phi''\}m''\{\psi''\} \bullet \{\phi\}m' \bullet m\{\psi\}$

= $\{\phi''\}m''\{\psi''\} \bullet (\{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\})$.

It can be seen that, in general $(c_1 \bullet c_2) \bullet c_3 = c_1 \bullet (c_2 \bullet c_3)$. □

When two contracts are composed, plain contracting ensures that the result ensures the precondition of the contract subject to refinement. If it is fulfilled, it ensures that the original postcondition is also fulfilled. Plain contracting does not allow to weaken or strengthen contracts, because contracts cannot be refined. We summarize all properties of plain contracting in Table 3.1.

The distinction between contract specification strategy and contract composition mechanism provides several insights. When using the contract composition mechanism of plain contracting, method refinements cannot change the contract of a method. On first sight, this may seem sufficient to specify a product line as described in the contract specification strategy of plain contracting. Given a configuration and a certain composition order, it is possible to determine for each method in a feature module whether they are introductions or refinements. However, when considering the whole software product line a method can be both an introduction and a refinement, depending on the configuration (e.g., in the case of an optional method introduction). This means that a contract composition mechanism alone cannot enforce all required properties of plain contracting. The developer must avoid such situations. Thus, we have included this requirement in the contract specification strategy of plain contracting.

Furthermore, there are several details of plain contracting that have not been discussed, yet. In the case of method introductions without a contract (i.e., a method is introduced but has no contract) it is not intuitively clear for us, whether a refinement should be allowed to introduce a contract. The idea of plain contracting is that only method introductions include contracts. We use this definition directly to follow the principle of simplicity, which was proposed as the underlying motivation of this approach. Our definition conforms to this decision as we identify a missing contract with the empty contract, denoted by ϵ.

We identify a method without a contract as a method with an empty contract. Thus, when a method is introduced without a contract, it is not possible to define a contract for it in a refining feature. The alternative would be to allow developers to introduce one contract for each method instead of allowing them to introduce contracts only for method introductions. This requires to modify the contract composition mechanism to handle the case of empty contracts separately. We give a definition of a contract composition mechanism for plain contracting that allows developers to refine empty method contracts. We call this approach plain contracting without ϵ-contracts, referring to the identification of missing contracts with the empty contract ϵ.

Definition 4 (Plain Contracting without ϵ-Contracts: Contract Composition Mechanism)**.** *We define the contract composition mechanism for plain contracting without ϵ-contracts as follows:*

$$\bullet : C \times C \rightarrow C$$

$$\{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\} = \begin{cases} \{\phi\}m' \bullet m\{\psi\} & \textit{if } \phi \neq \epsilon \vee \psi \neq \epsilon \\ \{\phi'\}m' \bullet m\{\psi'\} & \textit{otherwise} \end{cases}$$

Proposition. *The contract composition mechanism for plain contracting without ϵ-contracts is not associative.*

Proof. We give a counterexample. Let C be the set of all method contracts. Given a contract composition operator for plain contracting without ϵ-contracts as defined in Definition 4. Given three contracts $c_1 = \{\epsilon\}m\{\epsilon\}$, $c_2 = \{\phi\}m'\{\psi\}$, $c_3 = \{\phi'\}m''\{\psi'\}$,

c $\in C, \phi \neq \epsilon, \psi \neq \epsilon$.

Contract Composition Mechanism	Plain Contracting without ϵ-contracts
Definition:	$\{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\} = \begin{cases} \{\phi\}m' \bullet m\{\psi\} & \text{if } \phi \neq \epsilon \vee \psi \neq \epsilon \\ \{\phi'\}m' \bullet m\{\psi'\} & \text{otherwise} \end{cases}$
Associative	no
Ensured preconditions:	
ϕ	yes
ϕ'	no
Ensured postconditions:	
$\phi \implies \psi$	yes
$\phi' \implies \psi'$	no
Expressiveness:	
Weakening	no
Strengthening	no
Behavioral subtyping	no

Table 3.2: Properties of plain contracting without ϵ-contracts.

Then

$(\{\phi'\}m''\{\psi'\} \bullet \{\phi\}m'\{\psi\}) \bullet \{\epsilon\}m\{\epsilon\}$

$= (\{\phi\}m'' \bullet m'\{\psi\}) \bullet \{\epsilon\}m\{\epsilon\}$

$= \{\epsilon\}m'' \bullet m' \bullet m\{\epsilon\}$

$\neq \{\phi\}m'' \bullet m' \bullet m\{\psi\}$

$= \{\phi'\}m''\{\psi'\} \bullet \{\phi\}m' \bullet m\{\psi\}$

$= \{\phi'\}m''\{\psi'\} \bullet (\{\phi\}m'\{\psi\} \bullet \{\epsilon\}m\{\epsilon\})$.

It can be seen that, in this case $(c_1 \bullet c_2) \bullet c_3 \neq c_1 \bullet (c_2 \bullet c_3)$. Thus, the contract composition operator is not associative. □

Plain contracting without ϵ-contracts is a modification of plain contracting that allows developers to add contracts to methods with empty (missing) contracts in feature refinements. The price we have to pay is the loss of associativity. We summarize the properties of plain contracting without ϵ-contracts in Table 3.2.

Additionally, the contract specification strategy can be altered to require a non-empty contract for each method introduction. This means that every method must be specified with a contract. This approach can be used to apply design by contract as a development method.

When using design by contract, every method is specified with a contract and the developer only relies on contracts of methods rather than implementation details. The result is that the case of empty contracts can be assumed to never occur. Both definitions of the contract composition mechanism for plain contracting are equivalent under this assumption.

It is possible that a method introduction only contains a precondition(or only a postcondition). Both contract composition mechanisms defined above do not allow a method refinement to introduce a postcondition (or a precondition) to this contract. We give a definition of a contract composition mechanism that allows developers to introduce for each method exactly one precondition and one postcondition during feature composition. This mechanism is not associative. We give a definition of this variation of plain contracting that we call plain contracting without ϵ-conditions.

Definition 5 (Plain Contracting without ϵ-Conditions: Contract Composition Mechanism)**.** *We define the contract composition mechanism for plain contracting without ϵ-conditions as follows:*

$$\bullet \colon C \times C \rightarrow C$$

$$\{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\} = \begin{cases} \{\phi\}m' \bullet m\{\psi\} & \textit{if } \phi \neq \epsilon \wedge \psi \neq \epsilon \\ \{\phi'\}m' \bullet m\{\psi\} & \textit{if } \phi = \epsilon \wedge \psi \neq \epsilon \\ \{\phi\}m' \bullet m\{\psi'\} & \textit{if } \phi \neq \epsilon \wedge \psi = \epsilon \\ \{\phi'\}m' \bullet m\{\psi'\} & \textit{if } \phi = \epsilon \wedge \psi = \epsilon \end{cases}$$

Proposition. *The contract composition mechanism for plain contracting without ϵ-conditions is not associative.*

Proof. We give a counterexample. Let C be the set of all method contracts. Given a contract composition operator for plain contracting without ϵ-conditions as defined in Definition 5. Given three contracts $c_1 = \{\epsilon\}m\{\epsilon\}$, $c_2 = \{\phi'\}m'\{\psi\}$, $c_3 = \{\phi''\}m''\{\psi'\}, c \in C$, $\phi \neq \epsilon, \psi \neq \epsilon$.

Then

$(\{\phi'\}m''\{\psi'\} \bullet \{\phi\}m'\{\psi\}) \bullet \{\epsilon\}m\{\epsilon\}$

= $(\{\phi\}m'' \bullet m'\{\psi\}) \bullet \{\epsilon\}m\{\epsilon\}$

= $\{\epsilon\}m'' \bullet m' \bullet m\{\epsilon\}$

Contract Composition Mechanism	Plain Contracting without ϵ-conditions
Definition:	$\{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\} = \begin{cases} \{\phi\}m' \bullet m\{\psi\} & \text{if } \phi \neq \epsilon \wedge \psi \neq \epsilon \\ \{\phi'\}m' \bullet m\{\psi\} & \text{if } \phi = \epsilon \wedge \psi \neq \epsilon \\ \{\phi\}m' \bullet m\{\psi'\} & \text{if } \phi \neq \epsilon \wedge \psi = \epsilon \\ \{\phi'\}m' \bullet m\{\psi'\} & \text{if } \phi = \epsilon \wedge \psi = \epsilon \end{cases}$
Associative	no
Ensured preconditions:	
ϕ	yes
ϕ'	no
Ensured postconditions:	
$\phi \implies \psi$	yes
$\phi' \implies \psi'$	no
Expressiveness:	
Weakening	no
Strengthening	no
Behavioral subtyping	no

Table 3.3: Properties of plain contracting without ϵ-conditions.

$\neq \{\phi\}m'' \bullet m' \bullet m\{\psi\}$

$= \{\phi'\}m''\{\psi'\} \bullet \{\phi\}m' \bullet m\{\psi\}$

$= \{\phi'\}m''\{\psi'\} \bullet (\{\phi\}m'\{\psi\} \bullet \{\epsilon\}m\{\epsilon\}).$

It can be seen that, in this case $(c_1 \bullet c_2) \bullet c_3 \neq c_1 \bullet (c_2 \bullet c_3)$. Thus, the contract composition operator is not associative. □

Plain contracting without ϵ-conditions is a modification of plain contracting without ϵ-contract in which the developer is allowed to introduce preconditions (or postconditions) in method refinements if the method subject to refinement does not have a precondition (or postcondition). We summarize the properties of plain contracting without ϵ-condition in Table 3.3.

Summary

Plain contracting is a simple approach to specify methods in feature-oriented programming. It relies on a contract composition mechanism that does not allow method refinements to refine contracts. This contract composition mechanism is associative, if it allows developers to introduce contracts in method refinements in the case that the method subject to refinement does not have a contract already. This is achieved by identification of a missing contract with the empty contract ϵ.

There are some possible alternatives about how methods without contracts can be handled when using plain contracting. The first is to require that every method must be specified. This approach can be used when applying design by contract as a development method, where the developer only relies on contracts rather than on implementation details.

The second possibility is to require that every method introduction must contain a contract. However, in this case the situation must be avoided that a method can be both an introduction and a refinement depending on the configuration.

The third possibility is to modify the contract composition mechanism to allow feature refinements to refine empty contracts. We give a definition for this alternative mechanism that allows developers the refinement of such empty contracts. This approach is called plain contracting without ϵ-contracts. However, it is not associative which means that any feature composition that uses this mechanism to compose contracts is also not associative. Furthermore, the granularity of this approach can be reduced by allowing developers not only to override empty contracts but also empty preconditions and empty postconditions separately. We denote this approach by plain contracting without ϵ-conditions.

3.1.2 Contract Overriding

Contract overriding allows the developer to specify both method introductions and method refinements to include contracts [77]. When a method is refined, the existing contract will be overridden by the refining contract.

Figure 3.2 shows an example in which method *process* is refined. The initial contract is replaced by the refining contract during composition to include the desired changes. The precondition of the new contract is the same in both the base feature and the refinement. It is generally often necessary to duplicate the original method contract or parts of it and include it in the refining contract.

Definition 6 (Contract Overriding: Contract Specification Strategy)**.** *When using contract overriding both method introductions and method refinements may be specified with a contract. Method refinements override the contract of the original method. Thus, each method refinement must include a contract that completely specifies the behavior of the method after refinement.*

Our formalization of the contract composition mechanism for contract overriding has a similar structure as the one for plain contracting. The difference is that the resulting contract is equivalent to the contract of the refinement, instead of the base contract.

Definition 7 (Contract Overriding: Contract Composition Mechanism)**.** *Contract composition for contract overriding is defined as follows:*

$$\bullet\colon C \times C \to C$$

$$\{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\} = \{\phi'\}m' \bullet m\{\psi'\}$$

Proposition. *The contract composition mechanism for contract overriding is associative.*

Proof. Let C be the set of all method contracts. Given a contract composition operator for contract overriding with $\bullet\colon C \times C \to C, \{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\} = \{\phi'\}m' \bullet m\{\psi'\}$.

Given three contracts $c_1 = \{\phi\}m\{\psi\}$, $c_2 = \{\phi'\}m'\{\psi'\}$, $c_3 = \{\phi''\}m''\{\psi''\}, c \in C$.

Then

$(\{\phi''\}m''\{\psi''\} \bullet \{\phi'\}m'\{\psi'\}) \bullet \{\phi\}m\{\psi\}$

= $\{\phi''\}m'' \bullet m'\{\psi''\} \bullet \{\phi\}m\{\psi\}$

```
class FrankingMachine {                                    {Base}
/*@requires letter != null;
  @ensures letter.isWeighted() && letter.isFranked()
  @ && letter.isSealed();
  @*/
 void process(Letter letter){
  weigh(letter);
  frank(letter);
  seal(letter);
 }
}
```

•

```
class FrankingMachine {                         {Low Ink Warning}
/*@requires letter != null;
  @ensures (inkLevel >= INK_LIMIT) ==>
  @ (letter.isWeighted() && letter.isFranked()
  @ && letter.isSealed());
  @*/
 void process(Letter letter){
  if(inkLevel < INK_LIMIT ){
      displayLowInkWarning();
      return;
      }
  original(letter);
  }
}
```

=

```
class FrankingMachine {                   {Base, Low Ink Warning}
/*@requires letter != null;
  @ensures (inkLevel >= INK_LIMIT) ==>
  @ (letter.isWeighted() && letter.isFranked()
  @ && letter.isSealed());
  @*/
 void process(Letter letter){
  if(this.inkLevel < INK_LIMIT )
      displayLowInkWarning();
  weigh(letter);
  frank(letter);
  seal(letter);
  }
}
```

Figure 3.2: Contract Overriding always overrides the original contract when refining a method.

Contract Composition Mechanism	Contract Overriding
Definition:	$\{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\} = \{\phi'\}m' \bullet m\{\psi'\}$
Associative	yes
Ensured preconditions:	
ϕ	no
ϕ'	yes
Ensured postconditions:	
$\phi \implies \psi$	no
$\phi' \implies \psi'$	yes
Expressiveness:	
Weakening	yes
Strengthening	yes
Behavioral subtyping	no

Table 3.4: Properties of contract overriding.

= $\{\phi''\}m'' \bullet m' \bullet m\{\psi''\}$

= $\{\phi''\}m''\{\psi''\} \bullet \{\phi'\}m' \bullet m\{\psi'\}$

= $\{\phi''\}m''\{\psi''\} \bullet (\{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\})$.

It can be seen that, in general $(c_1 \bullet c_2) \bullet c_3 = c_1 \bullet (c_2 \bullet c_3)$. □

Contract overriding ensures that the resulting contract fulfills the precondition of the refining contract. If it is fulfilled, it ensures that the corresponding postcondition is also fulfilled. Contract overriding supports to weaken and to strengthen contracts by overriding. We summarize all properties of contract overriding in Table 3.4.

When a method introduction does not include a contract, a contract may be introduced by a refinement. This follows directly from the definition of the contract composition mechanism for contract overriding. However, if a method refinement does not include a contract, any existing contract is overridden by the empty contract. Thus, the result of composition will not include a contract in this case. If this behavior is not desired, it is necessary to change the contract composition mechanism to handle the case of method refinements without contract separately. The downside of this approach is that the resulting contract composition mechanism is no longer associative. We give a definition for this alternative mechanism, that we call contract overriding without ϵ-contracts.

Definition 8 (Contract Overriding without ϵ-contracts: Contract Composition Mechanism)**.** *Contract composition for contract overriding is defined as follows:*

$$\bullet\colon C \times C \to C$$

$$\{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\} = \begin{cases} \{\phi'\}m' \bullet m\{\psi'\} & \textit{if } \phi \neq \epsilon \vee \psi \neq \epsilon \\ \{\phi\}m' \bullet m\{\psi\} & \textit{otherwise} \end{cases}$$

Proposition. *The contract composition mechanism for contract overriding without ϵ-contracts is not associative.*

Proof. We give a counterexample. Let C be the set of all method contracts. Given a contract composition operator for contract overriding without ϵ-contracts as defined in Definition 4. Given three contracts $c_1 = \{\epsilon\}m\{\epsilon\}$, $c_2 = \{\phi'\}m'\{\psi\}$, $c_3 = \{\phi''\}m''\{\psi'\}, c \in C$.

Then

$(\{\epsilon\}m\{\epsilon\} \bullet \{\phi'\}m'\{\psi'\}) \bullet \{\phi\}m'\{\psi\}$

= $\{\phi'\}m'' \bullet m'\{\psi'\} \bullet \{\phi\}m\{\psi\}$

= $\{\phi\}m'' \bullet m' \bullet m\{\phi\}$

$\neq \{\phi'\}m'' \bullet m' \bullet m\{\psi'\}$

= $\{\epsilon'\}m''\{\epsilon'\} \bullet \{\phi'\}m' \bullet m\{\psi'\}$

= $\{\epsilon\}m''\{\epsilon\} \bullet (\{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\})$.

It can be seen that, in this case $(c_1 \bullet c_2) \bullet c_3 \neq c_1 \bullet (c_2 \bullet c_3)$.

Thus, the contract composition operator is not associative. □

Contract overriding without ϵ-contracts is a modification of contract overriding in which methods without a contract do not override the contract of the method subject to refinement with the empty contract. We summarize all properties of contract overriding without ϵ-contracts in Table 3.5.

The developer might want to override only a precondition or only a postcondition. We give a definition of a contract composition mechanism that provides this behavior. We call this approach contract overriding without ϵ-conditions.

Contract Composition Mechanism	Contract Overriding without ϵ-Contracts
Definition:	$\{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\} = \begin{cases} \{\phi'\}m' \bullet m\{\psi'\} & \text{if } \phi \neq \epsilon \vee \psi \neq \epsilon \\ \{\phi\}m' \bullet m\{\psi\} & \text{otherwise} \end{cases}$
Associative	no
Ensured preconditions:	
ϕ	no
ϕ'	yes
Ensured postconditions:	
$\phi \implies \psi$	no
$\phi' \implies \psi'$	yes
Expressiveness:	
Weakening	yes
Strengthening	yes
Behavioral subtyping	no

Table 3.5: Properties of contract overriding without ϵ-contracts.

Definition 9 (Contract Overriding without ϵ-Conditions: Contract Composition Mechanism)**.** *We define the contract composition mechanism for contract overriding without ϵ-conditions as follows:*

$$\bullet : C \times C \to C$$

$$\{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\} = \begin{cases} \{\phi'\}m' \bullet m\{\psi'\} & \textit{if } \phi \neq \epsilon \wedge \psi \neq \epsilon \\ \{\phi\}m' \bullet m\{\psi'\} & \textit{if } \phi = \epsilon \wedge \psi \neq \epsilon \\ \{\phi'\}m' \bullet m\{\psi\} & \textit{if } \phi \neq \epsilon \wedge \psi = \epsilon \\ \{\phi\}m' \bullet m\{\psi\} & \textit{if } \phi = \epsilon \wedge \psi = \epsilon \end{cases}$$

Proposition. *The contract composition mechanism for contract overriding without ϵ-conditions is not associative.*

Proof. We give a counterexample. Let C be the set of all method contracts. Given a contract composition operator for contract overriding without ϵ-conditions as defined in Definition 9. Given three contracts $c_1 = \{\epsilon\}m\{\epsilon\}$, $c_2 = \{\phi'\}m'\{\psi\}$, $c_3 = \{\phi''\}m''\{\psi'\}, c \in C$.

Then

$(\{\epsilon\}m\{\epsilon\} \bullet \{\phi'\}m'\{\psi'\}) \bullet \{\phi\}m'\{\psi\}$

Contract Composition Mechanism	Contract Overriding without ϵ-Conditions
Definition:	$\{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\} = \begin{cases} \{\phi'\}m' \bullet m\{\psi'\} & \text{if } \phi \neq \epsilon \wedge \psi \neq \epsilon \\ \{\phi\}m' \bullet m\{\psi'\} & \text{if } \phi = \epsilon \wedge \psi \neq \epsilon \\ \{\phi'\}m' \bullet m\{\psi\} & \text{if } \phi \neq \epsilon \wedge \psi = \epsilon \\ \{\phi\}m' \bullet m\{\psi\} & \text{if } \phi = \epsilon \wedge \psi = \epsilon \end{cases}$
Associative	no
Ensured preconditions:	
ϕ	no
ϕ'	yes
Ensured postconditions:	
$\phi \implies \psi$	no
$\phi' \implies \psi'$	yes
Expressiveness:	
Weakening	yes
Strengthening	yes
Behavioral subtyping	no

Table 3.6: Properties of contract overriding without ϵ-conditions.

= $\{\phi'\}m'' \bullet m'\{\psi'\} \bullet \{\phi\}m\{\psi\}$

= $\{\phi\}m'' \bullet m' \bullet m\{\phi\}$

$\neq$ $\{\phi'\}m'' \bullet m' \bullet m\{\psi'\}$

= $\{\epsilon'\}m''\{\epsilon'\} \bullet \{\phi'\}m' \bullet m\{\psi'\}$

= $\{\epsilon\}m''\{\epsilon\} \bullet (\{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\})$.

It can be seen that, in this case $(c_1 \bullet c_2) \bullet c_3 \neq c_1 \bullet (c_2 \bullet c_3)$.

Thus, the contract composition operator is not associative. □

Contract overriding without ϵ-conditions is a modification of contract overriding in which methods without a contract do not override the contract of the method subject to refinement with the empty contract. We summarize all properties of contract overriding without ϵ-conditions in Table 3.6.

Summary

Contract Overriding is a contract specification approach that allows features to refine contracts by overriding them. The contract of the original feature is overridden. We have given three alternative definitions of contract overriding. They differ in the handling of missing or empty contracts. The first does not distinguish between contracts and missing or empty contracts. Thus, when a method refinement does not include a contract, the original contract will be overridden by the empty contract. We assume this behavior might be counter-intuitive and requires to introduce specification clones if a method refinement does not change the contract.

The alternative is to define the composition mechanism in a way that empty contracts do not override contracts. This can be done by considering complete contracts, or pre- and post-conditions separately. Contract overriding without ϵ-contracts does not override a contract if the refining method has no contract. If the refining method has only a precondition but no postcondition, the original postcondition will be overridden by the empty postcondition. Contract overriding without ϵ-conditions does not override a precondition (or postcondition) if the refining method has no precondition (or postcondition). The downside of both alternatives is that these mechanism are not associative. Only contract overriding, in which empty contracts override contracts by default is associative.

3.1.3 Keyword `original` (Explicit Contract Refinement)

Explicit contract refinement is a contract specification approach that is based on overriding contracts [77]. The difference to contract overriding is that the developer can reference the original contract from inside the contract refinement by using the keyword `original`. This is a technique developers in feature-oriented programming are familiar with because it is similar to the way methods are refined. When keyword `original` occurs inside a precondition it references the original precondition, when it occurs inside a postcondition it references the original postcondition. The keyword `original` can be used additional with the previous defined composition mechanisms. However, it does not influence the result of composition in the case of plain contracting because contract refinements are ignored by this approach.

Our definition of explicit contract refinement is an extension of contract overriding that supports the use of keyword `original`.

Definition 10 (Explicit Contract Refinement: Contract Specification Strategy). *When using explicit contract refinement both method introductions and method refinements may be specified with a contract. Method refinements override the contract of the original method. Thus, each method refinement must include a contract that completely specifies the behavior of the method after refinement.*

The underlying contract composition mechanism is also an extension of the mechanism used for contract overriding. We give a definition that is more general in the sense that it can be applied to extend any contract composition mechanism, including but not limited to contract overriding, contract overriding with ϵ-contracts, and contract overriding with ϵ-conditions. The application of the definition of keyword `original` on contract overriding leads to the contract composition mechanism of explicit contracting. This allows us to extend other mechanisms for support of keyword `original`.

Definition 11 (keyword `original`)**.** *Given a contract composition mechanism, as a function t: $C \times C \rightarrow C$. We can modify t to support keyword* `original` *by replacing t by t' with $t'(c) = t(f(c, o)), c \in C, o \in \{true, false\}$, where o represents the support for keyword* `original` *of the given contract specification technique(i.e. function f performs the necessary transformations related to keyword* `original` *before the actual contract composition takes place).*

We define function f as follows:

$$f : C \times C \times \{true, false\} \rightarrow C \times C$$

$$f(\{\phi\}m\{\psi\}, \{\phi'\}m'\{\psi'\}, o) = \begin{cases} (\{\psi\}m\{\phi\}, \{\psi'\}m'\{\phi'\}) & \textit{if } o = \textit{false} \\ (\{\phi\}m\{\psi\}, \{\phi' \bullet \psi\}m'\{\psi' \bullet \psi\}) & \textit{if } o = \textit{true}\,, \end{cases}$$

where $\{\phi' \bullet \psi\} = \phi''$, where ϕ'' is derived from ϕ' by replacing each occurrence of keyword `original` *by ϕ.*

Explicit contract refinement is a contract specification approach that allows the developer to avoid specification clones (in comparison to contract overriding) by referencing the original contract. It relies a composition mechanism that extends the contract composition mechanism for contract overriding with support for keyword `original`.

The combination of a given contract composition mechanism with support for keyword `original` does not change the properties of the contract composition mechanism that we have discussed. For a given method contract with keyword `original` it is always possible to express the same contract by directly including the referenced specification. However, when considering the variability of the whole product line, keyword `original` does change the semantics of a given composition mechanism, as we have seen above.

3.1.4 Consecutive Contract Refinement

In consecutive contract refinement, method introductions can be specified with contracts and method refinements can introduce new contracts that must be fulfilled in addition to the original contract [77]. The idea is that both the original contract and the contract introduced by the refinement must not be violated.

Figure 3.3 shows an example of consecutive method refinement. Feature *Base* introduces method `process`, that performs the processing of letters. The precondition states that a letter must not be null and the postcondition states that each letter is weighted, franked, and sealed after execution. Feature *Low Ink Warning* refines method `process` by adding a check of the ink level. If it is too low, a warning is displayed before processing the letter as usual. The contract of the method refinement does only specify the additional behavior. The precondition states that the field `inkLevel` must not be null. The postcondition states that if the ink level is too low, a corresponding warning is displayed. The result of composition is a method, that includes both contracts. In this case, we use the JML keyword `also` which expresses this behavior.

This directly leads to our definition of the contract specification strategy for consecutive contract refinement.

Definition 12 (Consecutive Contract Refinement: Contract Specification Strategy)**.** *When using consecutive contract refinement both method introductions and method refinements may be specified with a contract. The intended meaning of a method refinement including a contract is that the new contract must be fulfilled in addition to the original contract. For the resulting method both contracts must be fulfilled.*

We identified three alternative interpretations of the informal requirement stated by Thüm et al. [77]. We refer to these approaches as conjunctive contract refinement, cumulative contract refinement, and consecutive contract refinement. The most simple variant is to require that all preconditions and postconditions must be fulfilled. We call this approach conjunctive contract refinement and define the composition mechanism as follows:

Definition 13 (Conjunctive Contract Refinement: Contract Composition Mechanism)**.**

$$\bullet : C \times C \rightarrow C$$

$$\{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\} = \{\phi' \land \phi\}m' \bullet m\{\psi' \land \ \psi)\}$$

Proposition. *The contract composition mechanism for conjunctive contract refinement is associative.*

```
class FrankingMachine {                                          {Base}
/*@requires letter != null;
  @ensures letter.isWeighted() && letter.isFranked()
  @ && letter.isSealed();
  @*/
 void process(Letter letter){
  weigh(letter);
  frank(letter);
  seal(letter);
  }
}
```

•

```
class FrankingMachine {                               {Low Ink Warning}
/*@requires inkLevel!=null;
  @ensures(inkLevel < INK_LIMIT) ==> display.warning==INK_WARNING;
  @*/
 void process(Letter letter){
  if(inkLevel <INK_LIMIT ){
      displayLowInkWarning();}
  original(letter);
  }
}
```

=

```
class FrankingMachine {                         {Base, Low Ink Warning}
/*@requires letter != null;
  @ensures letter.isWeighted() && letter.isFranked()
  @ && letter.isSealed();
  @also
  @requires inkLevel!=null;
  @ensures(inkLevel < INK_LIMIT) ==> display.warning==INK_WARNING;
  @*/
  void process(Letter letter){
  if(this.inkLevel < INK_LIMIT )
      displayLowInkWarning();
  weigh(letter);
  frank(letter);
  seal(letter);
  }
}
```

Figure 3.3: Consecutive contracting allows developers to add a contract when refining a method. The original contract and the refined contract both have to be fulfilled in the composition.

Proof. Let C be the set of all method contracts. Given a contract composition operator for conjunctive contract refinement with $\bullet : C \times C \rightarrow C, \{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\} = \{\phi' \wedge \phi\}m' \bullet m\{\psi' \wedge \ \psi)\}$.

Given three contracts $c_1 = \{\phi\}m\{\psi\}$, $c_2 = \{\phi'\}m'\{\psi'\}$, $c_3 = \{\phi''\}m''\{\psi''\}, c \in C$.

Then

$(\{\phi''\}m''\{\psi''\} \bullet \{\phi'\}m'\{\psi'\}) \bullet \{\phi\}m\{\psi\}$

= $(\{\phi'' \wedge \phi'\}m'' \bullet m'\{\psi'' \wedge \psi'\}) \bullet \{\phi\}m\{\psi\}$

= $\{(\phi'' \wedge \phi') \wedge \phi\}m'' \bullet m' \bullet m\{(\psi'' \wedge \psi') \wedge \psi\}$

= $\{\phi'' \wedge (\phi' \wedge \phi)\}m'' \bullet m' \bullet m\{\psi'' \wedge (\psi' \wedge \psi)\}$

= $\{\phi''\}m''\{\psi''\} \bullet (\{\phi' \wedge \phi\}m' \bullet m\{\psi' \wedge \psi\})$

= $\{\phi''\}m''\{\psi''\} \bullet (\{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\})$.

It can be seen that, in general $(c_1 \bullet c_2) \bullet c_3 = c_1 \bullet (c_2 \bullet c_3)$. □

Conjunctive contract refinement ensures that the resulting contract fulfills both preconditions of the composed contracts. If they are both fulfilled, it ensures that both corresponding postconditions are also fulfilled. Conjunctive contract refinement allows feature refinements to strengthen contracts by adding additional pre- and postconditions that must be fulfilled. However, it does not allow to weaken contracts, because all properties of the contract subject to refinement are ensured by the result of composition. We summarize all properties of contract overriding in Table 3.7.

Conjunctive contract refinement is more restrictive than would be necessary to apply the idea of consecutive contract refinement as proposed in literature. The reason is that requires both preconditions to be fulfilled at the same time which would not be the case if a method had multiple contracts. The question whether this approach is appropriate is related to the question if a precondition must hold for every contract or if the contract only requires the implication: If the precondition is fulfilled, then the postcondition must be fulfilled. Runtime-Assertion checking tools, usually assume that a precondition must be fulfilled, because they generate corresponding assertions and report any violation of pre- or postconditions [19].

We derive an alternative contract composition mechanism by changing the composed precondition. Instead of requiring all preconditions to hold, we consider it sufficient when one of them is fulfilled (i.e., if one of the preconditions is fulfilled, all the postconditions must be fulfilled). This approach is also used by researchers for the detection of feature interactions [69].

Contract Composition Mechanism	Conjunctive Contract Refinement
Definition:	$\{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\} =$ $\{\ \phi' \wedge \phi\}m' \bullet m\{\psi' \wedge\ \psi)\}$
Associative	yes
Ensured preconditions:	
ϕ	yes
ϕ'	yes
Ensured postconditions:	
$\phi \implies \psi$	no
$\phi' \implies \psi'$	no
Expressiveness:	
Weakening	no
Strengthening	yes
Behavioral subtyping	yes

Table 3.7: Properties of conjunctive contract refinement.

We define this approach as a contract composition mechanism and call it cumulative contract refinement.

Definition 14 (Cumulative Contract Refinement: Contract Composition Mechanism)**.**

$$\bullet \colon C \times C \to C$$

$$\{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\} = \{\phi' \vee \phi\}m' \bullet m\{\psi' \wedge\ \psi)\}$$

Proposition. *The contract composition mechanism for cumulative contract refinement is associative.*

Proof. Let C be the set of all method contracts. Given a contract composition operator for cumulative contract refinement with $\bullet \colon C \times C \to C, \{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\} = \{\phi' \vee \phi\}m' \bullet m\{\psi' \wedge\ \psi)\}$.

Given three contracts $c_1 = \{\phi\}m\{\psi\}$, $c_2 = \{\phi'\}m'\{\psi'\}$, $c_3 = \{\phi''\}m''\{\psi''\}, c_i \in C$.

Then

$(\{\phi''\}m''\{\psi''\} \bullet \{\phi'\}m'\{\psi'\}) \bullet \{\phi\}m\{\psi\}$

= $(\{\phi'' \vee \phi'\}m'' \bullet m'\{\psi'' \wedge \psi'\}) \bullet \{\phi\}m\{\psi\}$

= $\{(\phi'' \vee \phi') \wedge \phi\}m'' \bullet m' \bullet m\{(\psi'' \wedge \psi') \wedge \psi\}$

Contract Composition Mechanism	Cumulative Contract Refinement
Definition:	$\{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\} =$ $\{\ \phi' \vee \phi\}m' \bullet m\{\psi' \wedge\ \psi)\}$
Associative	yes
Ensured preconditions:	
ϕ	no
ϕ'	no
Ensured postconditions:	
$\phi \implies \psi$	yes
$\phi' \implies \psi'$	yes
Expressiveness:	
Weakening	no
Strengthening	yes
Behavioral subtyping	yes

Table 3.8: Properties of cumulative contract refinement.

= $\{\phi'' \vee (\phi' \wedge \phi)\}m'' \bullet m' \bullet m\{\psi'' \wedge (\psi' \wedge \psi)\}$

= $\{\phi''\}m''\{\psi''\} \bullet (\{\phi' \vee \phi\}m' \bullet m\{\psi' \wedge \psi\})$

= $\{\phi''\}m''\{\psi''\} \bullet (\{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\})$.

It can be seen that, in general $(c_1 \bullet c_2) \bullet c_3 = c_1 \bullet (c_2 \bullet c_3)$. □

Cumulative contract refinement ensures that the resulting contract fulfills both postconditions of the composed contracts if one of the corresponding preconditions is fulfilled. It does not allow to weaken contracts, in the sense used in this thesis. However, it does allow to weaken preconditions by adding additional preconditions disjunctively and to strengthen postconditions by adding additional postconditions that must be fulfilled. It does not ensure for a single precondition of original contracts that it must be fulfilled, but it ensures that if a certain precondition is fulfilled, the corresponding postcondition must be fulfilled, too. We summarize all properties of contract overriding in Table 3.8.

Cumulative contract refinement is less restrictive than conjunctive contract refinement. However, there are cases in which it demands the fulfillment of a postcondition without the related precondition to be fulfilled. We assume this is often not feasible, because the postcondition can usually not be fulfilled without the corresponding precondition (otherwise the precondition would have been unnecessary).

Contract Composition Mechanism	Consecutive Contract Refinement
Definition:	$\{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\} =$ $\{\ \phi' \wedge \phi\}m' \bullet m\{\psi' \wedge\ \psi)\}$
Associative	yes
Ensured preconditions:	
ϕ	no
ϕ'	no
Ensured postconditions:	
$\phi \implies \psi$	yes
$\phi' \implies \psi'$	yes
Expressiveness:	
Weakening	no
Strengthening	yes
Behavioral subtyping	yes

Table 3.9: Properties of consecutive contract refinement.

This problem leads us to the third interpretation, which is similar to the notion of behavioral subtyping as known from class hierarchies in object-oriented programming [48]. We define contract composition for consecutive contract refinement as follows:

Definition 15 (Consecutive Contract Refinement: Contract Composition Mechanism)**.**

$$\bullet : C \times C \to C$$

$$\{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\} = \{\phi' \vee \phi\}m' \bullet m\{(old(\phi) \implies \psi) \wedge\ (old(\phi') \implies \psi')\}$$

The occurrence of variables in a postcondition usually refers to their value after method execution. Function `old` is used to refer to the values of variables of a condition in the state before method execution.

Consecutive contract refinement ensures that both original postconditions must be fulfilled, if the corresponding preconditions are fulfilled. However, it does not enforce that all preconditions must be fulfilled. Consecutive contract refinement is associative. The proof can be done analogously to the previous proofs of associativity, but the necessary transformations to show the equivalence are rather long and do not provide much insight. Instead, we only sketch the proof and skip some details. We give an overview of all properties in Table 3.9.

Proposition. *The contract composition mechanism for consecutive contract refinement is associative.*

Proof Sketch. *Let C be the set of all method contracts. Given a contract composition operator for consecutive contract refinement with* $\bullet\colon C \times C \to C$, $\{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\} = \{\phi' \vee \phi\}m' \bullet m\{(old(\phi) \implies \psi) \wedge (old(\phi') \implies \psi')\}$.

Given three contracts $c_1 = \{\phi\}m\{\psi\}$*,* $c_2 = \{\phi'\}m'\{\psi'\}$*,* $c_3 = \{\phi''\}m''\{\psi''\}, c_i \in C$*.*

Then

$(c_1 \bullet c_2) \bullet c_3 = \{\phi'' \wedge \phi' \wedge \phi\}m'' \bullet m' \bullet m\{(\phi'' \implies \psi'') \wedge (\phi' \implies \psi') \wedge (\phi \implies \psi)\}$,
and

$c_1 \bullet (c_2 \bullet c_3) = \{\phi'' \wedge \phi' \wedge \phi\}m'' \bullet m' \bullet m\{(\phi'' \implies \psi'') \wedge (\phi' \implies \psi') \wedge (\phi \implies \psi)\}$.

It can be seen that, in general $(c_1 \bullet c_2) \bullet c_3 = c_1 \bullet (c_2 \bullet c_3)$*.*

The definitions we have given do not achieve the goal of consecutive contract refinement that methods can be assumed as having multiple contracts. The reason is that it is generally not possible to express the semantics of multiple contracts in only one contract. Consider two contracts $A = \{\phi\}m\{\psi\}$ and $B = \{\phi'\}m'\{\psi'\}$. Contract A states two properties that must be fulfilled. First, precondition ϕ must be fulfilled. Second, if precondition ϕ is fulfilled, postcondition ψ must be fulfilled (i.e. $\phi \implies \psi$). Analogously, contract B states that precondition ϕ' must be fulfilled and that $\phi' \implies \psi'$ must be true.

Table 3.10 gives an overview of which properties of the original contracts are enforced by our three definitions. Having two contracts would mean that all four properties of the original contracts(ϕ, ϕ',$\phi \implies \psi$, $\phi' \implies \psi'$) must be fulfilled. However, it is generally impossible to find such a definition. Any contract that requires both ϕ', ϕ in its precondition, cannot ensure that ψ is ensured if only ϕ is true (i.e., $\phi \implies \psi$ cannot be ensured).

All three definitions of the contract composition mechanism assume that no empty pre- and postconditions can occur. However, this assumption is too restrictive. When composing two contracts, the result contains conditions in which pre- and postconditions are used inside a propositional expression. These conditions can be empty. We must define for each case how the empty contract is evaluated when it occurs inside a propositional expression.

When using conjunctive contract refinement, the empty condition can only occur inside a conjunction. If we define that the empty condition evaluates to true in all cases, the composition works as intended.

In the case of cumulative contract refinement we must distinguish between pre- and postconditions. In a precondition, the empty condition can occur only inside a disjunction and must

	Conjunctive Contract Refinement	Cumulative Contract Refinement	Consecutive Contract Refinement
Associative	yes	yes	yes
ϕ	yes	no	no
ϕ'	yes	no	no
$\phi \implies \psi$	no	yes	yes
$\phi' \implies \psi'$	no	yes	yes
Weakening	no	no	no
Strengthening	yes	yes	yes
Behavioral subtyping	yes	yes	yes

Table 3.10: Overview of fulfilled properties of conjunctive contract refinement, cumulative contract refinement, and consecutive contract refinement.

evaluate to false. However, this assumption is not sufficient, because it leads to an unsatisfiable precondition if all preconditions are empty. Thus, this case must be handled separately. However, this leads to more complex composition rules that must distinguish between a precondition that cannot be satisfied because all preconditions used to compose it are empty and a precondition that cannot be satisfied for other reasons. In a postcondition, the empty condition can occur only inside a conjunction and must evaluate to true. The result, is that in all cases a missing contract is treated as desired.

Consecutive contract refinement is even more complicated. Preconditions must be handled like in cumulative contract refinement. The same problems arise. For empty postconditions, it is sufficient when empty conditions evaluate to true. The result is that a missing condition has no influence on the resulting expressions. It ensures the same properties of interest as cumulative contracting but is more likely to be applicable because it does not require that preconditions must be fulfilled without the corresponding preconditions being fulfilled. It is also similar to the notion of behavioral subtyping from object-oriented inheritance.

Summary

The idea of this specification approach is that contracts must be composed in a way, such that the resulting contract can be viewed as a method with two contracts. It is generally not possible to express all properties of two contracts in a single contract. Thus, it is not possible to find a composition mechanism that ensures these properties. Instead, we have defined three alternative contract composition mechanisms for the specification approach consecutive contracting. They all have in common that they allow strengthening of contracts, but not weakening (i.e. they follow the behavioral subtyping principle).

Conjunctive contract refinement is the only approach that ensures that both preconditions and both postconditions must be fulfilled. However, this property also makes it the most restrictive one. It does not ensure that if only one of the original preconditions is fulfilled that the corresponding precondition must be fulfilled.

Cumulative contract refinement is less restrictive, because it only requires that one of the original preconditions must be fulfilled. Thus, method callers can rely that the resulting contract ensures that if he provides one of the original preconditions the corresponding original postcondition is fulfilled. However, this approach imposes difficulty for method implementations. They must ensure that both preconditions are fulfilled if only one of the corresponding preconditions is fulfilled. Thus, this approach is only applicable if the postcondition can be fulfilled without the corresponding precondition being fulfilled. We assume that this is generally not possible because in this case the precondition was not necessary at all.

The composition mechanism consecutive contract refinement is named after the specification approach because we believe this definition comes closest to the desired properties of this specification approach.

We have identified that the handling of empty contracts must be considered explicitly if there absence cannot be assumed.

3.1.5 Pure Method Refinement

Pure method refinement is a specification approach for contracts in feature-oriented programming [77]. It relies on the ability of certain specification language for contracts, in which methods of the specified class can be called and executed. The Java Modeling Language supports this for methods that are marked as pure, which denotes that the method cannot influence existing objects. In feature-oriented programming it is possible to refine methods. Thus, it is possible to refine contracts semantically, by refining the implementation of pure methods.

The underlying, implicit assumption of pure method refinement as proposed in literature is that methods are specified in a way similar to plain contracting [77]. Thus, it can be seen as a modification of plain contracting in which features can be refined by refining pure methods. However, it is possible to combine pure method refinement with other approaches. We could assume any of the previously defined contract composition mechanisms and allow the refinement of pure contracts (by changing the contract specification strategy to allow it). If we do not mention that a specific specification approach supports the use of pure method refinement we assume that it is never used.

The advantage of pure method refinement is that contract refinements can be of finer granularity compared to other approaches. It is possible to place multiple pure method calls into one pre- or postcondition. Each of these pure methods can be refined individually.

We assume, a downside of pure method refinement is that each part of the specification that may be subject to refinement, must be defined by a pure method call. Thus, the specification of features is less flexible than their implementation. In feature-oriented programming, the developer can generally refine any method as long as it exists. When using pure method refinement, the developer can refine all methods but the corresponding contract can only be refined if a pure method call is used or introduced.

Whether the need to put specifications that may be refined in pure methods is an advantage or disadvantage depends on the used specification strategy. It is a disadvantage if the developer does not suspect this behavior and must create pure methods in the base program when specifying methods in a refinement. It can also be an advantage if pure method calls are used to specify parts of the specification that may be refined by features.

We do not define a contract composition mechanism because it can be used directly with any of the previously defined mechanisms. However, our definitions of the previous approaches assume that pure method refinement is not used. If pure method calls are allowed, the resulting contract specification approach supports both weakening and strengthening of contracts because pure methods can be refined arbitrarily.

We believe, that the combination with plain contracting is the most promising. It allows the developer to express behavior that must not be violated by any feature directly inside contracts rather than in pure methods. Properties that may be changed by features can be specified inside pure methods. This is applicable for properties of different granularity from single values that may be changed to whole methods that are specified inside a single pure method.

3.1.6 Summary

Our goal is to develop tool support for the application of design by contract in feature-oriented programming. We considered six approaches to specify contracts in feature oriented programming that are used in software product line research: plain contracting, contract overriding, explicit contract refinement, consecutive contract refinement, cumulative contract refinement, and pure method refinement. These approaches have been described informally in existing work and leave several open questions.

We propose to distinguish two characteristics for each of these approaches. First, the way in which developers must specify contracts, the contract specification strategy. Second, the way in which two contracts are composed during feature composition, the contract composition mechanism. This distinction is useful when trying to understand a given specification approach. It turns out that existing descriptions of approaches focus on describing the specification strategy in some cases. In other cases they focus on the composition mechanism.

We provide definitions of contract specification strategies and contract composition mechanisms for feature-oriented programming. A contract specification strategy is intended to be a guideline for developers. Thus, we focus on defining the main idea of each strategy informally and discuss several alternatives and extensions.

Each contract specification strategy assumes and relies on a specific contract composition mechanism. We give formal definitions of contract composition mechanisms, because they are intended to be implemented by tools. Thus, it is desirable to specify them precisely. Each contract states two properties: its precondition must be fulfilled, and if it is fulfilled the postcondition must hold. The three definitions differ in the properties of the original contracts that must be fulfilled for the resulting contract. Ideally, all properties should be fulfilled. However, this is not possible. On the one hand, we want that both original preconditions must be fulfilled. On the other hand, we want that if we provide a certain precondition the corresponding postcondition must be fulfilled. However, we cannot expect the method to ensure one of the original postconditions by only providing the corresponding precondition because its precondition requires that both must be fulfilled.

Table 3.11 summarizes the fulfilled properties for each composition mechanism. Some properties of contract composition mechanisms are not reflected in this overview (e.g., cumulative contract refinement is more restrictive than consecutive contract refinement because the postcondition requires that both original postconditions must be fulfilled in all cases).

The contract composition mechanism for plain contracting must ensure that contracts cannot be refined. We propose three alternative definitions that differ in the case that not every method has both pre- and postconditions. The advantage of these alternative definitions is that the developer is more flexible when introducing contracts. They allow to introduce contracts in method refinements, if the method subject to refinement does not have a contract (have the empty contract ϵ) in the previous refinement. Plain contracting without ϵ-contracts does this on the level of whole contracts and plain contracting without ϵ-conditions distinguishes between pre- and postconditions. The downside of both approaches is that they are not associative—a property that is desired because it provides more freedom for composition tools.

We propose three alternative contract composition mechanism that differ in the way missing (empty) contracts are handled. The question is whether a method refinement without a contract (i.e. with empty contract ϵ) should override the original contract. This is the default behavior of contract overriding. It is applicable if the developer follows a strategy in which this is reflected (i.e. if an empty contract overrides a contract, this should be the intention of the developer). We define contract overriding without ϵ-contracts and contract overriding without ϵ-conditions analogously to the similar definitions of plain contracting. They do not override contracts (pre- and postconditions) if the method refinement does not include one. Again, contract overriding is generally associative, but these modifications destroy this property.

However, contract overriding is not very flexible when it comes to feature composition A method refinement must include the whole specification as it is needed in the product. Generally, there are optional features that may have to be considered in the specification. However, this cannot be done using this approach.

We define explicit contract refinement as contract overriding with additional support for keyword `original`. Theoretically, it is possible to allow the same support for other approaches like consecutive contract refinement. Thus, we define original in a way that can be combined with an arbitrary contract composition mechanism.

We propose three definitions of contract composition strategies, including the ones described for consecutive contract refinement and cumulative contract refinement in literature. The third one is the most restrictive and we call it conjunctive contract refinement. When using conjunctive contract refinement, the result of composition ensures that both original preconditions and both original postconditions must be fulfilled. Cumulative contract refinement is less restrictive for the precondition: only one of the original preconditions must be fulfilled. Consecutive contract refinement also ensures that at least one precondition must be fulfilled. Furthermore, the original postconditions must be fulfilled only in the case that the corresponding precondition is fulfilled.

Pure method refinement describes the possibility to refine specifications by refining pure methods [77]. These pure methods are side-effect free methods that can be called from inside contracts. Thus, by refining pure methods the specification can be refined.

	Plain Contracting	Plain Contracting without ϵ-contracts	Contract Overriding	Contract Overriding without ϵ-contracts	Conjunctive Contract Refinement	Cumulative Contract Refinement	Consecutive Contract Refinement
Associative	yes	no	yes	no	yes	yes	yes
ϕ	yes	yes	no	no	yes	no	no
ϕ'	no	no	yes	yes	yes	no	no
$\phi \implies \psi$	yes	yes	no	no	no	yes	yes
$\phi' \implies \psi'$	no	no	yes	yes	no	yes	yes
Weakening	no	no	yes	yes	no	no	no
Strengthening	no	no	yes	yes	yes	yes	yes
Beh. subtyping	no	no	no	no	yes	yes	yes

Table 3.11: Overview of ensured properties for plain contracting, contract overriding, conjunctive contract refinement, cumulative contract refinement, and consecutive contract refinement. Plain contracting (and contract overriding) without ϵ-conditions has the same properties as plain contracting (contract overriding) without ϵ-contracts.

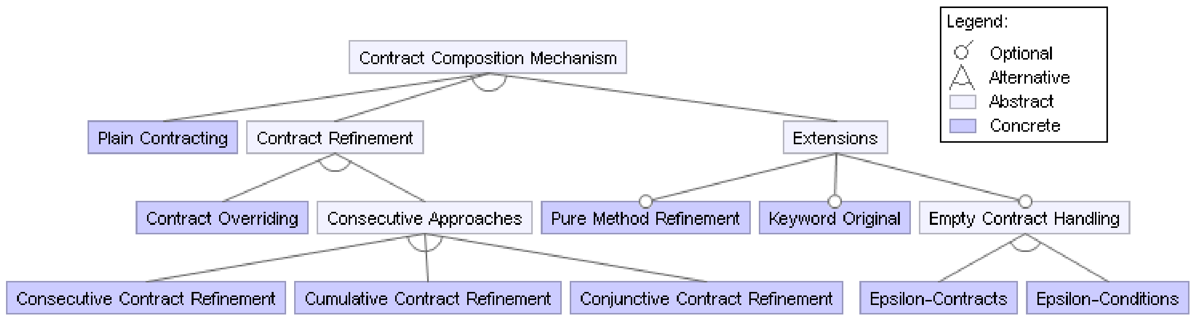


Figure 3.4: Variability model for contract composition mechanisms.

We view pure method refinement not as a specification approach but more like an optional extension for other approaches. It does not specify a contract specification strategy, but one is needed because we need to handle the composition of contracts in a certain way. A possible application of pure method refinement is to combine it with plain contracting. Contracts can generally not be refined and must be fulfilled in all products. The exception are specifications expressed in pure methods. They can be refined by usual feature composition.

As stated above, it is possible to combine specification approaches with support for keyword `original` or pure method refinement. Figure 3.4 shows a feature model that describes a product line of contract composition mechanism. This product line contains 17 valid configurations. However, the combination of plain contracting and keyword `original` is possible but not useful because contracts cannot be refined.

We have given definitions of composition mechanism for all existing approaches to specify contracts in feature-oriented programming. We have discussed advantages and disadvantages of different contract composition mechanisms and possible variations. Our definitions are language-independent and may be applied to different languages. We discuss how to implement these mechanisms in the next section.

3.2 Feature-oriented Programming with JML

The last chapter provided a definition of possible contract specification approaches in a language-independent way. In Section 3.2.2 we discuss important differences between the basic concept of design by contract and its support in JML. We discuss concrete realization possibilities for the proposed contract composition mechanisms in Section 3.2.1. Furthermore, the results of this chapter are used as a basis for our tool support.

3.2.1 Contract Composition in JML

We defined contract composition approaches in a general, language-independent fashion. We now give an overview of how the underlying contract composition mechanisms can be implemented in JML. In some cases, the definitions can be applied directly. In other cases, it is necessary to consider certain details.

Plain Contracting/Contract Overriding

Plain contracting and Contract Overriding can be performed in JML straight-forwardly. Both approaches can be implemented similarly, and distinguish in the contract that is used as the result of composition. As far as tool-support is concerned it could be useful to permit refining methods from containing contracts instead of ignoring them during composition.

We assume that plain contracting and contract overriding without ϵ-contracts are the variations that were assumed as composition mechanisms in existing work. We evaluate this assumption in Chapter 5.

Consecutive contract refinement

For consecutive contract refinement, we suggest two alternative implementation techniques. The first alternative is to apply the definition of consecutive contract refinement or one of the alternative definitions (cumulative contract refinement, conjunctive contract refinement), as defined in Section 3.1, directly. Alternatively, it is possible to concatenate multiple specification cases by using the keyword `also`. Figure 3.5 shows the implementation of this approach in JML. Keyword `also` is semantically equivalent to consecutive contract refinement [42].

Keyword `Original`

JML does not have native support for keyword `original`. We extended the JML language definition by support for keyword `original`. Some details regarding specific JML features are discussed in the next section.

3.2.2 Extended features of JML

The Java Modeling Language is a behavioral interface specification language that provides a great variety of means to specify Java programs. Design by contract is a software development methodology that evolved from language features of the Eiffel programming language. JML supports design by contract as a subset while providing a huge number of extending features. In the following section we will explain the major differences between the basic

```
class FrankingMachine {                                              {Base}
/*@requires letter != null;
  @ensures leter.isWeighted() && letter.isFranked()
  @ && letter.isSealed();
  @*/
 void process(Letter letter){
  weigh(letter);
  frank(letter);
  seal(letter);
  }
 }
```

●

```
class FrankingMachine {                                   {Low Ink Warning}
/*@requires inkLevel != null;
  @ensures (inkLevel >= INK_LIMIT) ==>
  @ (letter.isWeighted() && letter.isFranked()
  @ && letter.isSealed());
  @*/
 void process(Letter letter){
  if(inkLevel < INK_LIMIT ){
      displayLowInkWarning();
      return;}
  original(letter);
  }
}
```

=

```
class FrankingMachine {                             {Base, Low Ink Warning}
/*@requires letter != null;
  @ensures (inkLevel >= INK_LIMIT) ==>
  @ (letter.isWeighted() && letter.isFranked()
  @ && letter.isSealed());
  @*/
 void process(Letter letter){
  if(this.inkLevel < INK_LIMIT )
      displayLowInkWarning();
  weigh(letter);
  frank(letter);
  seal(letter);
  }
}
```

Figure 3.5: Example of consecutive contract refinement in JML.

```
class FrankingMachine {                                   {Base, Low Ink Warning}
/*@requires letter != null;
  @ensures letter.isWeighted() && letter.isFranked()
  @ && letter.isSealed();
  @also
  @requires inkLevel!=null;
  @ensures (inkLevel < INK_LIMIT) ==> display.warning==INK_WARNING;
  @*/
  void process(Letter letter){
  if(this.inkLevel < INK_LIMIT )
      displayLowInkWarning();
  weigh(letter);
  frank(letter);
  seal(letter);
  }
}
```

Figure 3.6: Keyword also can be used to define multiple specification cases in JML

concepts of design by contract and the features that JML provides. We argue that while JML offers extended specification possibilities, the contract composition mechanisms can still be applied.

Multiple Specification Cases

Considering the core principle of design by contract, every method can have exactly one precondition and one postcondition. However, JML supports the use of multiple method contracts (specification cases) for one method, by connection them with the keyword `also`. Figure 3.6 shows the usage of such specification cases. For plain contracting, multiple specification cases do not impose any problems. The difference is that method introductions are allowed to be specified with multiple specification cases in this case.

When applying contract overriding in JML, it is necessary to consider the case of multiple specification cases. We suggest the convention that when a contract is overridden, the whole contract including all specification cases is replaced with the new contract. An alternative approach would be to distinguish between different specification cases and override them separately. However, JML does not provide means to reference specification cases explicitly (such as names for specification cases). Therefore, specification cases would have to be referenced implicitly by their relative position. However this leaves several open questions, such as what happens if the number of specification cases in the refined method does not match the one of the refining method.

Multiple specification cases also need to be considered when using keyword `original`. We propose to support different granularities by distinguishing three keywords:

- Keyword `originalSpec`:
  Keyword `originalSpec` refers to the whole method specification. This approach can be used when the developer wants to specify a new property of a method and the old specification should still hold. Consecutive contract refinement uses the same approach but implicitly includes the keyword `original` to each method. This keyword can be implemented by adding the new part of specification as a new specification case using keyword also.

- Keyword `originalCase`:
  Keyword `originalCase` refers to a specific specification case. However, specification cases cannot be referenced explicitly and must therefor be matched by the order in which they appear.

- Keyword `original`:
  Keyword `original` refers to a specific assertion clause type within a specific specification case. All matching clauses will be merged into one conjunction that is inserted at the position of the keyword `original`.

Figure 3.7 shows the use of the different versions of keyword `original`. The result of all versions is similar except for the position in which the specification cases occur in the result.

These three approaches can also be combined by providing three keywords that can be used as needed.

When considering consecutive contract refinement, we can exploit the ability to specify multiple specification cases by adding the contract of the refining method as a novel specification case. This is done by concatenation of specification cases and connecting them with keyword `also`.

Multiple Pre- and Postconditions

In JML it is possible to define multiple pre- and postconditions for each specification case. However, it is possible to transform multiple pre- and postconditions to a contract with only one pre- and postcondition. A given set of preconditions (or postconditions) $S =$

Base

```
/** Return this + b (the sum of this and b). */
/*@ requires b != null;
  @ ensures \result != null;
  @ ensures JMLDouble.approximatelyEqualTo(
  @             this.realPart() + b.realPart(),
  @             \result.realPart(),
  @             tolerance);
  @ also
  @ requires tolerance != null;
  @*/

    Complex add(Complex b);
```

Refinement (originalSpec)

```
/** Return this + b (the sum of this and b). */
/*@ /original_spec
  @ also
  @ ensures JMLDouble.approximatelyEqualTo(
  @             this.imaginaryPart() + b.imaginaryPart(),
  @             \result.imaginaryPart(),
  @             tolerance);
  @*/
    Complex add(Complex b);
```

Refinement (originalCase)

```
/** Return this + b (the sum of this and b). */
/*@ /original_case
  @ also
  @ ensures JMLDouble.approximatelyEqualTo(
  @             this.imaginaryPart() + b.imaginaryPart(),
  @             \result.imaginaryPart(),
  @             tolerance);
  @ also
  @ /original_case
  @*/
    Complex add(Complex b);
```

Refinement (original)

```
/** Return this + b (the sum of this and b). */{
/*@ requires \original;
  @ ensures \original;
  @ ensures JMLDouble.approximatelyEqualTo(
  @             this.imaginaryPart() + b.imaginaryPart(),
  @             \result.imaginaryPart(),
  @             tolerance);
  @ also
  @ requires /original
  @*/
    Complex add(Complex b);
```

Figure 3.7: This example shows how method add can be refined using the different versions of keyword `original`. Keyword `originalSpec` refers to the whole specification defined in feature Base while `originalCase` refers to single specification cases. The first occurrence of `originalCase` refers to the first case etc. Keyword `original` refers to the type of clause it is used in, e.g. `requires /original` in the first specification case refers to all requires clauses in the first specification case of feature base.

```
/** Return this + b (the sum of this and b).
/*@ requires letter!=null;
  @ requires letter.isWeighted();
  @ requires letter.isFranked();
  @ ensures letter.isSealed();
  @*/
 void seal(Letter ){...};
```

```
/** Return this + b (the sum of this and b). */
/*@ requires letter!=null
  @   && letter.isWeighted()
  @   && letter.isFranked();
  @ ensures letter.isSealed();
  @*/
 void seal(Letter ){...};
```

Figure 3.8: Multiple pre- or postconditions can be transformed into a single contract with only one pre- and postcondition.

$\{\phi_1, \phi_2, ..., \phi_n\}$ can be replaced by a single precondition (postcondition) containing the expression $\bigwedge_{1\leq i\leq n} \phi_i$. Figure 3.8 shows how multiple preconditions can be replaced while preserving semantics. The result is that this feature does not restrict the applicability of our definitions, because we can avoid these cases by performing the transformation automatically during feature composition. Thus, developers do not have to avoid multiple pre- and postconditions when specifying methods.

Different Assertion Types

JML allows developers to specify preconditions using *requires* clauses and postconditions using *ensures* clauses. However, there are multiple additional types of clauses that can be used to specify properties of a given method. Important examples are *signals* and *assignable* clauses. *Signals* clauses specify a set of possible exception types that may be thrown by the method. *Assignable* clauses specify a set of variables that can be modified during method execution. In this thesis we focus on boolean assertion types (i.e., assertion types that can be evaluated to a boolean value). However, it is possible to define a composition operator for other types (e.g., concatenation with removing duplicates to compose two lists of assignable variables)

3.2.3 Summary

In this chapter, we have sketched how our definitions of contract composition mechanisms can be implemented in JML. We have taken characteristics of the Java Modeling Language that extend the notion of a contract into account. The most important differences that are relevant for contract composition are support for multiple specification cases, multiple pre- and postconditions and different assertion types. Multiple specification cases demand special treatment for composition mechanisms, especially when using keyword `original`. However, they also provide a convenient possibility to implement consecutive contract refinement by using keyword `also`. Multiple pre- and postconditions can be considered as syntactic sugar because it is possible to transform them in equivalent contracts with only one pre- and postcondition. Different assertion types have to be considered when developing tool support because some types are expressed as lists rather than propositional expressions. Thus it is necessary to develop type-specific composition mechanisms such as the concatenation of signals clauses.

4. Tool Support

In the last chapter, we discussed how to specify feature modules using the Java Modeling Language. So far, the composition of features had to be performed without tool support. The goal of our tool is to support design by contract in feature-oriented programming with Java Modeling Language. The tool should be integrated into a development environment to increase usability. Our tool allows users to develop feature modules using Java Modeling Language and automatically perform contract-aware feature composition to generate variants. Furthermore, the tool allows users to switch between different contract composition mechanisms during runtime in order to compare specification approaches.

In this chapter, we present our tool support for design by contract in feature-oriented programming. We extended FeatureHouse to support the Java Modeling Language and integrated this version of FeatureHouse into FeatureIDE. Section 4.1 gives a short introduction to FeatureIDE and explains the integration of our tool. In Section 4.2 we describe our extension of FeatureHouse that supports the Java Modeling Language. Known limitations of our implementation are summarized in Section 4.3.

4.1 Extension of FeatureIDE

We extended FeatureIDE with support for contracts in feature-oriented programming expressed in the Java Modeling Language. The implementation of contract composition mechanisms is based on our results of Section 3.2. FeatureIDE is a framework for feature-oriented software development and deployed as plug-in for Eclipse [74]. It can be installed through the Eclipse Marketplace or by downloading it from the FeatureIDE website. FeatureIDE is open-source, which allows us to extend it to suit our needs. Furthermore, it supports the developer in all phases of the software product line development process (i.e., domain engineering and application engineering). FeatureIDE integrates tools for different implementation techniques for software product lines. We only sketch such aspects of the framework that are necessary to understand our implementation.

In FeatureIDE, domain analysis is supported by a graphical editor that can be used to create and edit feature models. Several views provide additional support during this phase. Editors and views related to this phase are independent from the implementation of code artifacts and shared between all implementation tools integrated in FeatureIDE. Thus, our tool support can benefit from this part of FeatureIDE without additional work.

When creating a new project, FeatureIDE will generate the initial project infrastructure such as a feature model, and folders for feature modules and configuration files. The feature model can then be edited graphically, while FeatureIDE automatically manages the mapping from features to code artifacts. Furthermore, FeatureIDE provides a configuration editor to select the desired features to build a variant. This provides instant feedback for the developer in the case of compiler errors.

For domain implementation, FeatureIDE provides support for a variety of tools supporting different implementation techniques, namely feature-oriented programming [67], aspect-oriented programming [38] [37], preprocessor directives [59] [65], and delta-oriented programming [68]. One of the integrated tools that supports feature-oriented programming is FeatureHouse [5].

We extended FeatureIDE to support choosing between different composition mechanisms when using FeatureHouse. The user can use a configuration file (contract.style) to specify which composition mechanism to use for each project. The configuration file must be placed in the configuration folder provided by FeatureIDE projects. The choice for a certain composition mechanism is propagated to FeatureHouse by passing each line in the configuration file as a command-line parameter. Possible parameters are: plain_contracting, contract_overriding, consecutive_contract_refinement. Support for keyword `original` can be activated by additionally using the parameter "original". Pure method refinement is supported by default, without the need for a parameter.

4.2 Extension of FeatureHouse

The core functionality of our tool support is based on FeatureHouse. FeatureHouse is a framework for the composition of software artifacts. FeatureHouse can compose software artifacts written in different languages (e.g., source code, test cases, models, documentation) [5]. FeatureHouse includes FSTComposer, a tool for superimposition of feature structure trees as described in Section 2.2, and FSTGenerator that provides support for integrating new languages.

Figure 4.1 visualizes the composition of code artifacts in FeatureHouse. An input file is processed as a stream of characters by a language-specific lexical analyzer. The lexical analyzer is capable of processing the character stream and provides a stream of tokens that serves as input for the corresponding parser. The parser performs the actual processing. It takes the token stream from the lexical analyzer as input and creates the corresponding feature structure tree as output. When composing two features containing the same class, the parser will create a feature structure tree for each of them. These feature structure trees are composed by the tool FSTComposer of FeatureHouse using superimposition. Terminal nodes with same name and type are composed according to type-specific composition rules. Finally, a language-specific pretty printer creates the resulting output files.

We extended FeatureHouse by support for the Java Modeling Language. Currently, our extension of FeatureHouse supports plain contracting, contract overriding, and consecutive contract refinement—combined with optional support for keyword `original` and pure method refinement. We define these approaches in Section 3.1. Furthermore, our extension supports JML language levels 0-3 as described in the JML reference manual [46] and Java 1.5 features (e.g. Java‘s generics) [27].

In order to extend FeatureHouse with support for the Java Modeling Language we have to provide a lexical analyzer, a parser, a pretty printer and all necessary composition rules, to perform the composition of terminal nodes in the FST. The FSTGen tool of FeatureHouse automates the creation of the lexical analyzer, the parser, and the pretty printer based on a FeatureBNF grammar that defines the desired language. Thus, we developed a FeatureBNF grammar for the Java Modeling Language. Furthermore, we implemented JML-specific composition rules to extend FeatureHouse with support for JML.

4.2.1 Grammar

The first step to integrate support for a new language into FeatureHouse is to develop a FeatureBNF grammar that defines the language precisely. A FeatureBNF grammar consists of two parts: the first part includes directives that are used to build the lexical analyzer. Our first approach to develop a FeatureBNF grammar for JML was to use the JML Grammar defined in the JML reference manual and translate it into a FeatureBNF grammar. However,

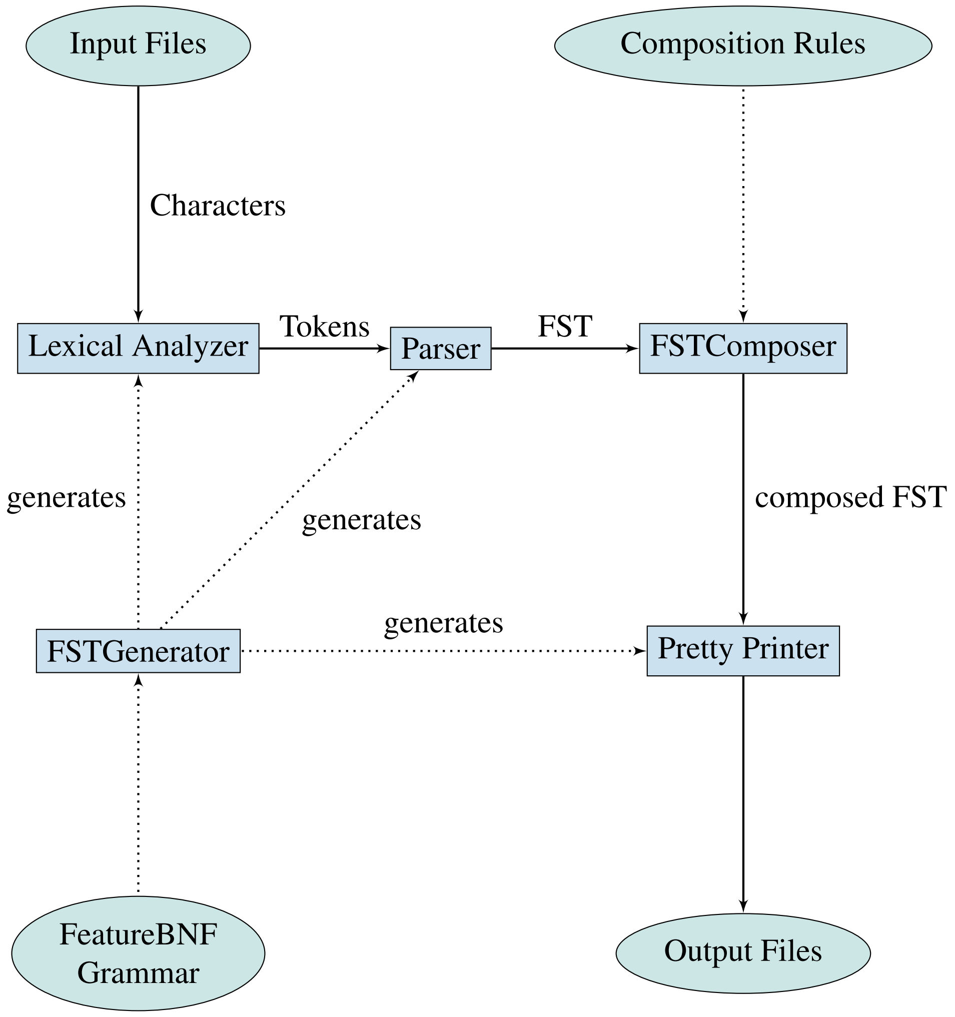


Figure 4.1: Visualization of the composition process in FeatureHouse.

this approach turned out to come with several problems. First, the JML grammar as defined in the reference manual contains several errors, that require an extensive knowledge of the Java language specification to correct. Our first attempts to correct these errors resulted in errors in the Java subset of the grammar. Thus, our grammar did not recognize some correct Java programs as conforming to the grammar. Second, the grammar does not always distinguish between the Java subset and the JML subset of the language explicitly in all cases. This made it even more difficult to identify errors. Third, the JML grammar supports only Java 1.4 language features, but we wanted to support Java 1.5 features.

To avoid the above mentioned problems with the JML grammar we started from an existing Java 1.5 FeatureBNF grammar provided by FeatureHouse and extended it with support for JML. We give a brief overview about how to use the FeatureBNF language to define a language. We explain only features that are important to understand our implementation. However, there are additional features provided by the FeatureBNF language that may be useful when developing a grammar for other languages [4].

As far as the lexical analyzer is concerned, we had to perform two main changes to the existing FeatureBNF grammar for Java 1.5 for Java Modeling Language support.

First, we added JML-specific keywords to the list of strings that are recognized as tokens by the parser. We added 148 additional keywords to the Java grammar to support all JML language features as defined in the JML Reference Manual [46]. Figure 4.2 shows an excerpt of our FeatureBNF grammar, which defines JML-specific keywords. Each keyword definition consists of a name that can be used to reference the keyword in grammar rules and a string defining the character sequence that is recognized as a token representing the keyword.

The second change is related to the fact that JML specifications are placed inside special Java comments. The problem is that Java comments are usually skipped by the lexical analyzer. We extended the lexical analyzer by adding a second lexical state in which JML-specific processing is performed. Each production is then marked with the set of lexical states in which they are applied (and ignored in all other states). In Figure 4.2 the Java keywords are marked with <Java, JML> which means that these keywords are visible in both lexical states. The JML-specific keywords are marked with <JML> which denotes that these keywords are only visible in the lexical state named JML. This distinction is necessary because valid Java identifiers are reserved keywords in JML. Thus, they can be used in the Java subset of JML, but not inside JML-comments.

```
//Java keywords
<Java, JML>
TOKEN :
{
  < ABSTRACT: "abstract" >
| < ASSERT: "assert" >
| < BOOLEAN: "boolean" >
| < BREAK: "break" >
| < BYTE: "byte" >
| < CASE: "case" >
...
  < VOLATILE: "volatile" >
| < WHILE: "while" >
}

//JML keywords
<JML>
TOKEN :
{
  <ORIGINAL: "\\original">
| < MODEL: "model" >
| < SPEC_PROTECTED: "spec_protected" >
| < GHOST: "ghost" >
| < PURE: "pure" >
| < INSTANCE: "instance" >
...
| < NORMAL_EXAMPLE: "normal_example" >
| < MODEL_PROGRAM: "model_program" >
}
```

Figure 4.2: Excerpt from the FeatureBNF grammar for JML: Definition of reserved keywords.

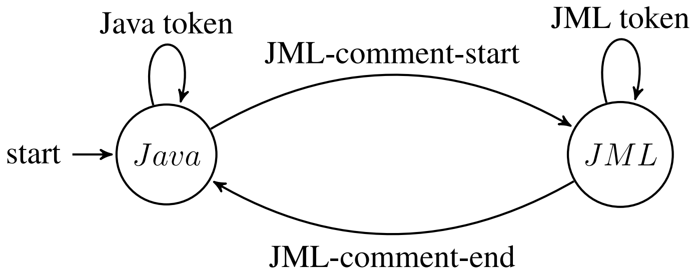


Figure 4.3: The lexical analyzer distinguishes two lexical states representing JML and Java.

When the lexical analyzer finds the beginning of a JML comment it switches to the lexical state in which JML is processed correctly. When the JML comment ends the lexical analyzer switches back to the state in which only Java is processed. Thus, the lexical analyzer implements a finite state automaton with two states. This behavior is visualized in Figure 4.3. The automaton starts in the state *Java* in which Java keywords are recognized as tokens. Processing of a beginning JML comment triggers the automaton to switch into the state *JML* in which JML keywords are recognized as tokens.

The second part of the FeatureBNF grammar includes all grammar productions in Backus-Naur form. The changes we have made to grammar productions can be divided into two types: additional JML-specific grammar rules and changes to the existing Java rules that specify where a JML construct can be used. This distinction is useful, because we tried to minimize changes to existing rules to avoid undesired behavioral changes to the Java subset of the grammar. The additional rules for JML are derived from the JML grammar as defined in the JML reference manual [46].

The granularity in which implementation units are translated into FSTs is a design decision that must be made for every language. Given an object-oriented language, we could decide that refinements on class level are sufficient for our needs and translate whole classes into terminal nodes that can be composed. Alternatively, classes can be treated as non-terminal nodes, which contain a child element for each field and method. In this case, methods could be translated to terminal nodes, allowing composition rules that perform on method level. It would also be possible to introduce terminal nodes at even finer levels of granularity: method definitions can be treated as non-terminals, while the list of parameters could be a terminal, allowing to specify a rule that performs refinement on the parameters of a method. An example from our implementation is that we added method specifications as a child node to method definitions to distinguish between a method and the corresponding contract.

Figure 4.4 shows an example of three grammar productions representing a method specification, a (general) specification, and a specification case sequence. The first rule is annotated with `@FSTNonTerminal()`. This means that a method specification will be translated into a non-terminal FST-node by the JML parser. Productions that are marked with `@FSTTerminal()` will be translated into terminal nodes. In this case we must specify the name of the composition rule to be used when composing this type of terminal node. We can see that specification case sequences (SpecCaseSeq) are terminal nodes which are composed following a rule called *ContractComposition*. The specified rule is automatically mapped to a Java class with the same name (in this case *ContractComposition*).

```
@FSTNonTerminal()
MethodSpecification:
 [<ALSO>] Specification
;



Specification:
@\% SpecCaseSeq @&
|
@\% SpecCaseSeq RedundantSpec @&
;

@FSTTerminal(compose="ContractComposition")
SpecCaseSeq:
SpecCase (LOOK_AHEAD(3) AlsoSpecCase )*
;
```

Figure 4.4: Grammar production representing a JML method specification

The production for method specifications in Figure 4.4 starts with the name MethodSpecification followed by a colon. A MethodSpecification starts with the keyword also followed by a Specification. The pointy brackets indicate that ALSO refers directly to a token named ALSO. The brackets around them mean that the keyword also is optional. A Specification is either just a SpecCaseSeq or a SpecCaseSeq followed by a RedundantSpec. The operator "|" marks a decision (i.e., the parser has two possibilities to continue). Another type of decision can be seen in the production SpecCaseSeq. The pattern "(production)*" means that the production can occur zero or more times at this position. The parser needs to make a decision on how often to match the production. In some cases, the parser cannot make the right decision based on the previously processed tokens alone. This leads to incorrect behavior of the parser. We can instruct the parser to use a look-ahead which instructs the parser to consider a number of additional tokens (i.e., to look ahead in the token stream) to solve this problem. The downside of using a look-ahead is that it decreases the performance of the parser.

Finally, the example in Figure 4.4 contains the directives "@%" and "@&". These are instructions for the pretty printer that indicate to insert the beginning and end of JML comments at this place. It is necessary to include such directives because the information about where a JML comment begins and ends would be lost. We extended FeatureHouse to support this in addition to existing directives that refer to whitespace characters.

4.2.2 Composition Rules

As we have explained, we need to provide composition rules in addition to the FeatureBNF grammar. These composition rules contain the implementation that performs the merging of two terminal nodes with a certain type. We provided a new composition rule for contract composition. Technically, the composition rule is defined for specification case sequences, which allows us to handle multiple specification cases as an atomic unit representing the method contract. The composition rule implements contract composition techniques as described in Chapter 3.

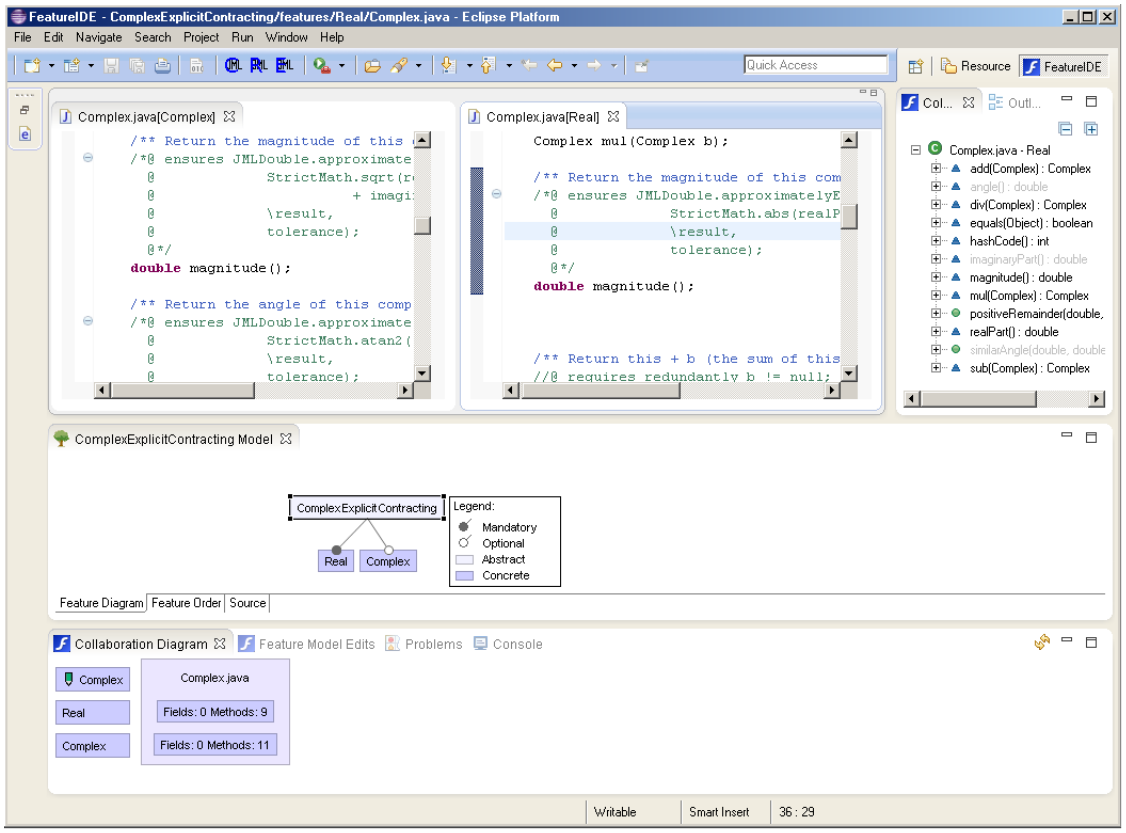
4.3 Known limitations

It was necessary to change the behavior related to Java comments in FeatureHouse. Traditionally, FeatureHouse copies Java commentaries in a feature refinement into the composed class. We had to disable this feature, because JML-comments are also Java-comments but need special treatment by our extension of FeatureHouse. Furthermore, we had to change the order in which terminal nodes are included in the composition, because the standard behavior would copy method contracts below the method definition while in JML they are usually written above the method definition.

Furthermore, Java Modeling Language allows specifications to be written inside separate files rather than directly in the Java source files. However, our implementation does not support this possibility, but could be extended.

4.4 Summary

We have developed tool support for design by contract in feature-oriented programming. Our tool supports the composition of contracts defined in the Java Modeling Language using



```
    /** Return the magnitude of this 
    /*@ ensures JMLDouble.approximate
      @             StrictMath.sqrt(r
      @                       + imagi
      @             \result,
      @             tolerance);
      @*/
    double magnitude();

    /** Return the angle of this comp
    /*@ ensures JMLDouble.approximate
      @             StrictMath.atan2(
      @             \result,
      @             tolerance);
```

```
    Complex mul(Complex b);

    /** Return the magnitude of this com
    /*@ ensures JMLDouble.approximatelyE
      @             StrictMath.abs(realP
      @             \result,
      @             tolerance);
      @*/
    double magnitude();



    /** Return this + b (the sum of this
    //@ requires redundantly b != null;
```

```
Complex.java - Real
  add(Complex) : Complex
  angle() : double
  div(Complex) : Complex
  equals(Object) : boolean
  hashCode() : int
  imaginaryPart() : double
  magnitude() : double
  mul(Complex) : Complex
  positiveRemainder(double,
  realPart() : double
  similarAngle(double, double
  sub(Complex) : Complex
```

Figure 4.5: Screenshot of FeatureIDE

different contract composition techniques. It allows the developer to combine feature-based specification with product-based verification. This approach reduces the specification effort compared to product-based specifications because we do not have to specify each product separately. Furthermore, our tool allows developers to use existing JML tools to be applied on generated products. This includes specification type-checking, runtime debugging, static analysis, and verification [16].

We integrated our tool into FeatureIDE, which provides the benefits of a sophisticated development environment and support for the whole development process including feature modeling and configuration of variants. Figure 4.5 shows a screenshot of FeatureIDE with our extension. There are several views and editors that support developers when developing software product lines.

Technically, our tool support was developed using FeatureHouse. The main artifacts that we developed is a language-definition in form of a FeatureBNF grammar and contract-specific composition rules implemented in Java. The FeatureBNF grammar conforms to language levels 0-3 of the JML Reference Manual and additionally supports Java 1.5 features such as generics.

5. Evaluation

We have defined different approaches to specify contracts in feature-oriented programming and formalized the required contract composition mechanisms. We defined three alternative mechanisms for plain contracting, three for contract overriding, and three for consecutive contract overriding. Contract overriding and consecutive contract refinement can be combined with keyword `original`, resulting in another six approaches. Furthermore, each approach can be combined with pure method refinement. In addition to the variations we have discussed, programs can also be specified completely (heavyweight specification), or only certain properties of a program can be specified (lightweight specification). This results in at least 48 approaches that can be used to specify contracts in feature-oriented programming.

Which contract specification approach should be used?

The usual answer to such a question is that it depends on the situation. What kind of product line are we developing? How many features does it contain? What is the relationship between features? What kind of properties are we specifying formally? Do we want to specify properties that must be fulfilled by a subset of all products? What is the purpose of the specification? What kind of analysis should be performed? All these questions may influence the applicability of a given specification approach.

Ideally, we would compare all possible approaches regarding their applicability for the development of different kinds of product lines. However, we face several problems:

- Feature-oriented programming is a novel paradigm and publicly available examples of its usage, such as open-source projects, are hardly available. Thus, the mining of software repositories as performed in other disciplines (e.g., by searching open source repositories) is not applicable.

- Design by contract in feature-oriented programming has not been applied for large product lines.
- The effort needed to develop realistic product lines (no toy programs) using feature-oriented programming and the Java Modeling Language is high. Limited resources do not allow us to develop sophisticated case studies with representative product lines.

We conclude, that we cannot evaluate contract composition as performed by our tool on a set of projects that is representative for the whole development paradigm. As a result, we use the following strategy:

- We have derived composition mechanisms based on informal descriptions. To validate that our definitions conform with the intended meaning for each approach we use our tool on product lines that have been developed as case studies by other researchers, in which the composition was performed manually. If our tool is capable to perform the required compositions, we gain confidence that our formalization of contract composition mechanisms conforms to their intended meaning.
- To evaluate contract composition mechanisms we have performed a set of case studies. Each case study focuses on simple characteristics that are likely to occur in most product lines.
- We focus on evaluating contract composition mechanisms, rather than specification strategies. If a composition mechanism is not applicable in a situation that is likely to occur in real product lines, we can conclude that any specification strategy that relies on this mechanism is generally not applicable. However, if a composition mechanism is applicable, we can only conclude that the use of certain specification strategies is technically possible.

Thus we validate our definitions of contract composition mechanisms by using our tool on existing case studies, in which contracts have been composed manually and evaluate the applicability of contract composition mechanisms by performing case studies. Our case studies aim to cover common characteristics of feature-oriented programming, rather than being representative for the whole paradigm. In each case study, we analyze the applicability of different contract composition mechanisms.

We present the results of the validation of our definitions of contract composition mechanisms in Section 5.1. The evaluation of the applicability of contract composition mechanisms is presented in Section 5.2, and summarize the results in Section 5.3.

5.1 Validation of contract composition mechanisms

Thüm et al. have performed five case studies to evaluate different contract specification approaches for feature-oriented programming [77]. In these case studies, the contract composition had to be performed manually because no tool support was available.

The first case study, BankAccount, is a product line that represents a bank account with typical operations on it (e.g. deposit and withdrawal) [77]. It was specified using explicit contract refinement. We have experienced, that our tool was able to perform the desired operations as expected (i.e. the results conform to the manual composition). However, only the variant of keyword `original` that references a single clause (rather than a whole case or the complete specification) has been used. Furthermore, the result of a composition in which the refining method has no contract is a method with the original contract. Thus, the underlying composition mechanism that was assumed by the developers is contract overriding without ϵ-contracts, as defined in Section 3.1.

Case study ExamDB is a product line of programs that manage the result of exams for students [77]. It was specified using pure method refinement. Thus, we validate our assumption that the underlying contract composition mechanism is usually plain contracting. We have found pure methods that are specified with contracts. These contracts were not refined. The intention is that the refinements of such contracts is not allowed. Thus, our assumption that plain contracting has been used as composition mechanism seems correct.

The third case study, DiGraph, is a product line of graph libraries. It does not contain any method refinements and is specified using plain contracting. All three definitions of plain contracting can be applied because methods are not refined.

Paycard is a case study in which a product line of smart cards that can be used for electronic money transactions has been specified using consecutive contract refinement. We did not have access to the composition results in this case. Thus, we cannot validate our tool on this case study. The same applies to case study IntList, in which we do not have access to the composition results.

To summarize, we were able to gain some confidence that our definitions of contract composition mechanisms specify the intended behavior. Our tool is capable to perform the necessary transformations, but in some cases we cannot validate the intended result. Furthermore, the use of alternative variations of a given contract composition mechanism (e.g. regarding ϵ-contracts) has not been considered in these case studies. Thus, we can only draw conclusions for certain cases:

- Pure method refinement assumes plain contracting as underlying composition mechanism.
- The underlying mechanism of explicit contract refinement is contract overriding without ϵ-contracts (with support for keyword `original`).

5.2 Evaluation of contract composition mechanisms

In this section, we give an overview of the product lines we have developed in our case studies and experiences we have made when specifying them.

Case Study NumbersSPL

For the case study NumbersSPL we extracted a product line from a given product. It is based on a Java program representing complex numbers and operations on them, that is part of the JML project and was specified using the Java Modeling Language by Gary T. Leavens.[1] We extracted a feature that only supports real numbers instead of complex numbers and use this as a base feature for our software product line.

Figure 5.1 shows the feature model of NumbersSPL. Feature *Real* is mandatory and the base feature of this product line. Feature *Complex* refines feature *Real* by adding support for complex numbers. It should be noted that the concept of complex numbers subsumes the concept of real numbers. This is why we have chosen to implement Feature *Complex* as an optional refinement rather than an alternative.

Feature *Real* introduces the interface `Numbers` including the definition and specification of eight methods and one model method (only visible inside contracts). Feature *Complex* refines six of the methods introduced by feature *Complex* and introduces one new method definition (that returns the value of the imaginary part of the complex number) and one model method.

The products defined by NumbersSPL include only specifications and no implementation. Furthermore, the amount of variability is rather low, as it contains only two products. However, it would be possible that the NumbersSPL can be a part of a product line including more

[1]http://www.eecs.ucf.edu/ leavens/JML/index.shtml

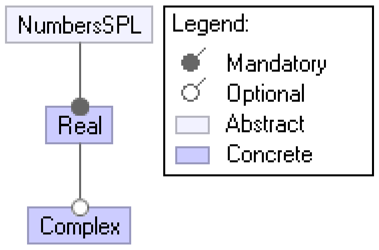


Figure 5.1: Feature model of case study NumbersSPL.

features and actual implementations. We view the NumbersSPL as an instance of an optional feature refinement rather than an example of a typical product line.

Experiences with NumbersSPL

The product line developed in case study NumbersSPL could be completely specified with contract overriding, explicit contract refinement, and consecutive contract refinement. These solutions are semantically equivalent, but differ in the lines of code used to specify method refinements. Feature *Real* is the same for all approaches, because it only includes method introductions. Feature *Complex* includes 172 lines of code when using contract overriding, 150 lines of code when using explicit contract refinement, and 145 lines of code when using consecutive contract refinement.

When using contract overriding we have to include the original contract as defined in feature *Real* in the contract refinements in feature *Complex*. An example is the refinement of method `mul(Complex b)` which multiplies the current number with b. Figure 5.2 shows the contract for this method. The contract of this method, as it is introduced in feature *Real*, only considers multiplication of real numbers. The refinement adds specification regarding the imaginary part of the complex number. However, the contract refinement must include the whole specification about the real part. As a result, the contract of method `mul(Complex b)` includes 14 lines of cloned specification and one additional line to specify the handling of the imaginary part of the complex number. Only the last line of the contract includes behavior specific to feature *complex*, the rest is a duplicate of the original contract.

We did not experience situations in which the alternative definitions of contract overriding (with different handling of empty contracts), as described in Section 3.1, would have had an influence on the specification. The reason is that there is no method that is refined more than once to generate a product in this case study.

When using explicit contract refinement rather than contract overriding it is possible to avoid the above mentioned specification clone by using keyword `original`. Figure 5.3 shows

```
/** Return this * b (the product of this and b). */ {                {Complex}
/*@   requires_redundantly b != null;
  @   requires !Double.isNaN(this.magnitude() * b.magnitude());
  @   requires !Double.isNaN(this.angle()) && !Double.isNaN(b.angle());
  @   ensures_redundantly \result != null;
  @   ensures JMLDouble.approximatelyEqualTo(
  @               this.magnitude() * b.magnitude(),
  @               \result.magnitude(),
  @               tolerance);
  @   ensures similarAngle(this.angle() + b.angle(),
  @                        \result.angle());
  @   also
  @   requires_redundantly b != null;
  @   requires Double.isNaN(this.magnitude() * b.magnitude())
  @            Double.isNaN(this.angle())  Double.isNaN(b.angle());
  @   ensures Double.isNaN(\result.realPart());
  @   ensures \result.imaginaryPart() == 0.0;
  @*/
    Complex mul(Complex b);
```

Figure 5.2: Excerpt of case study NumbersSPL specified with contract overriding

```
/** Return this * b (the product of this and b). */                  {Complex}
/*@   /original_case
  @   also
  @   requires /original;
  @   ensures /original && \result.imaginaryPart() == 0.0;
  @*/
    Complex mul(Complex b);
```

Figure 5.3: Excerpt of case study NumbersSPL specified with explicit contract refinement

the refinement of method `mul` using explicit contract overriding. We have used keyword `original_case` to refer to the first specification case (separated by keyword `also`). Furthermore, we included the precondition and postcondition of the second specification case, while adding the specification of the imaginary part to the postcondition.

We can see that it is necessary to know details about the structure of the contract subject to refinement when using keyword `original_case` because cases cannot be referenced explicitly, but have to be identified by their position (i.e. when using keyword `original_case` in the n-th specification case, it refers to the n-th specification case of the original contract). In this case, we could have avoided this by referring to the whole specification with keyword `original_spec`, but this is not always possible.

When using consecutive contract refinement, we do not need to use keyword `original` because the original contract is implicitly included in the result of composition. Figure 5.4

```
/** Return this * b (the product of this and b). */                  {Complex}
/*@ensures \result.imaginaryPart() == 0.0;
  @*/
 Complex mul(Complex b);
```

Figure 5.4: Excerpt of case study NumbersSPL specified with consecutive contract refinement

shows the refinement of method `mul(Complex b)` specified with consecutive contract refinement. This contract includes only the additional specification that is directly related to the method refinement in feature *Complex*.

The contracts in this case study include pure method calls. As an example, the contract of method `mul(Complex b)` calls the pure method `magnitude()` to express the result of multiplication of two numbers in terms of the product of both magnitudes. Method `magnitude()` is refined by feature *Complex* to calculate the correct magnitude for complex numbers. The information about how to calculate the magnitude for a number can be found at only one location, avoiding the need to refine each location at which this information is needed. Thus, the use of pure methods can reduce specification clones, just like the use of methods can reduce code clones.

The distinctive characteristic of case study NumbersSPL is that it includes an optional feature with method refinements. The optional feature extends the base feature (i.e., the original contracts are not violated). In such a case, it is feasible to use any approach that supports strengthening of the original contract. It is desirable to use an approach that enforces that contracts cannot be weakened, because this avoids possible sources of error (unintended weakening). Consecutive contract refinement is the only approach that provides this property. However, when pure method refinement is allowed, these pure methods must be specified with a contract or it is possible to change the specification in an arbitrary way. When a pure method is specified with a contract, this contract must be refined according to the rules of consecutive contract refinement. Thus, the properties of consecutive contract refinement are transfered to the use of pure method refinement.

Plain contracting, as expected, cannot be used to specify this product line in which contracts are subject to refinement.

Case Study IntegerSetSPL

IntegerSetSPL is a case study, in which we followed the same approach as in case study NumbersSPL. We started from a program specified with the Java Modeling Language. The program, in this case, is also part of the JML project and was developed by Gary T. Leavens.

Figure 5.5 shows the feature model of the IntegerSetSPL. The base feature *IntegerSet* contains public methods `insert()`, `remove()`, and `isMember()` and their specifications. The implementation is provided by one of the alternative child features resulting in two possible products. The public interface of class `IntegerSet` including their contracts is defined in feature *IntegerSet* and should not be violated in any product.

Feature *TreeSet* and feature *HashSet* are alternative features. They refine method `IntegerSet` by providing the actual implementation. The main difference between this case study and NumbersSPL is that the represented characteristic is the use of alternative features rather than an optional feature.

Experiences with IntegerSetSPL

We were able to use all specification approaches, but it was not necessary to refine contracts. Our first observation is that for each method that can be specified using plain contracting, it is also possible to use consecutive contract refinement. The reason is that consecutive contract refinement subsumes plain contracting if method refinements are not specified with contracts.

Furthermore, we observed that it is possible to specify feature-specific behavior with plain contracting in our case study. This is realized by defining a contract for each method that must be implemented by one or more feature refinements. This specification cannot be refined, but it is possible to specify additional methods with new contracts.

This is the first case study in which we explicitly had to answer the question whether we should use contract overriding with or without ϵ-contracts, as introduced in Section 3.1. We experienced that the former forces the developer to specify each method refinement with a full specification clone of the original contract, while the latter does not override the original contract if the refinement has no contract.

Case Study UnionFindSPL

The next case study, UnionFindSPL, is a product line in which each product describes a variation of the Union-Find algorithm [70]. In the previously described case studies, we have

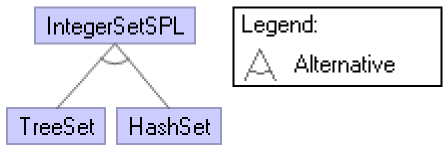


Figure 5.5: Feature model of case study IntegerSetSPL.

used existing programs specified with contracts and derived a product line. In this case study, we developed both implementation and specification.

All products in the UnionFindSPL implement the same operations:

- UnionFind(int n) - a constructor to initialize a data structure of size n,
- void union(int p, int q) - a method to connect elements p and q,
- boolean connected(int p, int q) - returns true if p and q are in the same component (i.e., if p and q are connected),
- int find(int p) - returns the identifier of the component for p
- int count() - counts the number of components

Figure 5.6 shows the feature model of the union find product line. Feature *UnionFind* is mandatory and introduces the class `UnionFind` which includes the above mentioned operations. For the methods `connected` and `find`, it only contains the specification and no implementation. The implementation of both methods is alternatively provided by feature *QuickFind* or feature *QuickUnion*. These implementations differ in details of the underlying data structure, resulting in different performance of operations. Feature *QuickUnion* can optionally be refined by feature *Weighted*, which modifies the algorithm in a way that the underlying data structure is always a balanced tree. In this case, a comparator is needed, that is provided by one of the alternative features *CompareBySize* or *CompareByHeight*.

Furthermore, feature *Weighted* can be refined by feature *PathCompression*, which adds support for path compression (i.e. each node is directly connected with the root of its component). Finally, feature *PathCompression* can be refined by feature *WithHalving*, which is a special case of path compression in which the length of any traversed path is reduced.

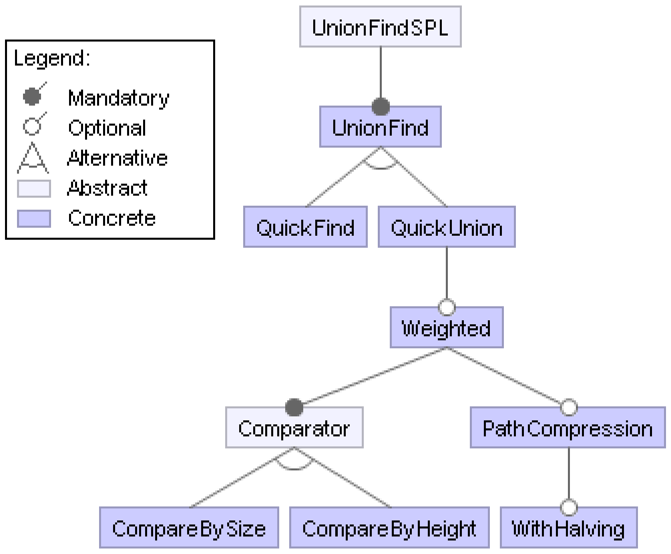


Figure 5.6: Feature model of case study UnionFindSPL.

Experiences with UnionFindSPL

We were not able to specify the product line using plain contracting. It would have been possible to specify it in a way abstracting from the different properties of products, but restricting on a specification of the common properties.

We could specify the product line using contract overriding, consecutive contract refinement, and explicit contract refinement. In this case, there was no significant difference in the lines of code for each of these variants. While contract overriding introduced some specification clones, they mostly contained only few lines of code resulting in no signification increase in the overall lines of code. Similarly, explicit contract refinement introduces some additional uses of keyword `original`.

We were also able to specify the product line using pure method refinement (based on plain contracting), but we experienced that we did not take advantage of the fine granularity of refinements provided by pure method refinement, because we introduced one pure method for each contract subject to refinement.

Case Study PokerSPL

In case study PokerSPL, we extracted features from an existing open-source project in order to develop a product line. The project is a poker system specified with contracts in JML. Figure 5.7 shows the feature model of our product line. We extracted 10 features. The only mandatory features are feature *Player* and feature *Table*. Feature *Cards* contains additional

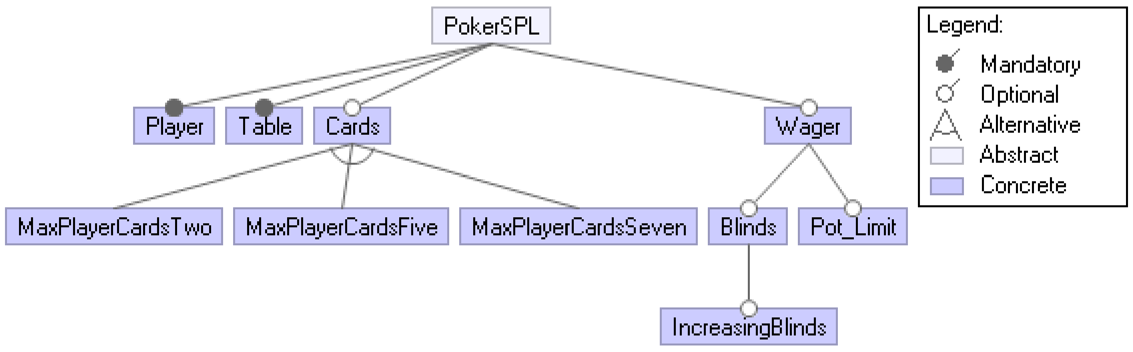


Figure 5.7: Feature model of case study PokerSPL.

functionality for poker cards. The maximum number of cards a player can hold at a time is refined by the alternative child features of feature *Cards*. Additionally, optional feature *Wager* allows players to bet money while playing. The amount of money that can be bet in one turn can optionally be limited to the amount that is currently in the pot by feature *PotLimit*. A minimum amount of a wager can be introduced by feature *Blinds*. This amount can optionally be increased automatically between rounds.

Experiences with PokerSPL

We were able to specify the product line using contract overriding, explicit contract refinement and pure method refinement (with plain contracting). The use of plain contracting (without pure method refinement), does not allow us to refine methods. However, it would have been possible to decide that we are not interested in specifying feature-specific properties. In this case, plain contracting can always be used.

When using explicit contract refinement, we used keyword `original` in two contracts. When using contract overriding, these method contracts must explicitly express the specifications of the method subject to refinement. Thus, they include specification clones.

In this case study, we have experienced an advantage of pure method refinement. When using pure method refinements, we could specify properties such as the maximum number of cards using pure method calls. These pure methods are refined in other features. When using the other approaches, we rather used the field in which the maximum number of cards is saved for the implementation. Thus, pure method refinements can provide a decoupling of implementation and specification. Consider the case of an error in the implementation of the field that defines the maximum number of cards. This error cannot be found by verification, because both the specification and implementation contain the same error. In this case, the coupling between specification and implementation is a disadvantage.

	NumbersSPL	IntegerSetSPL	UnionFindSPL	PokerSPL
Plain Contracting	no	yes	no	no
Contract Overriding	yes	yes	yes	yes
Explicit Contract Ref.	yes	yes	yes	yes
Consecutive Contract Ref.	yes	yes	yes	no
Pure Method Ref.	yes	yes	yes	yes

Table 5.1: Overview of case studies and contract specification approaches that could be used to specify them. For pure method refinement we assume plain contracting as underlying composition mechanism. For all other approaches we assume that pure method refinement is not allowed.

5.3 Summary

We performed four case studies in which we developed software product lines and examined which specification approaches can be used to specify them with contracts. We have considered cases, in which the product line was developed using different strategies. Three product lines, namely NumbersSPL, IntegerSetSPL and PokerSPL are based on existing single products specified with the Java Modeling Language. We developed and specified UnionFindSPL based on the descriptions of different algorithm variations.

Table 5.1 summarizes which approach could be used to specify the product lines in our case studies. Plain contracting (without pure method refinement) could be used to specify only one of four case studies. Contract overriding and explicit contract refinement could be used for all four case studies. Consecutive contract refinement could be used specify three of four case studies. Pure method refinement based on plain contracting could be used for all case studies, but in only one case study we used the fine granularity of specifications provided by this approach.

Plain contracting seems to be applicable in less situations than the other approaches. However, we doubt that this result can be generalized. The question whether plain contracting is appropriate is more a design decision rather than an intrinsic property of a given product line. It is always possible to decide that we only want to specify the behavior that is common to all products. However, in our case studies we mostly assumed that we wanted to specify feature-specific behavior. Thus, the observation that we could not use plain contracting is not surprising. Generally, plain contracting can be used for any product line in which we want to specify common behavior rather then feature-specific behavior. However, as we have seen in case study IntegerSetSPL, it is still possible to specify feature-specific behavior if it is expressed in method introductions rather then refinements.

Contract overriding was applicable in all our case studies. However, in all cases it would have been better to use explicit contract refinement or consecutive contract refinement to avoid specification clones. Additionally, we did not experience any benefits that the use of contract overriding could provide compared to these alternative approaches. Furthermore, we expect that larger product lines in which a given method is refined multiple times by optional features would not even allow the use of contract overriding, because the required variability cannot be expressed in the contracts.

Explicit contract refinement was also applicable in all case studies. When plain contracting was also a possible alternative the use of contract overriding did not provide any advantage, because method refinements were not specified with contracts. In each case study, explicit contract refinement could be used instead of contract overriding, while avoiding specification clones by using keyword `original`.

Consecutive contract refinement could be used in three of four case studies. It is more restrictive than explicit contract refinement. This restrictiveness is the reason why it cannot be applied in some cases. However, when it could be applied the specification is shorter, because we do not have to include any explicit references to keyword `original`. Furthermore, it ensures that contracts are not accidentally weakened.

Pure method refinement could be used to specify three of the four case studies. In case study NumbersSPL pure method refinements could not be used, because the case study contains only contracts and no implementation. However, the contracts of pure methods had to be refined, which means that the implementation would have to be refined, too. The only case in which we experienced pure method refinement to be superior to other approaches was the case study PokerSPL, in which it provided a convenient way to specify properties of fine granularity.

The product lines we have used to evaluate specification approaches contain some typical characteristics of feature-oriented programming. However, we cannot generalize the results to larger product lines because we expect that large product lines used in practice may impose requirements on specification approaches that cannot be predicted.

As far as our tool is concerned, we have validated each product of the product lines using a type checker for the Java Modeling Language. If our composition mechanism generated invalid products, either the product line or our composition tool contains an error. In fact, we have used this approach to debug the specification while developing the case studies. This is an example of product-based analysis, that our tool promised to support. However, we have not evaluated larger case studies, in which the generation of all products might not be possible. In the case of larger product lines it might be useful, to generate a subset of products based on heuristics (e.g. pair-wise combinations [61] [17]).

When comparing our tool with the manual composition of contracts, we experienced that the effort needed to specify the product line could be reduced. This advantage increases with the number of products in a product line. In the case study NumbersSPL, our tool did not reduce the number of methods that need to be specified, because the number of features and products were equal. Furthermore, composition of contracts is a task that is error-prone when performed manually.

However, when automating this task, it becomes necessary to define the used composition mechanisms precisely, as we have done in Section 3.1. This is both an advantage and disadvantage. It is an advantage because it is possible to rely on certain characteristics of specifications when composing them (e.g., in the case of plain contracting or consecutive contract refinement). However, it is also an disadvantage because it does not allow the developer to decide on details of each composition separately.

6. Related Work

In this thesis, we used method contracts in feature-oriented programming to formally specify software product lines. Many applications related to the verification of software product lines require formal specifications. As a result, multiple approaches to specify software product lines have been proposed by researchers. Thüm et al. distinguishes between product-based, feature-based, family-based, and global specification approaches [73]. We give an overview of existing work on specification of software product lines as far as they are related to this thesis.

6.1 Specifications in Product-line Engineering

Feature-based Specification

The application of design by contract to feature-oriented programming, as used in this thesis, is a feature-based specification approach. Feature-based specification refers to approaches that specify each feature separately [73]. Apel et al. use feature-based specifications for model checking [9]. Another application for feature-based specification is the composition of proofs to verify software product lines [76]. The used specification technique is similar to contract overriding. Cumulated contract refinement has been used in a specification approach for the automated detection of feature interactions [69]. These works focus on the applications that rely on specifications rather than the specification approaches itself. Thüm et al. moves the specification itself in the focus of attention by proposing design by contract as a specification approach for feature-oriented programming and considering alternative approaches [77]. This thesis can be seen as building directly on top of this work, by providing tool support and formalizing contract composition techniques.

Product-based Specification

Another approach to specify software product lines is to specify each product individually [73]. An advantage of this approach is that no product-line specific specification techniques must be considered. Specifying each product separately can be performed by using conventional specification approaches. However, product-based specification does not scale well, because the number of product generally grows exponentially to the number of features [73].

Global Specification

Another approach that can be performed using conventional specification approaches is to define a specification that must be fulfilled by every single product of the product line [73]. In the case of our running example, the franking machine product line, we could specify that each product must properly weight, frank, and seal envelopes. However, we could not specify any properties that are not shared by every product. This approach has been considered in the domain of medical devices [50], and for verification of product lines using model checking [22] [39] [49]. Plain contracting, as defined in this thesis, is a feature-based specification approach. However, it could be used to specify a global specification if the developer takes into account that it does not enforce a global specification automatically in the case of optional features introducing method contracts. The developer has to ensure that alternative method introductions cannot introduce different contracts.

Family-based Specification

We have seen that a global specification does not take into account the variability of a product line. By using the knowledge about variability in a product line it is possible to derive a specification with variable parts that depend on the selection of specific features or feature combinations [73].

Contracts in Delta-oriented programming

Delta-oriented programming is a programming paradigm in which delta modules are composed to generate products of a product line. The main difference to feature-oriented programming is that delta modules can also remove classes, fields and methods. Contracts has been considered as a means to specify delta modules. Bruns et al. use the Java Modeling Language to specify delta modules [15]. A core module is specified with contracts as usual. The delta language that is used to define delta modules is extended to support operations such as adding and removing contracts. Hähnle et al. use a special syntax to specify delta modules. They restrict the possible changes performed by a delta module to enforce a Liskov principle (analogously to behavioral subtyping) for delta-oriented programming. The goal is to allow modular reasoning, similar to consecutive contract refinement as discussed in this thesis.

6.2 Contracts in Single-System Engineering

We have discussed different approaches to specify software product lines. However, contracts are not only used to specify software product lines. Contracts, as a special kind of assertion, can be seen as a general concept independently from software product lines.

Assertions as a means to specify properties that must hold at certain states of program execution are a traditional concept in software development. The topic of this thesis, the application of method contracts to feature-oriented programming as suggested by Thüm et al., blends in with prior work that suggests ways of applying the principle of assertions to novel development paradigms. Meyer proposes design by contract which applies the principle of assertions on the level of classes and methods to object-oriented programming[56]. Several extensions of object-oriented programming that introduce new forms of abstractions raise the question how method contracts can be applied to them.

Contracts for Components

Component-oriented software development is a development methodology. It introduces the notion of components that encapsulate multiple classes [60]. The vision of this development paradigm is that developers provide a marketplace of independently developed components that can be used to assemble larger programs. In order to enable correct reuse of components they must provide a well-defined interface that defines how this component can be used and the expected behavior. Contracts have been proposed by several researchers to specify the interface of components [51] [24]. The main difference to contract composition, as used in this thesis, is that components can only be aggregated and not refine each other. Features, unlike components, are not self-containing units of implementation. Thus, the composition of contracts as discussed in this thesis has not been considered in component-oriented software development.

Contracts in Aspect-Oriented Programming

Aspect oriented programming, introduces aspects as abstractions, orthogonal to classes. Aspect-orientation subsumes feature-oriented programming and provides extended abilities in specifying the way in which an aspect is composed with the rest of the program [7]. Design by contract has been suggested to specify aspects [52] [2] [71] [84] [40]. Contracts have also been suggested for the detection of behavioral conflicts in aspect-oriented programming [28]. While being a very related approach, research in aspect-oriented programming does not take variability into account (i.e., the absence of aspects is usually not considered).

Contracts in Context-Oriented Programming

Context-oriented programming is a paradigm in which runtime-variability is achieved by assigning functionality to different layers [29]. The goal is to provide means to change implementation depending on the current context of the software (e.g., mobile devices must adapt to the available services in the current environment). Layers have been used to achieve variability in design by contract by assigning contracts to certain layers (features) [30]. Composition of contracts has not been considered in this research area.

Composition of Specifications

The terminology of composition and refinement of specifications is also used in research work about specification theories. However, the meaning of these terms in this context differs from the one used in this thesis. The composition of specifications refers to the aggregation of specifications rather than the more general sense discussed in this thesis [1]. The refinement of contracts usually expresses a relationship between contracts, which can be viewed as one contract being a sub-contract of another contract. Contrary, in feature-oriented programming composition is not limited to aggregation.

7. Conclusion

The application of design by contract to feature-oriented programming has been proposed as a means to specify features in software product lines [77] [76] [69] [75]. In feature oriented-programming, programs are assembled by composing classes and methods. Recent work raised awareness to the question of how methods can be specified with contracts [77].

Existing descriptions of possible contract specification approaches for software product lines are formulated informally. Thus, we identified possible interpretations and give precise definitions of these approaches. For each specification approach, we distinguish between the strategy developers must follow to specify features (i.e., the contract specification strategy) and the mechanism that is used by tools to compose two contracts during feature composition (i.e., the contract composition mechanism). We give formal definitions for contract composition mechanisms, which provided several insights:

- Plain contracting is a contract specification approach, in which the idea is that every method in a software product line is specified only once. Our formalization of its underlying composition mechanism made clear, that this property cannot be expressed in terms of a feature composition operator alone. The developer must be aware that in the case of alternative method introductions (i.e., a method may be introduced by different features) the resulting contract of that method may vary between products.

- Consecutive contract refinement is a contract specification approach, in which the composition of two contracts is expected to fulfill both the original contract and the new contract. Here, our formalization also made clear that this desired property cannot be achieved by any resulting contract. However, we proposed three alternative formalizations (with different semantics) and discussed their advantages and disadvantages.

- We have to consider a number of alternative variations for each specification approach. For plain contracting, we have defined three possible contract composition mechanisms that differ in the handling of methods without contract. Analogously, we have defined three variants of contract overriding. For consecutive contract refinement, we have defined three alternative contract composition mechanisms that differ in their restrictiveness regarding preserved properties.

- Associativity is a generally desired property for feature composition that provides tools more freedom for feature composition [8]. The formalization of contract composition mechanisms allows us to reason about associativity for each approach. This provides useful information about the applicability of these approaches. We identified that contract overriding, as used in existing work, is not associative. We proposed alternative definitions that are associative. The difference of these alternative composition mechanism is the handling of methods without contract. When using contract overriding, researchers can now choose the appropriate mechanism depending on whether associativity is required. We proposed similar modifications for plain contracting. Plain contracting can either be associative or allow method refinements to introduce contracts if the method subject to refinement has no contract.

Several applications that make use of contracts in feature-oriented programming have been proposed by researchers. However, the actual composition of contracts had to be performed manually or with rather limited tool support. This approach to composition is error-prone and laborious.

We developed tool support that allows the developer to express contracts in Java Modeling Language in feature-oriented programming and to compose contracts. Thus, existing tools for JML can be used off-the-shelf for testing or verifying products.

For our tool support, we developed a FeatureBNF grammar that supports Java 1.5 and the Java Modeling Language (Levels 0-3) as defined in the JML reference manual [46]. We used FeatureHouse to generate a parser based on this grammar. Furthermore, we provide contract-aware rules for feature composition supporting different composition mechanisms. It is possible to choose between different contract composition mechanisms: plain contracting, contract overriding, consecutive contract refinement. These mechanisms can be combined with the use of keyword `original` and pure method refinement. The combination of contract overriding and keyword `original` is known as explicit contract refinement. Consecutive contract refinement with support for keyword `original` has not been considered yet, but is possible. However, plain contracting does not allow contract refinement. Accordingly, it cannot be combined with the use of keyword `original`.

We integrated our tool into FeatureIDE, a framework for feature-oriented development of software product lines. We expect that the integration of our tool support into FeatureIDE makes it easier for other researchers and developers to use our tool for case studies and applications that require the specification of feature modules. Furthermore, our tool allows the developer to combine feature-based specification with product-based verification. By specifying features rather than products the specification effort is relative low. Furthermore, it is possible to use existing tool support for JML on products generated by our tool. Thus, it is possible to apply specification type-checking, runtime debugging, static analysis, and verification for each product. We refer to Burdy et al. for an overview of existing JML tools [16].

Furthermore, we developed four case studies to evaluate our tool. As far as our tool is concerned, we have experienced that the effort to specify product lines can be reduced compared to manual composition. We made extensive use of the possibility to type-check the generated specification of the products in our case studies. This serves as an example and proof-of-concept of product-based analysis in which the specification effort is reduced by feature-based specification.

We have experienced that the applicability of plain contracting is rather a design decision than a property of a given product line. Developers can simply decide to specify only properties that are common to all products and use plain contracting. However, if the developer decides that feature-specific specifications are desired, plain contracting is generally not applicable. Our results suggest that contract overriding is in all cases inferior to explicit contract refinement. However, explicit contract refinement makes it necessary to extend the specification language to support keyword `original`. Consecutive contract refinement is more restrictive than explicit contract refinement which can be both an advantage and disadvantage. It allows the developer and tools to rely on certain properties of contracts. Method callers do not have to be aware of contracts in all features. It is sufficient to consider only one feature. However, it cannot be used in situations that require to break these properties.

The results of this thesis provide several benefits for researchers. Our definitions of contract specification approaches allow developers to describe the underlying contract composition mechanism precisely by distinguishing specification strategy and composition mechanism explicitly. This distinction makes clear that certain properties cannot be ensured by a composition mechanism alone, but have to be considered by the developers. We have proposed several variations for each composition mechanism and described their advantages and disadvantages. Furthermore, we provide tool support that can be used to compare different specification approaches by developing software product lines in which contracts are expressed in the Java Modeling Language. Our tool simplifies the development of case studies by automating contract composition.

8. Future Work

This thesis aims to provide a foundation for future research related to the application of method contracts in feature-oriented programming. We defined contract specification strategies and proposed several modifications to existing approaches. We formalized possible contract composition mechanisms that can be used for feature oriented programming. In future work, it should be investigated whether and how our formalization can be applied to related paradigms, namely aspect-oriented programming and delta-oriented programming.

In this thesis, we have focused on method contracts. However, the handling of invariants must be considered in future work.

A special case of pure methods are model methods, which are only visible for the specification. The refinements of model methods should be considered. These approaches need to be compared to the contract specification techniques used in this thesis.

Our comparison of contract specification techniques for feature modules has focused on expressiveness. Thüm et al. also takes the usability of approaches into account. We think it is desirable to investigate the human factor of specifications even more. Empirical studies are needed to investigate the influence on the developer by different approaches. Are specifications easily understandable? Is it easy to recognize errors? Investigating such questions, could provide useful insights related to the applicability of specification approaches.

The Java Modeling Language enforces specification inheritance (i.e., subclasses inherit the specification of their superclass). Thus, specifications can be changed by changing the class hierarchy. It should be investigated how this concept relates to contract-aware feature composition.

We developed tool support that can be used for contracts expressed in the Java Modeling Language in feature-oriented programming. In future work, we plan to provide several extensions to our tool. We integrated it into FeatureIDE, which allows the use of several useful editors and views that help the developer when using our tool. However, we plan to extend FeatureIDE by several views for design by contract in feature-oriented programming. These views can provide additional information such as statistics about the usage of contracts. How many features contain contracts? How many features refine contracts? So far, such questions have to be answered manually by exploring the source code. We expect that it will be helpful to automate the generation of such information.

Our tool support is also intended to serve as the basis for applications related to the verification of product lines. In this thesis we focused on the feature-based generation of product specifications. Other verification approaches, such as the generation of a meta program from a specified product line may impose unique requirements on specification of features. Future work should take the requirements for novel verification techniques for the specification into account.

Furthermore, our tool support allows to perform type-checking of contracts for generated products. In future work, type-checking approaches for software product lines that take the variability into account should be extended to support contracts.

Another possible extension of our work is to support choosing between composition methods for each method rather than for each product line. This could be realized by additional keywords in the specification language. This possibility gives the developer more freedom when specifying product lines and may lead to novel combinations of existing approaches with possibly synergistic effects.

So far, we have hardly discussed design by contract as a development method rather then only a means to specify functionality for automated analysis. Design by contract as a development method means that the developer when designing a system only relies on properties stated in contracts rather than implementation. It should be investigated in further detail, whether the traditional benefits of design by contract as a development method provides the same advantages (e.g. modular reasoning) for feature-oriented programming as it promises for object-oriented programming.

To answer the question whether a certain methodology such as design by contract in feature-oriented programming provides a real benefit for developers, it is not sufficient to show whether it is possible to use it but it needs to be shown that it is efficient. But the practical usefulness of a tool, is always depended on the way it is used. If the result is not as desired it could be that either the tool is bad, or the tool is used in a wrong way.

Thus, we believe it is desirable to gather practical knowledge about the application of contracts in feature-oriented programming in form of best practices, similar to code-conventions, and design patterns in object-oriented programming, for design by contract in feature-oriented programming. We believe, that this is a topic that should be addressed for feature-oriented software development in general. Thus, we believe it is desirable to share existing product lines with other researchers by making them publicly available. SPL2go is a repository for feature-oriented software product lines. All product lines we have developed in our case studies are available on SPL2go[1], a repository for feature-oriented software product lines. We hope that researchers will increasingly share case studies and product lines in order to gather a large collection of product lines for future research.

[1]spl2go.cs.ovgu.de

Bibliography

[1] Martín Abadi and Leslie Lamport. Composing specifications. *ACM Trans. Program. Lang. Syst.*, 15(1):73–132, January 1993. (cited on Page 88)

[2] Sérgio Agostinho, Ana Moreira, and Pedro Guerreiro. Contracts for Aspect-Oriented Design. In *Proceedings of the Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT)*, pages 1:1–1:6. ACM, 2008. (cited on Page 88)

[3] Sven Apel and Christian Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology (JOT)*, 8(5):49–84, 2009. (cited on Page 3)

[4] Sven Apel, Christian Kästner, and Christian Lengauer. FeatureHouse: Language-Independent, Automated Software Composition. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 221–231. IEEE Computer Science, 2009. (cited on Page 10, 17, and 63)

[5] Sven Apel, Christian Kästner, and Christian Lengauer. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Transactions on Software Engineering (TSE)*, 2012. (cited on Page 4, 5, 11, 17, 60, and 61)

[6] Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 125–140. Springer, 2005. (cited on Page 4)

[7] Sven Apel, Thomas Leich, and Gunter Saake. Aspectual Feature Modules. *IEEE Transactions on Software Engineering (TSE)*, 34(2):162–180, 2008. (cited on Page 87)

[8] Sven Apel, Christian Lengauer, Don Batory, Bernhard Möller, and Christian Kästner. An Algebra for Feature-Oriented Software Development. Technical Report MIP-0706,

Department of Informatics and Mathematics, University of Passau, 2007. (cited on Page 11, 13, 20, and 90)

[9] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. Detection of Feature Interactions using Feature-Aware Verification. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 372–375. IEEE Computer Science, 2011. (cited on Page 85)

[10] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. Feature-Aware Verification. Technical Report MIP-1105, University of Passau, Germany, 2011. (cited on Page 3)

[11] Don Batory. 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)*, pages 3–35. Springer, 2006. (cited on Page 4)

[12] Don Batory, Jacob N. Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6):355–371, 2004. (cited on Page 10)

[13] Grd Berry. Synchronous design and verification of critical embedded systems using scade and esterel. In Stefan Leue and Pedro Merino, editors, *Formal Methods for Industrial Critical Systems*, volume 4916 of *Lecture Notes in Computer Science*, pages 2–2. Springer Berlin / Heidelberg, 2008. (cited on Page 1)

[14] Danilo Beuche. *Composition and Construction of Embedded Software Families*. PhD thesis, University of Magdeburg, Germany, 2003. (cited on Page 7 and 8)

[15] Daniel Bruns, Vladimir Klebanov, and Ina Schaefer. Verification of Software Product Lines with Delta-Oriented Slicing. In *Proceedings of the International Conference on Formal Verification of Object-Oriented Software (FoVeOOS)*, pages 61–75. Springer, 2011. (cited on Page 86)

[16] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An Overview of JML Tools and Applications. 7(3):212–232, 2005. (cited on Page 4, 15, 68, and 91)

[17] Muffy Calder and Alice Miller. Feature Interaction Detection by Pairwise Analysis of LTL Properties—A Case Study. *Formal Methods in System Design*, 28(3):213–261, 2006. (cited on Page 84)

[18] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In *Proceedings of the International Symposium on Formal Methods for Components and Objects (FMCO)*, pages 342–363. Springer, 2005. (cited on Page 15)

[19] Lori A. Clarke and David S. Rosenblum. A historical perspective on runtime assertion checking in software development. *SIGSOFT Softw. Eng. Notes*, 31(3):25–37, May 2006. (cited on Page 14, 17, and 40)

[20] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, USA, 2001. (cited on Page 3 and 7)

[21] Edsger W. Dijkstra. Structured programming. In *Software Engineering Techniques*. NATO Science Committee, August 1970. (cited on Page 1)

[22] Kathi Fisler and Shriram Krishnamurthi. Modular Verification of Collaboration-based Software Designs. In *Proceedings of the European Software Engineering Conference/-Foundations of Software Engineering (ESECFSE)*, pages 152–163. ACM, 2001. (cited on Page 86)

[23] Kathi Fisler and Shriram Krishnamurthi. Decomposing Verification Around End-User Features. In *Proceedings of the IFIP Working Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*, pages 74–81. Springer, 2005. (cited on Page 3)

[24] G. Florin, D. Enselme, and F. Legond-Aubry. Design by contracts: analysis of hidden dependencies in component based applications. *J. of Objects Technology*, 2004. note. (cited on Page 87)

[25] Robert W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967. (cited on Page 1)

[26] Jeremy Gibbons. Formal methods: Why should i care? - the development of the t800 transputer floating-point unit. In *In Proc. 13th New Zealand Computer Society Conference*, pages 207–217, 1993. (cited on Page 1)

[27] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, Amsterdam, 2005. (cited on Page 61)

[28] Chengwan He and Zheng Li. Automatic detection to the behavioral conflict in aop application based on design by contract. *Journal of Software*, 6(11), 2011. (cited on Page 88)

[29] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology, March-April 2008, ETH Zurich*, 7(3):125–151, 2008. (cited on Page 88)

[30] Robert Hirschfeld, Michael Perscheid, Christian Schubert, and Malte Appeltauer. Dynamic Contract Layers. pages 2169–2175. ACM, 2010. (cited on Page 88)

[31] Tony Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, January 2003. (cited on Page 1 and 14)

[32] Jean-Marc Jézéquel and Bertrand Meyer. Design by Contract: The Lessons of Ariane. 30(1):129–130, 1997. (cited on Page 14 and 17)

[33] Victoria Stavridou Jonathan P. Bowen. Safety-critical systems, formal methods and standards. *Software Engineering Journal*, 8:189–209, 1993. (cited on Page 1)

[34] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990. (cited on Page 7 and 8)

[35] Kyo C. Kang, Jaejoon Lee, and Patrick Donohoe. Feature-Oriented Product Line Engineering. *IEEE Software*, 19(4):58–65, 2002. (cited on Page 3)

[36] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 311–320. ACM, 2008. (cited on Page 10)

[37] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 327–354. Springer, 2001. (cited on Page 60)

[38] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242. Springer, 1997. (cited on Page 60)

[39] Tomoji Kishi and Natsuko Noda. Formal Verification and Software Product Lines. *Communications of the ACM*, 49:73–77, 2006. (cited on Page 86)

[40] Herbert Klaeren, Elke Pulvermueller, Awais Rashid, and Andreas Speck. Aspect composition applying the design by contract principle. In *Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering-Revised Papers*, GCSE '00, pages 57–69, London, UK, UK, 2001. Springer-Verlag. (cited on Page 88)

[41] Peter Gorm Larsen, John Fitzgerald, and Tom Brookes. Applying formal specification in industry. *IEEE Softw.*, 13(3):48–56, May 1996. (cited on Page 1)

[42] Gary T. Leavens. Jml's rich, inherited specifications for behavioral subtypes. In *ICFEM*, pages 2–34, 2006. (cited on Page 52)

[43] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *Software Engineering Notes (SEN)*, 31(3):1–38, 2006. (cited on Page 15)

[44] Gary T. Leavens and Yoonsik Cheon. Design by Contract with JML, 2005. (cited on Page 15)

[45] Gary T. Leavens and Curtis Clifton. Lessons from the jml project. In Bertrand Meyer and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, volume 4171 of *Lecture Notes in Computer Science*, pages 134–143. Springer, 2005. (cited on Page 15)

[46] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Mller, Joseph Kiniry, and Patrice Chalin. Jml reference manual, 2008. Available from http://www.jmlspecs.org/. (cited on Page 61, 63, 65, and 90)

[47] John A. Lewis, Sallie M. Henry, Dennis G. Kafura, and Robert S. Schulman. An empirical study of the object-oriented paradigm and software reuse. *SIGPLAN Not.*, 26(11):184–196, 1991. (cited on Page 2)

[48] Barbara H. Liskov and Jeannette M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994. (cited on Page 43)

[49] Jing Liu, Samik Basu, and Robyn Lutz. Compositional Model Checking of Software Product Lines using Variation Point Obligations. *Automated Software Engineering (ASE)*, 18(1):39–76, 2011. (cited on Page 86)

[50] Jing Liu, Josh Dehlinger, and Robyn Lutz. Safety Analysis of Software Product Lines using State-based Modeling. *Journal of Systems and Software (JSS)*, 80(11):1879–1892, 2007. (cited on Page 86)

[51] Yi Liu and H. Conrad Cunningham. Software component specification using design by contract. In *Proceeding of the SouthEast Software Engineering Conference, Tennessee Valley Chapter, National Defense Industry Association*, 2002. (cited on Page 87)

[52] David H. Lorenz and Therapon Skotiniotis. Extending Design by Contract for Aspect-Oriented Programming. *Computing Research Repository (CoRR)*, abs/cs/0501070, 2005. (cited on Page 88)

[53] John McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962. (cited on Page 1)

[54] M. D. Mcilroy. Mass Produced Software Components. Technical report, NATO, 1969. (cited on Page 2)

[55] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988. (cited on Page 2)

[56] Bertrand Meyer. Applying Design by Contract. 25(10):40–51, 1992. (cited on Page 3, 14, 15, 17, and 87)

[57] Steven P. Miller. The industrial use of formal methods: Was darwin right? In *WIFT'98*, pages 74–74, 1998. (cited on Page 1)

[58] F. L. Morris and C. B. Jones. An early program proof by Alan Turing. 6(2):139–143, April/June 1984. (cited on Page 13)

[59] Munge Development Team. Munge: A Purposely-Simple Java Preprocessor. Website, 2011. Available online at http://github.com/sonatype/munge-maven-plugin; visited on January 11th, 2011. (cited on Page 60)

[60] Oscar Nierstrasz, Simon Gibbs, Dennis Tsichritzis, and Universit? De Gen??ve. Component-oriented software development. *Communications of the ACM*, 35:160–165, 1992. (cited on Page 87)

[61] Sebastian Oster, Florian Markert, and Philipp Ritter. Automated Incremental Pairwise Testing of Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 196–210. Springer, 2010. (cited on Page 84)

[62] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972. (cited on Page 1)

[63] D. L. Parnas. On the design and development of program families. *IEEE Trans. Softw. Eng.*, 2(1):1–9, January 1976. (cited on Page 10)

[64] David Lorge Parnas. Software aspects of strategic defense systems. *Commun. ACM*, 28(12):1326–1335, December 1985. (cited on Page 1)

[65] Jörg Pleumann, Omry Yadan, and Erik Wetterberg. Antenna: An Ant-to-End Solution For Wireless Java. Website, 2011. Available online at http://antenna.sourceforge.net/; visited on November 22nd, 2011. (cited on Page 60)

[66] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering : Foundations, Principles and Techniques*. Springer, 2005. (cited on Page 3, 7, and 8)

[67] Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer, 1997. (cited on Page 3, 10, and 60)

[68] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-Oriented Programming of Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 77–91. Springer, 2010. (cited on Page 60)

[69] Wolfgang Scholz, Thomas Thüm, Sven Apel, and Christian Lengauer. Automatic Detection of Feature Interactions using the Java Modeling Language: An Experience Report. In *Proceedings of the International SPLC Workshop Feature-Oriented Software Development (FOSD)*, pages 7:1–7:8. ACM, 2011. (cited on Page 4, 18, 40, 85, and 89)

[70] Robert Sedgewick. *Algorithms*. Addison-Wesley, 1983. (cited on Page 78)

[71] Therapon Skotiniotis and David H. Lorenz. Cona: Aspects for Contracts and Contracts for Aspects. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 196–197. ACM, 2004. (cited on Page 88)

[72] Susan Stepney, David Cooper, and Jim Woodcock. An Electronic Purse: Specification, Refinement, and Proof. Technical monograph PRG-126, Oxford University Computing Laboratory Programming Research Group, July 2000. (cited on Page 1)

[73] Thomas Thüm, Sven Apel, Christian Kästner, Martin Kuhlemann, Ina Schaefer, and Gunter Saake. Analysis Strategies for Software Product Lines. Technical Report FIN-004-2012, School of Computer Science, University of Magdeburg, Germany, 2012. (cited on Page 4, 18, 20, 21, 85, and 86)

[74] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming (SCP)*, 2012. (cited on Page 5 and 59)

[75] Thomas Thüm, Ina Schaefer, Sven Apel, and Martin Hentschel. Family-Based Deductive Verification of Software Product Lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, 2012. (cited on Page 18 and 89)

[76] Thomas Thüm, Ina Schaefer, Martin Kuhlemann, and Sven Apel. Proof Composition for Deductive Verification of Software Product Lines. In *Proceedings of the International Workshop on Variability-intensive Systems Testing, Validation and Verification (VAST)*, pages 270–277. IEEE Computer Science, 2011. (cited on Page 18, 85, and 89)

[77] Thomas Thüm, Ina Schaefer, Martin Kuhlemann, Sven Apel, and Gunter Saake. Applying Design by Contract to Feature-Oriented Programming. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 255–269. Springer, 2012. (cited on Page 4, 18, 21, 23, 30, 36, 38, 46, 49, 73, 85, and 89)

[78] Jan Tretmans, Klaas Wijbrans, and Michel Chaudron. Software engineering with formal methods: The development of a storm surge barrier control system - revisiting seven myths of formal methods, 2001. (cited on Page 1)

[79] Alan Turing. Checking a Large Routine. In *Conference on High Speed Automatic Calculating Machines*, pages 67–69, 1949. (cited on Page 13 and 14)

[80] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, 2007. (cited on Page 10)

[81] David M. Weiss. The Product Line Hall of Fame. In *Proceedings of the International Software Product Line Conference (SPLC)*, page 395. IEEE Computer Science, 2008. (cited on Page 3)

[82] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4):19:1–19:36, October 2009. (cited on Page 1)

[83] Jim Woodcock, Susan Stepney, David Cooper, John Clark, and Jeremy Jacob. The certification of the mondex electronic purse to itsec level e6. *Form. Asp. Comput.*, 20(1):5–19, December 2007. (cited on Page 1)

[84] Jianjun Zhao and Martin C. Rinard. Pipa: A Behavioral Interface Specification Language for AspectJ. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 150–165. Springer, 2003. (cited on Page 88)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den