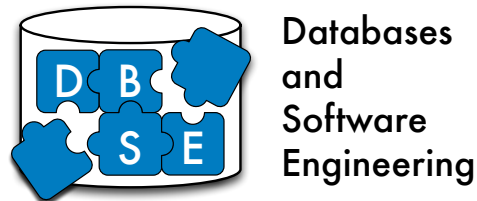University of Magdeburg

School of Computer Science

Dissertation

Product-Line Specification and Verification with Feature-Oriented Contracts

Author:
Thomas Thüm

February 23, 2015

Reviewers:

Prof. Dr. Gunter Saake (University of Magdeburg, Germany)

Prof. Don Batory, Ph.D. (University of Texas at Austin, USA)

Prof. Dr. Reiner Hähnle (University of Darmstadt, Germany)

Product–Line Specification and Verification
with Feature–Oriented Contracts

DISSERTATION

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.–Ing.)

angenommen durch die Fakultät für Informatik
der Otto–von–Guericke–Universität Magdeburg

von Dipl.–Inform. Thomas Thüm

geb. am 10.09.1984   in Magdeburg

Gutachterinnen/Gutachter

Prof. Dr. Gunter Saake
Prof. Don Batory, Ph.D.
Prof. Dr. Reiner Hähnle

Magdeburg, den 23.02.2015

Abstract

Variability is ubiquitous in today's software development. While techniques to efficiently implement software product lines are used for decades, verification techniques have been a hot research topic in the last years. We give an overview on how existing verification techniques were applied to product lines. Based on our insights, we overcome two shortcomings of previous research on product-line verification. First, we systematically investigate how contracts can be utilized for product-line specification. Our theoretical discussion and practical evaluation lead us to the notion of feature-oriented contracts. In particular, we found that behavioral subtyping applies to most, but not all feature-oriented contracts. Second, we use these feature-oriented contracts to compare different verification techniques and strategies for the same product-line implementation and specification. We measured synergistic effects when using theorem proving and model checking for product-line verification in concert.

Zusammenfassung

Variabilität ist allgegenwärtig in der heutigen Softwareentwicklung. Während Techniken zur effizienten Implementierung von Software-Produktlinien seit Jahrzehnten eingesetzt werden, waren Verifikationstechniken ein Forschungsschwerpunkt in den letzten Jahren. Wir geben eine Übersicht darüber, wie existierende Verifikationstechniken auf Produktlinien angewandt worden. Mithilfe unserer Erkenntnisse beheben wir zwei Defizite früherer Forschungsarbeiten zur Produktlinienverifikation. Einerseits untersuchen wir systematisch wie Kontrakte zur Produktlinienspezifikation genutzt werden können. Unsere theoretische Diskussion und praktische Evaluierung führt uns zum Konzept der Feature-orientierten Kontrakte. Insbesondere haben wir beobachtet, dass viele aber nicht alle Feature-orientierten Kontrakte dem liskovschen Substitutionsprinzip folgen. Anderseits nutzen wir Feature-orientierte Kontrakte zum Vergleich verschiedener Verifikationstechniken und -strategien für die selbe Produktlinienimplementierung und -spezifikation. Wir konnten Synergien bei der Kombination von Theorembeweisern und Modellprüfern für die Produktlinienverifikation messen.

Acknowledgements

Contents

List of Figures

List of Tables

List of Code Listings

1. Introduction

A major part of our today's software is not developed from scratch, but rather by starting from existing software [Antkiewicz et al., 2014; Hemel and Koschke, 2012; Laguna and Crespo, 2013; Rubin and Chechik, 2013; Xue et al., 2012]. Conflicting requirements, such as functional requirements and given resource restrictions, force developers to build variants of a software. Nevertheless, these software variants have similarities. In software-product-line engineering, these similarities are modeled explicitly by means of features [Clements and Northrop, 2001; Pohl et al., 2005]. Instances of generative programming can be utilized to automatically generate each product of a software product line only by providing a selection of features [Apel et al., 2013a; Batory et al., 2004; Czarnecki and Eisenecker, 2000]. A typical example for a software product line is an operating system [Parnas, 1976], such as the Linux kernel, but software product lines have been used in various domains [Weiss, 2008] to increase return on investment, shorten time to market, and improve software quality [Clements and Northrop, 2001; Lutz, 2007; van der Linden et al., 2007].

Software-product-line engineering is increasingly used for safety-critical and mission-critical systems, including embedded, medical, automotive, and avionic systems [Weiss, 2008]. Existing verification techniques, such as theorem proving, model checking, and type checking, can be applied to a software product line by verifying each product individually. However, due to the similarities, this strategy involves redundant effort. Especially if products are generated automatically, generation and verification of each product individually is often infeasible [Liebig et al., 2013], as the number of products is up-to exponential in the number of features. More efficient strategies have been proposed in separate research areas, first for model checking [Fisler and Krishnamurthi, 2001; Nelson et al., 2001], then for type checking [Aversano et al., 2002] and theorem proving [Poppleton, 2007]. Since then, many new approaches have been presented for each verification technique with a different nomenclature each, making it hard to understand their differences and hindering their systematic application by practitioners and researchers.

In a survey of the research literature on product-line verification, we identified two major shortcomings of current proposals. First, the focus of the literature is on verification techniques and their scalability, whereas specification techniques are often not justified empirically. Most verification techniques, such as theorem proving and model checking, require to specify the expected behavior, which is then used during the analysis of the actually implemented or modeled behavior [Clarke et al., 1999; Schumann, 2001; Smith, 1985]. While specification techniques need to be applied to product lines as well, their application is typically not scrutinized. Second, even though each verification technique has its strengths and weaknesses, a combination has not yet been used to verify the same specification for a given product line. In particular, some verification techniques may be superior to others in certain situations (e.g., in an earlier development step, in which defects are more likely to occur).

Our long-term vision is that given a software product line, we can recommend verification techniques and strategies based on static properties, such as the product-line's size, the cohesion of each feature's artifacts, and the feature specifications. This thesis has three major contributions, each being a stepping stone towards this long-term vision:

- In Chapter 3, we propose a classification of product-line analyses. Our classification identifies the strategy to scale an existing verification technique to software product lines, as well as the underlying strategies for specification and implementation. We survey the research literature on product-line verification and classify a corpus of 137 articles. Based on our insights, we infer a research agenda to guide future research on product-line verification.

- In Chapter 4, we systematically discuss how to specify product lines, whereas our discussion is based on method contracts (i.e., preconditions and postconditions) for specification and feature-oriented programming for implementation. We propose a taxonomy and mechanisms for the composition of feature-oriented contracts and discuss fundamental properties of contract composition. By means of case studies, we evaluate which contract-composition mechanisms are superior to others and which compositions are typical in practice.

- In Chapter 5, we present and evaluate different verification techniques and strategies for feature-oriented contracts. With proof composition, we propose to decompose proof scripts along with source code and specification. With variability encoding, we show how to verify the same product line by means of theorem proving and model checking. In our evaluation, we found that a combination of several tools can improve efficiency and effectiveness at the same time.

Besides these three main chapters, we give a brief introduction to contracts and software product lines in Chapter 2. We conclude our thesis and discuss future work in Chapter 6.

2. Background

This chapter shares material with the ACM Computing Surveys article "A Classification and Survey of Analysis Strategies for Software Product Lines" [Thüm et al., 2014a] and with the Science of Computer Programming article "FeatureIDE: An Extensible Framework for Feature-Oriented Software Development" [Thüm et al., 2014b].

The goal of this chapter is to give a brief introduction into the two main topics of the thesis. We introduce contracts as a means to formally specify and verify behavior in Section 2.1. Then, we present main concepts of software product lines in Section 2.2.

2.1 Specification and Verification with Contracts

We give a historical perspective on contracts and their ingredients in Section 2.1.1. In Section 2.1.2, we discuss the relation of contracts in the presence of subtyping. Finally, we give an overview on techniques for program verification that are based on contracts in Section 2.1.3.

2.1.1 Design by Contract

Contracts are almost as old as programming languages, but have been known under different terms in the first steps. Turing [1949] argued that we need assertions to support programmers in reasoning about correctness of large routines. [Floyd, 1967] proposed assertions to be used for verification by means of mechanical theorem proving. Some of these assertions have later been named preconditions [Hoare, 1969, 1972] and postconditions [Dijkstra, 1976; Gries, 1981]. We found the first usage of the term *contract* by Liskov and Guttag [1986], which has been promoted by Meyer [1988, 1992] for object-oriented programming in Eiffel as *design by contract*. Other popular languages with

```
class Graph {
  //@ invariant nodes != null && edges != null;
  Collection<Node> nodes = new ArrayList<Node>();
  Collection<Edge> edges = new ArrayList<Edge>();
  /*@ requires node != null;
    @ assignable nodes;
    @ ensures nodes.contains(node) && (\forall Node n; n != node;
    @   \old(nodes.contains(n)) <==> nodes.contains(n)); @*/
  void addNode(Node node) {
    nodes.add(node);
  }
  /*@ requires nodes.contains(edge.first) && nodes.contains(edge.second);
    @ assignable edges;
    @ ensures hasEdge(edge) && (\forall Edge e; e != edge;
    @   \old(hasEdge(e)) <==> hasEdge(e)); @*/
  void addEdge(Edge edge) {
    edges.add(edge);
  }
  /*@ pure @*/
  boolean hasEdge(Edge edge) {
    for (Edge e : edges)
      if (e.first == edge.first && e.second == edge.second)
        return true;
    return false;
  }
}
```

Listing 2.1: A graph implementation with contracts and invariants in JML.

support for contracts are the Java Modeling Language (JML) [Burdy et al., 2005] and Spec# [Barnett et al., 2011].

Technically, contracts can be viewed as an extension of type specifications [Meyer, 1988]. That is, contracts extend the usual type signatures to include constraints on behavior to capture behavioral dependencies between objects [Helm et al., 1990]. With contracts, programmers can make implicit assumptions about possible input and output values of object-oriented methods explicit [Meyer, 1988]. Explicitly stating these assumptions can improve program understanding and reuse [Helm et al., 1990] and avoid defensive programming [Meyer, 1992]. Hatcliff et al. [2012] give an overview on further applications, such as verifying implementations against specifications as well as the generation of test cases and test oracles.

Example 2.1. *In Listing 2.1, we give an example of contracts in JML, which is an extension of Java with support for contracts [Leavens et al., 2006]. The class* Graph *stores nodes and edges. Both, nodes and edges can be added by means of the methods* addNode *and* addEdge*. The preconditions of these methods denoted by the* requires *clause*

state that only initialized objects may be passed. Furthermore, an edge can only be added if it connects nodes that have been added before. The postconditions of both methods are denoted by ensures *and state that the new node or edge is added and that no further node or edge, respectively, has been added or removed. Keyword* old *can be used in postconditions to refer to the state prior to method execution. The quantifiers* forall *and* exists *may be used to reason about sets of objects.*

Hoare [1972] proposed two types of invariants for correctness proofs of data representations, which are also known as class invariants and loop invariants [Meyer, 1988]. A loop invariant is an assertion that is assumed to hold before the first loop execution and after each execution. Loop invariants are often needed to prove termination. In contrast, a class invariant is an assertion that needs to be established by the constructor, and every method can rely on the invariant when it is called, but need to establish the invariant on return [Hoare, 1972]. That is, a class invariant (for short *invariant*) defines the legal values of its type [Liskov and Wing, 1994]. We give a trivial invariant in Listing 2.1 stating that the fields nodes and edges must be initialized.[1] Hence, the method addNode does neither need to handle the case nodes == null in the method body, nor does it need to state this as a precondition.

Instead of introducing a postcondition \old(f) == f for each field f of a class that is not changed by the method, one can add a frame condition [Hatcliff et al., 2012]. Frame conditions are also known as modifies clauses [Dhara and Leavens, 1996; Leino, 1998; Liskov and Guttag, 1986] and *assignable clauses* [Beckert et al., 2007; Chalin et al., 2005; Leavens and Müller, 2007]. Assignable clauses avoid unnecessary long postconditions, but are also essential for information hiding [Leavens and Müller, 2007] and formal verification [Beckert et al., 2007; Weiß, 2011]. In our example, method addNode is only allowed to change field nodes.

Preconditions, postconditions, and invariants can contain calls to methods that have no side-effects and terminate. In JML, such methods need to be marked as pure (cf. method hasEdge in Listing 2.1), which is a shorthand for assignable \nothing; diverges false [Beckert et al., 2007]. Keyword diverges is followed by a logical expression and the semantics of diverges is that non-termination is only allowed if the expression evaluates to true. If no assignable clause or diverges clause is given for a non-pure method, the default assignable \everything and diverges true is assumed, respectively [Beckert et al., 2007; Chalin et al., 2005]. A further type of methods that may be called from within contracts only is known as model method [Hatcliff et al., 2012]. Such methods are completely defined in JML comments and are invisible to the implementation [Beckert et al., 2007].

2.1.2   Behavioral Subtyping

In the presence of subtyping, contracts of a type and its subtypes should be in a special relation for several reasons. First, "objects of the subtype ought to behave the

---

[1]In recent versions of JML, non-null is assumed as the default for fields and parameters. However, in this thesis we make this attribute explicit to avoid ambiguities.

same as those of the supertype as far as anyone or any program using supertype objects can tell" [Liskov and Wing, 1994]. In particular, preventing type errors due to incompatible interfaces is not enough [Liskov and Wing, 1994]. Second, to enable modular specification and verification, we should avoid respecification and reverification in subtypes [Dhara and Leavens, 1996]. Adding a new subtype whose objects behave like supertype objects does not require reverification of types expecting supertype objects [Dhara and Leavens, 1996].

Meyer [1988] introduced the concept of subcontracting as a central idea of object orientation. America [1991] coined the term behavioral subtyping in the context of type systems, which has been adopted by others [Dhara and Leavens, 1996; Hatcliff et al., 2012; Liskov and Wing, 1994]. *Behavioral subtyping* is an extension of structural subtyping. That is, syntactical constraints of structural subtyping ensure that a supertype expression can be replaced by subtype expression without causing type errors [Dhara and Leavens, 1996]. Avoiding type errors includes contravariance of arguments (same number of arguments, types may be replaced by supertypes) and covariance of result (type may be replaced by subtype) [Liskov and Wing, 1994]. In addition, behavioral subtyping introduces semantic constraints to ensure that replacing supertype objects by subtype objects does not produce any unexpected behavior [Dhara and Leavens, 1996].

Meyer [1988] proposes that each redeclaration of a method in a subclass must fulfill original assertions and formulated the *assertion redeclaration rule* (a.k.a. methods rule [Liskov and Wing, 1994]). It states that preconditions in a redeclaration must be weaker or equal to the original precondition and that postconditions must be stronger or equal to the original postcondition [Meyer, 1988]. A sound mathematical definition of weaker and stronger based on implications can be found elsewhere [America, 1991; Liskov and Wing, 1994]. As checking the assertion redeclaration rule requires theorem proving [Meyer, 1988] and is infeasible for compilers [America, 1991], Meyer [1988] proposes a simple language rule that uses a disjunction for preconditions and a conjunction for postconditions instead. Given a superclass contract $c = \{\phi\}m\{\psi\}$ and a subclass contract $c' = \{\phi'\}m'\{\psi'\}$ in Hoare triples [Hoare, 1969], the subclass method m' needs to establish the contract $c'' = \{\phi \vee \phi'\}m'\{\psi \wedge \psi'\}$. With *specification inheritance*, Dhara and Leavens [1996] relaxed this rule such that $c'' = \{\phi \vee \phi'\}m'\{(old(\phi) \Rightarrow \psi) \wedge (old(\phi') \Rightarrow \psi')\}$, in which *old* evaluates a predicate as it would be evaluated before method execution. The advantage of specification inheritance over the assertion redeclaration rule is that a method needs to establish only those postconditions to which the according precondition has been fulfilled by the caller.

Beyond preconditions and postconditions, Meyer [1988] proposes that subclass methods have to establish all superclass invariants. Liskov and Wing [1994] formalized this expectation in the *invariant rule* stating that the subtype invariant implies the supertype invariant. In contrast, there is no consensus how to deal with assignable clauses. Dhara and Leavens [1996] argue that assignable clauses of subclass and superclass methods should be concatenated, whereas Leino [1998] proposes that assignable clauses may only be refined indirectly by means of data groups.

2.1.3   Contract-Based Verification

In the previous sections, we introduced how to specify the intended behavior of programs by means of contracts. In the following, we discuss several techniques to verify programs. Most of these techniques rely on some kind of specification, which can be given by means of contracts. We briefly discuss strengths and limitations of each verification technique and argue that a wide variety of techniques is needed to increase the quality of software. We distinguish techniques based on commonly used terms, even there are no clear distinctions between them and they can all be defined as some form of abstract interpretation [Cousot and Cousot, 1977].

Type Checking

A *type system* is a syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute [Pierce, 2002]. Type systems can be used to syntactically classify programs into well-typed and ill-typed programs, based on a set of inference rules. *Type checking* refers to the process of analyzing whether a program is well-typed according to a certain type system defined for a particular programming language. A *type checker* is the actual tool analyzing programs written in a certain language, usually part of a compiler or linker [Pierce, 2002]. In model-driven development, type checking is essentially the analysis of well-formedness of a model with respect to its meta-model [Atkinson and Kühne, 2003].

Type checking can detect type errors, such as incompatible type casts, dangling method references, and duplicate class names. For instance, a dangling method reference occurs if a method with a certain signature is called that has not been declared. Other examples are that a programmer may have misspelled the name of a method, or that the number of arguments is not correct.

A type system can be seen as a formal specification that all programs written in a certain language must conform to. Pierce [2002] argues that, in principle, types can be created to check arbitrary specifications. However, in practice, type systems are defined once for each language and not for each program. Hence, type checking is used to verify specifications that are common to all programs in a certain language. The focus of type systems is to automatically detect faults, but type systems are usually limited in the faults they can detect [Liskov and Wing, 1994]. In particular, they cannot check contracts defined in first-order logic [America, 1991]. Nevertheless, type checkers are typically included in compilers and scale to large programs. In this sense, type checking is a necessary step before verifying programs with respect to contracts.

Static Analysis

The term *static analysis* (a.k.a. program analysis) refers to analyses that operate at compile-time and approximate the set of values or behaviors arising dynamically at runtime when executing a program [Nielson et al., 2010]. Examples for static analyses

are traditional data-flow and control-flow analyses, but also alias analyses, program slicing, and constraint-based analyses [Muchnick, 1997; Nielson et al., 2010; Weiser, 1981]. A key technique in static analysis is that the undecidability of program termination due to loops or recursion is handled using approximation [Nielson et al., 2010].

Originally, static analyses have been used for compiler optimizations [Muchnick, 1997; Nielson et al., 2010] and debugging [Weiser, 1981]; a more recent application is program verification [Nielson et al., 2010]. For example, a static analysis is able to find accesses to uninitialized memory regions or variables. Some static-analysis tools operate on source code (e.g., LINT for C [Darwin, 1986]), others on byte code (e.g., FINDBUGS for Java byte code [Hovemeyer and Pugh, 2004]). Static analyses are either integrated into compilers such as CLANG [Lattner, 2008] or implemented in the form of dedicated tools such as FINDBUGS [Hovemeyer and Pugh, 2004].

Similar to type checking, static analyses run automatically. The difference to type checking is that not every behavioral property of interest can be encoded with types. Instead, they can be encoded by means of contracts. The difference with verification techniques, such as model checking or theorem proving, is that branches in programs are typically not interpreted and values are approximated.

Model Checking

Model checking is an automatic technique for formal verification. Essentially, it verifies that a given formal model of a system satisfies its specification [Clarke et al., 1999]. While early work concentrated on abstract system models or models of hardware, recently, software systems, such as C or Java programs, came into focus in software model checking [Beyer and Keremoglu, 2011; Visser et al., 2000]. Often, specifications are concerned with safety or liveness properties, such as the absence of deadlocks and race conditions, but also application-specific requirements can be formulated. To solve a model-checking problem algorithmically, both the system model and the specification must be formulated in a precise formal language.

A model checker is a tool that performs a model-checking task given a system to verify and its specification. Some model checkers require models with dedicated input languages for this task (e.g., Promela in SPIN [Holzmann, 1997], CMU SMV in NuSMV [Cimatti et al., 1999]), others extract models directly from source code (e.g., C in BLAST [Beyer et al., 2007] or CPACHECKER [Beyer and Keremoglu, 2011], Java in JPF [Visser et al., 2000]). After encoding a model-checking problem into the model checker's input language, the model-checking task is fully automated; each property is either stated valid, or a counterexample is provided. The counterexample helps the user to identify the source of invalidity. The most severe practical limitation of model checkers is the limited size of the state space they can handle [Schumann, 2001] (e.g., they may run out of time or main memory).

Model checking usually requires a model of the program input, which is not needed for type checking and static analyses. In addition, model checking usually scales only

to much smaller programs than type checking and static analyses. Avoiding the state-space explosion requires manual effort for system abstraction or to configure heuristics of model checkers. Nevertheless, model checking can uncover faults that type checking and static analyses can not. In software model checking, contracts can be utilized by translating them into runtime assertions or by generating test cases [Beckert and Hähnle, 2014; Hatcliff et al., 2012].

Theorem Proving

Theorem proving is a deductive approach to prove the validity of logical formulas. A theorem prover is a tool processing logical formulas by applying inference rules to them [Schumann, 2001]; it assists the programmer in verifying the correctness of formulas, which can be achieved interactively or automatically. Interactive theorem provers, such as COQ [Bertot and Castéran, 2004], PVS [Owre et al., 1992], and ISABELLE/HOL [Nipkow et al., 2002], require the user to write commands applying inference rules. Instead, automated theorem provers, such as PROVER9,[2] SPASS [Weidenbach et al., 2009], and SIMPLIFY [Detlefs et al., 2005], try to evaluate the validity of theorems without further assistance by the user.

Theorem provers usually provide a language to express logical formulas (theorems). Additionally, interactive theorem provers also need to provide a language for proof commands. Automated theorem provers are often limited to first-order logic or subsets thereof, whereas interactive theorem provers are available for higher-order logic and typed logic. Theorem provers are able to generate proof scripts containing deductive reasoning that can be inspected by humans.

Theorem provers are used in many applications, because of their high expressiveness and generality. In this thesis, we only discuss theorem provers in the context of program verification. Given a specification in some formal language and an implementation, a verification tool generates theorems, which are the input for the theorem prover. If a theorem cannot be proved, theorem provers point to the part of the theorem that could not be proved.

Compared to other verification techniques, the main disadvantage of theorem proving is that experts with an education in logical reasoning and considerable experience are needed [Clarke et al., 1999]. Contrary to type checking and static analysis, model checking and theorem proving often do not scale to large programs. When specifications are given in form of contracts, the program is typically translated into logic by means of the weakest-precondition calculus [Dijkstra, 1976; Gries, 1981].

2.2   Software Product Lines

McIlroy [1968], Dijkstra [1972], and Parnas [1976] proposed program families as a means to reduce the cost for the development and maintenance of similar programs as well as

---

[2]http://www.cs.unm.edu/~mccune/prover9/

their verification: A *program family* is a set of "programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members" [Parnas, 1976]. McIlroy [1968] proposed to implement components as interchangeable parts to avoid redundant implementations. For performance reasons, Parnas [1976] suggested to use generators for program generation instead of implementing runtime variability.

Today, the term program family is almost completely replaced by the term software product line, even though the fundamental ideas are similar. Bass et al. [1998] and Clements and Northrop [2001] define 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". A *feature* is an end-user-visible behavior or characteristic of a software system that is used to communicate commonalities and differences between the products of a product line [Apel et al., 2013a]. An example for a market segment is database management for embedded systems. A product line for that domain could have features such as multi-user support, transaction management, and recovery.

In Section 2.2.1, we give a brief introduction to feature modeling, which is used to specify the valid combinations of features for a particular product line. In Section 2.2.2, we exemplify feature-oriented programming as an implementation technique for software product lines. Finally, we give an overview on domain engineering and application engineering in Section 2.2.3, which are the main development phases in product-line engineering.

2.2.1   Feature Modeling

We distinguish the products of a software product line by means of features, but not all combinations of features are valid. For instance, a product line may have support for different platforms, such as Linux and Windows, but it is not allowed to choose several platforms for the same product. A *feature model* can be used to define the features of a product line and their valid combinations [Batory, 2005]. A *feature diagram* is a graphical representation of a feature model and defines a hierarchy between features, whereas the selection of a feature implies the selection of its parent feature [Kang et al., 1990]. Each feature may have child features that are either *optional*, *mandatory*, or belong to a group. Common group types are *alternative* (exactly one of the children needs to be selected) and *or* (at least one of the children needs to be selected) [Czarnecki and Eisenecker, 2000]. A feature model may also have *cross-tree constraints* (i.e., propositional formulas over the set of features) to define dependencies which cannot be expressed otherwise.

Example 2.2. *In Figure 2.1a, we give an example feature model for a simple object store consisting of three features. Feature* SingleStore *implements an object store that can hold a single object, including methods for read and write access. Feature* Multi-Store *implements a more sophisticated object store that can hold multiple objects, again*

(a) Feature diagram

$P_1 = \{SingleStore\}$
$P_2 = \{SingleStore, AccessControl\}$
$P_3 = \{MultiStore\}$
$P_4 = \{MultiStore, AccessControl\}$

(b) Enumeration of all valid combinations

$$Store \wedge (Store \Rightarrow Type) \wedge (Type \vee AccessControl \Rightarrow Store) \wedge$$
$$(Type \Leftrightarrow SingleStore \vee MultiStore) \wedge (\neg SingleStore \vee \neg MultiStore)$$

(c) Propositional formula
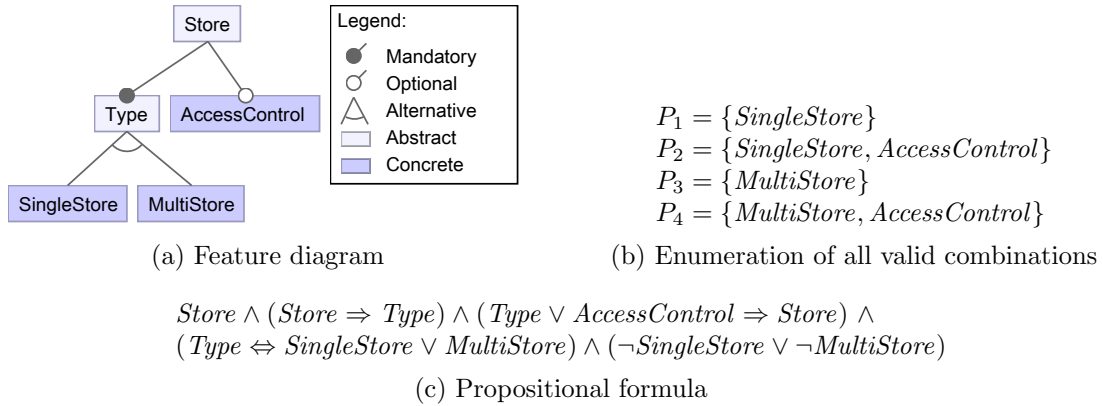
Figure 2.1: Feature model of an object store in three alternative representations [Thüm et al., 2014a].

including corresponding methods for read and write access. Feature AccessControl *provides a basic access-control mechanism that allows a client to seal and unseal the store, and thus to control access to stored objects. In our example, each object store either stores a single object (feature* SingleStore*) or several (feature* MultiStore*). Furthermore, an object store may have the optional feature* AccessControl*.*

A further representation of feature models besides feature diagrams is based on sets. That is, we can enumerate all valid selections of features. A valid selection of features is called *configuration* and can be expressed as a set of features (i.e., those features that are selected). A feature model is then specified by a set of features F and a *set of configurations* C with $C \subseteq 2^F$. In our object store example, there are four configurations that we enumerate in Figure 2.1b. In this representation, abstract features (i.e., features without mapping to artifacts) are usually omitted, as they have no influence on generated products [Thüm et al., 2011a]. A problem with this representation is that it is inefficient and even infeasible for larger product lines.

A *propositional formula* is a more efficient representation of feature models, which is often used for the analysis of feature models [Benavides et al., 2010]. A feature diagram can be automatically translated into a propositional formula [Batory, 2005]. A boolean variable is used for each feature, and the propositional formula evaluates to true, if and only if the selection of features is valid. Every relation between a feature and its child features is translated into a propositional formula, which are then conjoined to a large formula representing the whole feature model [Batory, 2005]. We give a propositional formula for the object store in Figure 2.1c.

2.2.2 Feature-Oriented Programming

An implementation technique for software product lines establishes a mapping between features as defined in the feature model and artifacts. Simply mapping features to classes is often not sufficient, because features are typically cross-cutting to classes [Tarr et al., 1999]. In this thesis, we focus on feature-oriented programming for product-line

implementation, which has its roots in mixins and collaboration-based designs. A *mixin* is a "subclass definition that may be applied to different superclasses to create a related family of modified classes" [Bracha and Cook, 1990]. In particular, a mixin can add fields and methods to an existing class and override existing methods [Flatt et al., 1998]. The difference to multiple inheritance is that we can also apply a mixin to one superclass and another superclass separately.

In *collaboration-based design*, an application does not only consist of objects and classes, but also of collaborations and roles [Smaragdakis and Batory, 2002; VanHilst and Notkin, 1996]. A class can be modularized into several roles, whereas each role is also part of a different collaboration. A collaboration consists of several roles that contribute to a different class each. At composition time, a class hierarchy is build from all roles defined for that class [VanHilst and Notkin, 1996].

Similarly, Prehofer [1997] proposed *feature-oriented programming* as an extension to object-oriented programming. Classes are decomposed into feature modules each implementing a certain feature. A *feature module* consists of classes and class refinements, whereas a *class refinement* can introduce additional fields and methods to a given class, and refine existing methods. A *method refinement* overrides the original method, while also allowing the developer to reference the original method body with keyword original. A method and all its method refinements build up a *refinement chain*, which is based on a total order of the feature modules. Given a certain configuration, a program can be generated automatically by composing the feature modules of the selected features with superimposition [Apel et al., 2013b; Batory et al., 2004].

Example 2.3. *In Listing 2.2, we show the implementation of the three features of the object store using feature-oriented programming. Feature module* SingleStore *introduces a class* Store *that implements the simple object store. Analogously, feature module* MultiStore *introduces an alternative class* Store *that implements a more sophisticated object store. Feature module* AccessControl *refines class* Store *by introducing a field* sealed, *which represents the accessibility status of a store, and by extending the methods* read *and* set *to control access. The keyword* original *is used to refer from the overriding method to the overridden method.*

Once a user has selected a list of desired features, a composer generates the final product. In our example, we use the tool FEATUREHOUSE *[Apel et al., 2013b] for the composition of the feature modules that correspond to the selected features. Essentially, the composer assembles all classes and all class refinements of the features modules being composed. Similar to subclassing, class refinement allows the programmer to override or extend existing methods. While the features* SingleStore *and* MultiStore *introduce only regular Java classes, feature* AccessControl *refines an existing class by adding new members. The result of the composition of the feature modules* MultiStore *and* AccessControl *is shown in Listing 2.3.*

In feature-oriented programming, each program is the result of an incremental composition process, in which feature modules refine other feature modules. In this thesis,

```
class Store {                                    feature module SingleStore
  private Object value;
  Object read() { return value; }
  void set(Object nvalue) { value = nvalue; }
}
```

```
class Store {                                     feature module MultiStore
  private LinkedList values = new LinkedList();
  Object read() { return values.getFirst(); }
  Object[] readAll() { return values.toArray(); }
  void set(Object nvalue) { values.addFirst(nvalue); }
}
```

```
class Store {                                   feature module AccessControl
  private boolean sealed = false;
  Object read() {
    if (!sealed) { return original(); }
    else { throw new RuntimeException("Access denied!"); }
  }
  void set(Object nvalue) {
    if (!sealed) { original(nvalue); }
    else { throw new RuntimeException("Access denied!"); }
  }
}
```

Listing 2.2: A feature-oriented implementation of an object store [Thüm et al., 2014a].

```
class Store {                              {MultiStore, AccessControl}
  private LinkedList values = new LinkedList();
  private boolean sealed = false;
  Object read() {
    if (!sealed) { return values.getFirst(); }
    else { throw new RuntimeException("Access denied!"); }
  }
  Object[] readAll() { return values.toArray(); }
  void set(Object nvalue) {
    if (!sealed) { values.addFirst(nvalue); }
    else { throw new RuntimeException("Access denied!"); }
  }
}
```

Listing 2.3: An object store composed for features *MultiStore* and *AccessControl* [Thüm et al., 2014a].
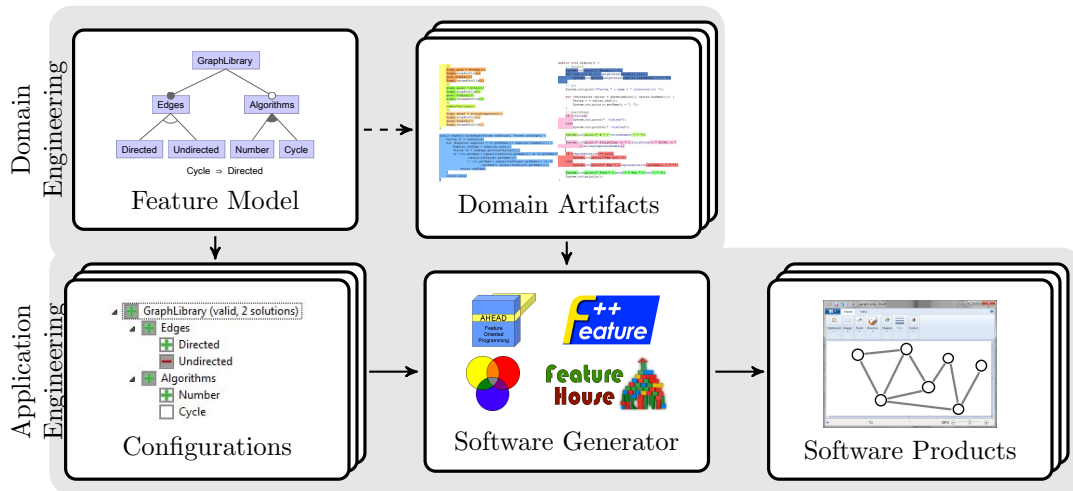
Figure 2.2: Simplified overview on domain engineering and application engineering [Thüm et al., 2014a].

we rely on an existing formalization of feature modules and their composition [Apel et al., 2010b]. Feature-module composition is formalized based on a binary composition operator $\bullet$ with $F'' = F' \bullet F$ on the set of features. The composition operator can be overloaded to express the composition of specific artifacts such as methods: $m'' = m' \bullet m$. In general, feature composition is not commutative [Apel et al., 2010b]. Thus, we distinguish between the original class or method that is subject to refinement, and the class or method refinement.

Feature-oriented programming was initially introduced for Java and intended as an extension for object-oriented programming [Prehofer, 1997]. With the *principle of uniformity*, Batory [2006] and Apel et al. [2013b] applied feature-oriented programming also to software artifacts which are not object-oriented. The principle states that all artifacts building up a software system should be composed in the same way including documentations, specifications, or models.

2.2.3 Domain Engineering and Application Engineering

In Figure 2.2, we illustrate the processes of domain engineering and application engineering, both central to the development of software product lines. For our purposes, a simplified illustration is sufficient and more elaborate description can be found elsewhere [Clements and Northrop, 2001; Pohl et al., 2005]. In *domain engineering*, a developer defines a feature model describing the valid combinations of features. Furthermore, a developer creates reusable software artifacts (i.e., domain artifacts) that implement each feature. For example, the feature modules of the object store are domain artifacts. In *application engineering*, the developer determines a selection of features that serves the needs of the user best and that is valid according to the feature model. Based on this selection and the domain artifacts created during domain engineering, the software product containing the selected features is created. For example, composing the feature modules *SingleStore* and *AccessControl* results in a store tailored for a particular user.

In this thesis, we focus on implementation techniques for software product lines that support the automatic generation of products based on a selection of features. Once a user selects a valid subset of features, a *generator* derives the corresponding product, without further user assistance, such as manual assembly or providing glue code. Besides feature-oriented programming, there are many examples of such implementation techniques. Apel et al. [2013a] distinguish between annotation-based implementation approaches, such as preprocessors [Liebig et al., 2010; Tartler et al., 2011] and generative programming [Czarnecki and Eisenecker, 2000], and composition-based implementation approaches, such as feature-oriented programming [Batory et al., 2004; Prehofer, 1997], delta-oriented programming [Schaefer et al., 2010a], and aspect-oriented programming [Kiczales et al., 1997]. The overall goal is to minimize the effort to tailor software products to the needs of the user.

Ideally, application engineering is reduced to a manual feature selection and automatic generation of verified software products. However, then we need to implement, specify, and verify software product lines during domain engineering. We give an overview on the state-of-the-art of such approaches for product-line specification and verification in the next chapter.

3. Classification and Survey of Product-Line Analyses

This chapter shares material with the ACM Computing Surveys article "A Classification and Survey of Analysis Strategies for Software Product Lines" [Thüm et al., 2014a]. We presented initial ideas at VAST'11 [Thüm et al., 2011b], VaMoS'13 [von Rhein et al., 2013], and ISSTA'13 [Thüm, 2013]. Furthermore, we have given an overview on analysis tools for product lines at SPLat'14 [Meinicke et al., 2014].

Software product lines challenge traditional analysis techniques, such as type checking, model checking, and theorem proving, in their quest of ensuring correctness and reliability of software. Simply creating and analyzing all products of a product line is usually not feasible, due to the potentially exponential number of valid feature combinations. Recently, researchers began to develop analysis techniques that take the distinguishing properties of software product lines into account, for example, by checking feature-related code in isolation or by exploiting variability information during analysis. The emerging field of product-line analyses is both broad and diverse, so it is difficult for researchers and practitioners to understand their similarities and differences.

In this chapter, we propose a classification of product-line analyses to enable systematic research and application. Our classification identifies different strategies for product-line analysis, as well as for implementation and specification of product lines. While we are faithful that our classification applies to a wide variety of software analyses, we focus on particular analyses in our survey for clarity: We concentrate on development techniques, in which products are generated automatically based on a feature selection. In contrast to the typically low number of products when manual assembly is required, automatic generation often leads to a huge number of products and thus is especially challenging for product-line analyses. Furthermore, we survey analysis approaches that operate

statically, such as type checking, model checking, and theorem proving. Analyses that focus exclusively on requirements engineering and domain analysis or that focus only on testing are outside the scope of our survey – we refer the reader to dedicated surveys on feature-model analysis [Benavides et al., 2010; Janota et al., 2008] and on product-line testing [Da Mota Silveira Neto et al., 2011; Engström and Runeson, 2011; Lee et al., 2012; Oster et al., 2011; Tevanlinna et al., 2004].

In Section 3.1, we give a high-level overview on our classification. The following four sections contain definitions, examples, a discussion of advantages and disadvantages for each product-line analysis strategy, and an overview on existing approaches. For a detailed discussion of all surveyed articles and the methodology used to perform the survey, we refer to our original article [Thüm et al., 2014a]. Based on our insights with classifying and comparing a corpus of 137 research articles, we present a research agenda to guide future research on product-line analyses in Section 3.6. We discuss related classifications and surveys in Section 3.7 and conclude our findings in Section 3.8.

3.1   Classification Overview

We exemplify the problem of analyzing product lines by means of the object store example. Then, we give an overview on our classification of analysis strategies and specification strategies.

The Object Store as a Running Example

We refer again to our object store shown in Listing 2.2 on Page 13. An interesting issue introduced deliberately is that one of the four valid products misbehaves. The purpose of feature *AccessControl* is to prohibit access to sealed stores. We could specify this intended behavior formally, for example, using temporal logic:

$$\models \mathbf{G} \ AccessControl \Rightarrow (state\_access(\mathsf{Store} \ s) \Rightarrow \neg \, s.\mathsf{sealed})$$

The formula states, given that feature *AccessControl* is selected, whenever the object store s is accessed, the object store is not sealed. If we select feature *AccessControl* in combination with *MultiStore* as illustrated in Listing 2.3 on Page 13, the specification of feature *AccessControl* is violated; a client can access a store using method readAll even though the store is sealed.

There are several solutions to solve this misbehavior. We could modify the feature model to forbid the critical feature combination, we could change the specification, or we could resolve the problem with alternative implementation patterns. For instance, we can alter the implementation of feature *AccessControl* by refining method readAll in analogy to methods read and set. While this change resolves the misbehavior when combining *MultiStore* and *AccessControl*, it introduces a new problem: The changed implementation of *AccessControl* no longer composes with *SingleStore*, because it attempts to override method readAll, which is not present in this configuration. The

illustrated problem is called the *optional feature problem* [Kästner et al., 2009b; Liu et al., 2006]: The implementation of a certain feature may rely on the implementation of another feature (e.g., caused by method references), and thus the former feature cannot be selected independently, even if it is desired by the user.

The point of our example is to illustrate how products can misbehave or cause type errors even though they are valid according to the feature model. Even worse, such problems may occur only in specific feature combinations, out of potentially millions of combinations that are valid according to the feature model; hence, they are hard to find and may show up only late in the software life cycle. Inconsistencies between the feature model and the implementation have repeatedly been observed in real product lines and are certainly not an exception [Abal et al., 2014; Kästner et al., 2012a; Kolesnikov et al., 2013; Medeiros et al., 2013; Tartler et al., 2011; Thaker et al., 2007]. Ideally, analysis strategies for software product lines are applied in domain engineering rather than application engineering, to detect faults as early as possible.

Classification of Product-Line Analyses

In the last decade, researchers have proposed a number of analysis approaches tailored to software product lines. The key idea is to exploit knowledge about features and the commonality and variability of a product line to systematically reduce analysis effort. Existing product-line analyses are typically based on standard analysis methods, in particular, type checking, static analysis, model checking, and theorem proving. All these methods have been used successfully for analyzing single software products. As discussed in Section 2.1.3, they have complementary strengths and weaknesses with regard to practicality, correctness guarantees, and complexity; so, all of them appear useful for product-line analysis. However, in most cases, it is hard to compare these analysis techniques regarding scalability or even to find the approach that fits a given product-line scenario best. The reason is that the approaches are often presented using varying nomenclatures, especially if multiple software analyses are involved.

In our survey, we classify existing product-line analyses based on how they attempt to reduce analysis effort – the *analysis strategy*. We distinguish three basic strategies, indicating whether the analysis is applied to products, features, or the whole product line: product-based, feature-based, and family-based analyses. We explain the basic strategies and discuss existing approaches implementing each strategy. While surveying the literature, we found approaches that actually combine some of the basic strategies. Hence, we discuss possible combinations, as well. For each strategy, we provide a definition and an example, we discuss advantages and disadvantages, and we classify existing approaches.

Classification of Product-Line Specifications

Many software analyses, such as model checking and theorem proving, require specifications defining the expected behavior of the programs to analyze. These analyses check the conformance of the actual behavior of a given program with the expected behavior.

While surveying the literature, we identified different strategies to define specifications for product-line analyses. We briefly present each specification strategy and will use them to classify approaches for product-line analyses in later sections.

For some analyses, it is sufficient to define a specification independent of the analyzed product line – referred to as *domain-independent specification*. A prominent example for a domain-independent specification is a type system, which is assumed to hold for every software product line written using a particular product-line implementation technique and programming language. Further examples for domain-independent specifications are parsers (i.e., syntax conformance) [Kästner et al., 2011b], the absence of runtime exceptions [Post and Sinz, 2008; Rubanov and Shatokhin, 2011], path coverage [Shi et al., 2012], or that every program statement in a software product line appears in, at least, one product [Tartler et al., 2011]. However, a domain-independent specification can only describe properties that are common across product lines.

If a domain-independent specification is insufficient, we can define a specification for a particular product line that is assumed to hold for all products – called *family-wide specification*. For example, in a product line of pacemakers, all products have to adhere to the same specification, stating that a heart beat is generated whenever the heart stops beating [Liu et al., 2007]. A limitation of family-wide specifications is that we cannot express varying behavior that is common to some but not all products of the product line.

In principle, we could define a specification for every software product individually – referred to as *product-based specification*. We can use any specification technique from single-system engineering without adoption for product-based specification. However, specifying the behavior for every product scales only for software product lines with few products. Furthermore, it involves redundant effort to define behavior that is common for two or more products.

In order to achieve reuse for specifications, we can specify the behavior of features instead of products – called *feature-based specification* [Apel et al., 2013c]. Every feature is specified without any explicit reference to other features. Nevertheless, they may be used to verify properties across features (e.g., for feature-interaction detection) [Apel et al., 2013c]. For example, in our object store, we could define a specification for feature *AccessControl* that objects cannot be accessed, if the store is sealed. This specification would apply to all products that contain feature *AccessControl*.

Finally, it is also possible to define specifications that particular subsets of all products have in common – referred to as *family-based specification*. In a family-based specification, we can specify properties of individual features or feature combinations. Basically, we can provide specifications together with a *presence condition*, which describes a subset of all valid configurations (e.g., by a propositional formula). Alternatively, features can be referenced directly in the specification. For example, in our object store we might want to specify that objects cannot be accessed using method readAll, if the store is sealed *and* the product contains the features *MultiStore* and *AccessControl*. In fact, family-based specification generalizes family-wide, product-based, and feature-based

specifications, in a sense that such specifications can be expressed as special family-based specifications. With a family-based specification, we can automatically generate specifications of individual products, similar to product generation. Several family-based specifications require extensions to existing specification techniques [Asirelli et al., 2012; Classen et al., 2013], as features are referenced explicitly to model variability in properties.

3.2 Product-Based Analyses

Pursuing a product-based analysis, the products of a product line are generated and analyzed individually, each using a standard analysis technique. The simplest approach is to generate and analyze *all* products in a brute-force fashion, but this is feasible only for product lines with few products. A typical strategy is to sample a smaller number of products, usually based on some coverage criteria, such that still reasonable statements on the correctness or other properties of the entire product line are possible [Nie and Leung, 2011; Oster et al., 2010; Perrouin et al., 2010].

Definition 3.1 (Product-based analysis)**.** *An analysis of a software product line is* product-based, *if it operates only on generated products or models thereof, whereas the feature model may be used to generate all products or to implement optimizations. A product-based analysis is called* optimized, *if it operates on a subset of all products (a.k.a.* sample-based analysis*) or if intermediate analysis results of some products are reused for other products; it is called* unoptimized *otherwise (a.k.a.* exhaustive, comprehensive, brute-force, *and* feature-oblivious analysis*).*

Example 3.2. *In our object-store example, we can generate and compile* every *product to detect type errors. However, we could save analysis effort when checking whether the specification of feature* AccessControl *is satisfied: First, all products that do not contain* AccessControl *do not need to be checked. Second, if two products differ only in features that do not concern class* Store *(not shown in our example; e.g., features that are concerned with other data structures), only one of these products needs to be checked.*

Advantages and Disadvantages

The main advantage of product-based analyses is that every existing software analysis can easily be applied in the context of software product lines. In particular, existing off-the-shelf tools can be reused to analyze individual products. Furthermore, product-based analyses can easily deal with changes to software product lines that alter only a small set of products, because only changed products need to be re-analyzed.

A specific advantage of an unoptimized product-based analysis is the soundness and completeness with respect to the analysis that is scaled from single-system engineering (i.e., the *base analysis*). First, every fault detected using this strategy, is a fault of a software product that can be detected by the base analysis (soundness). Second, every

fault that can be detected using the base analysis, is also detected using an unoptimized product-based analysis (completeness). Note that, while the base analysis itself might be unsound or incomplete with regard to some specification and analysis goal, this strategy is still sound and complete with regard to the base analysis (i.e., it will detect the same faults).

However, there are serious disadvantages of product-based analyses. Already generating all products of a software product line is often infeasible, because the number of products is up-to exponential in the number of features. Even if the generation of all products is possible, separate analyses of individual products perform inefficient, redundant computations, due to similarities between the products.

The analysis results of product-based analyses refer necessarily to generated artifacts of products, and not to domain artifacts implemented in domain engineering, which gives rise to two difficulties. First, a programmer may need to read and understand the generated code in order to understand the analysis results (e.g., the composed class Store in Listing 2.3 contains all members introduced by the features of the analyzed product). Second, if a change to the code is necessary, it must be applied to the domain artifacts instead of generated artifacts, and automatic mappings are not always possible [Kuhlemann and Sturm, 2010].

While an unoptimized product-based strategy is often not feasible in practice, it serves as a baseline for other strategies in terms of soundness, completeness, and efficiency. Ideally, an analysis strategy is sound and complete with respect to the base analysis, and, at the same time, it is more efficient than the unoptimized product-based strategy. However, we will also discuss strategies that are incomplete or unsound to increase the efficiency of the overall analysis.

Unoptimized Product-Based Analyses

Product-based strategies are widely used in practice, because they are simple and can be applied without creating and using new concepts and tools. For example, when generating and compiling individual software products, type checking is usually done internally by the compiler (e.g., the Java compiler). Type checking is redundant when different products share implementation artifacts, and sharing artifacts between products is the common case and goal in product-line engineering [Apel et al., 2013a; Czarnecki and Eisenecker, 2000].

In general, we found no proposal in the literature explicitly suggesting an unoptimized product-based analysis. However, we found some approaches that actually use product-based analyses for specific implementation mechanisms and do not discuss how to deal with many products; these approaches apply type checking [Apel et al., 2008a; Buchmann and Schwägerl, 2012; Istoan, 2013], static analyses [Klaeren et al., 2001; Scholz et al., 2011], model checking [Apel et al., 2010c; Bessling and Huhn, 2014; Fantechi and Gnesi, 2008; Istoan, 2013; Kishi and Noda, 2006; Ubayashi and Tamai, 2002], and

theorem proving [Harhurin and Hartmann, 2008] to software product lines. The unoptimized product-based analysis strategy has been used with domain-independent specifications [Apel et al., 2008a; Buchmann and Schwägerl, 2012; Istoan, 2013], family-wide specifications [Fantechi and Gnesi, 2008; Istoan, 2013; Kishi and Noda, 2006; Ubayashi and Tamai, 2002], and feature-based specifications [Apel et al., 2010c; Bessling and Huhn, 2014; Harhurin and Hartmann, 2008; Istoan, 2013; Klaeren et al., 2001; Scholz et al., 2011]. These approaches considered composition-based implementation [Apel et al., 2008a; Klaeren et al., 2001; Scholz et al., 2011; Ubayashi and Tamai, 2002], composition-based design [Apel et al., 2010c; Bessling and Huhn, 2014; Harhurin and Hartmann, 2008; Istoan, 2013], and annotation-based design [Buchmann and Schwägerl, 2012; Fantechi and Gnesi, 2008; Kishi and Noda, 2006] as domain artifacts.

Optimized Product-Based Analyses

Several optimized product-based strategies have been proposed to improve scalability and reduce redundant computations. Optimizations proposed in the literature focus either on detecting redundant parts in analyses [Bruns et al., 2011; Cordy et al., 2012d; Hähnle et al., 2013] or on eliminating products that are already covered by other analysis steps [Apel et al., 2013d; Jayaraman et al., 2007; Katz, 2006; Liebig et al., 2013; Plath and Ryan, 2001], according to certain coverage criteria. The optimized product-based strategy has been applied to type checking [Jayaraman et al., 2007; Liebig et al., 2013], static analysis [Liebig et al., 2013], model checking [Apel et al., 2013d; Cordy et al., 2012d; Katz, 2006; Lochau et al., 2014; Plath and Ryan, 2001], and theorem proving [Bruns et al., 2011; Hähnle et al., 2013]. This strategy has been discussed for composition-based implementation [Apel et al., 2013d; Bruns et al., 2011; Hähnle et al., 2013; Katz, 2006], composition-based design [Jayaraman et al., 2007; Lochau et al., 2014; Plath and Ryan, 2001], and annotation-based implementations [Liebig et al., 2013]. We classify their used specification strategies as domain-independent specification [Jayaraman et al., 2007; Liebig et al., 2013; Plath and Ryan, 2001], family-wide specification [Liebig et al., 2013; Lochau et al., 2014], and feature-based specification [Apel et al., 2013d; Bruns et al., 2011; Hähnle et al., 2013; Katz, 2006].

3.3 Family-Based Analyses

The main problem with product-based analyses are redundant computations, because the products of a software product line share code [Apel et al., 2013a; Czarnecki and Eisenecker, 2000]. Besides an optimized product-based strategy, another option to achieve a more efficient analysis is to consider domain artifacts such as feature modules instead of generated artifacts (i.e., products).

Family-based analyses operate on domain artifacts and valid combinations thereof, as specified by a feature model. The feature model is usually converted into a logic formula to allow analysis tools to reason about all valid combinations of features (e.g., a satisfiability solver can be used to check whether a method is defined in all valid feature combinations, in which it is referenced). The overall idea is to analyze domain artifacts

and feature model in concert from which we can conclude that some intended properties hold for all products. Often, all implementation artifacts of all features are merged into a single virtual product (a.k.a. *metaproduct* or *product simulator*). The virtual product is not necessarily a valid product due to optional and mutually exclusive features [Thüm et al., 2012].

Definition 3.3 (Family-based analysis). *An analysis of a software product line is family-based, if it (a) operates only on domain artifacts and (b) incorporates the knowledge about valid feature combinations.*

Example 3.4. *A family-based type checker, for instance, can analyze the code base of the object store example (i.e., all feature modules) in a single pass, although the features are combined differently in the individual products. To this end, it takes variability into account, in the sense that individual feature modules may be present or absent in certain products. Regarding method invocations, it checks whether a corresponding target method is declared in* every *valid product in which it is invoked. In Listing 3.1, we illustrate how a family-based type system checks whether the references of a slightly modified feature module* AccessControl *to the methods* read *and* readAll *are well-typed in* every *valid product. For method* read*, the type system infers that the method is introduced by the feature modules* SingleStore *and* MultiStore*, and that one of them is always present (checked using a satisfiability solver; solid arrows).*[1] *FM denotes the feature model of Figure 2.1 on Page 11 as propositional formula. For method* readAll*, it infers that the method is introduced only by feature module* MultiStore*, which may be absent when feature module* AccessControl *is present (dashed arrow). Hence, the type system reports a fault and produces a counter example in terms of a valid feature selection that contains a dangling method invocation:* {SingleStore, AccessControl}.

Advantages and Disadvantages

Family-based strategies have advantages and disadvantages compared to product-based strategies; we begin with the advantages. First of all, not every individual product must be generated and analyzed, because family-based analyses operate on domain artifacts. Thus, family-based strategies avoid redundant computations across multiple products, in which reasoning about variability and commonality prevents these duplicate analyses.

Second, the analysis effort is *not* proportional to the number of valid feature combinations. While the satisfiability problem is in NP-complete, in practice, satisfiability solvers perform well when reasoning about feature models [Mendonça et al., 2009; Thüm et al., 2009]. Intuitively, the performance of family-based analyses is mainly influenced by the number and size of feature implementations and the amount of sharing during analysis [Brabrand et al., 2013], but largely independent of the number of valid feature combinations. For comparison, the effort for product-based approaches increases with every new product.

---

[1]A satisfiability solver can be used to check whether a propositional formula is a tautology by checking whether the negation of the whole formula is unsatisfiable.

```
class Store {                                          feature module SingleStore
  private Object value;
  Object read() { return value; }
  void set(Object nvalue) { value = nvalue; }
}                                    FM ⇒ (AccessControl ⇒ SingleStore ∨ MultiStore)
```
```
class Store {                                           feature module MultiStore
  private LinkedList values = new LinkedList();
  Object read() { return values.getFirst(); }
  Object[] readAll() { return values.toArray(); }
  void set(Object nvalue) { values.addFirst(nvalue); }
}
```
```
class Store {                                          feature module AccessControl
  private boolean sealed = false;
  Object read() {
    if (!sealed) { return original(); }
    else { throw new RuntimeException("Access denied!"); }
  }
  Object[] readAll() {
    if (!sealed) { return original(); }   FM ⇒ (AccessControl ⇒ MultiStore)
    else { throw new RuntimeException("Access denied!"); }
  }
  void set(Object nvalue) {
    if (!sealed) { original(nvalue); }
    else { throw new RuntimeException("Access denied!"); }
  }
}
```

Listing 3.1: Family-based type checking for methods read and readAll.

However, family-based strategies also have disadvantages. Often, known analysis methods for single products cannot be used as they are. The reason is that the analysis method must be aware of features and variability. Existing analysis methods and off-the-shelf tools need to be extended, if possible, or new analysis methods need to be developed. For some software analyses, such as model checking and theorem proving, there exist techniques to encode the analysis problem in an existing formalism or language (e.g., using a metaproduct simulating all products) and reuse off-the-shelf tools [Apel et al., 2011; Post and Sinz, 2008; Thüm et al., 2014; Thüm et al., 2012], but it is not clear whether these techniques can be used for all kinds of software analyses.

Second, changing the domain artifacts of one feature or a small set of features, usually requires to analyze the whole product line again from scratch [Cordy et al., 2012b]. Hence, the effort for very large product lines with many features is much higher than actually necessary, while the product line evolves over time. However, it is possible to cache certain parts of the analysis, which may reduce the overall analysis effort [Kästner et al., 2012a].

Third, changing the feature model usually requires to analyze the whole product line again. For instance, if we add a new product or a small set of new products, we may be faster analyzing these new products with a product-based strategy than analyzing the product line again using a family-based strategy. Similar to domain-artifact changes, this depends on the analysis approach and available caching strategies. When the feature model was specialized or refactored (i.e., no new products are added), reanalyzing the product line cannot reveal new faults [Thüm et al., 2011a].

Fourth, as family-based analyses consider all domain artifacts as a whole, the size of the analysis problem can easily exceed physical boundaries such as the available memory [Apel et al., 2013d]. Thus, family-based analysis may be infeasible for large software product lines and expensive analyses.

Finally, family-based analyses assume a closed world – all features have to be known during the analysis process (e.g., to look up all potential targets of method invocations). In practice, this may be infeasible, for example, in multi-team development or software ecosystems such as Eclipse. Note, whenever we want to analyze the *whole* software product line, a closed world is required – independent of the chosen strategy.

Family-Based Syntax Checking

Although parsing detects only certain defects in source code (i.e., syntax conformance with respect to a domain-independent specification), it is a necessary step for many analyses such as type checking. While parsing is straightforward for composition-based implementation approaches such as feature-oriented programming or aspect-oriented programming, it is complicated for product lines implemented with conditional compilation. There are several approaches for family-based parsing of C code with preprocessor directives that avoid to preprocess the code for each product separately by generating a variability-aware abstract syntax tree [Gazzillo and Grimm, 2012; Kästner et al., 2011b; Medeiros et al., 2013; Nguyen et al., 2014a].

Family-Based Type Checking

Family-based strategies have been proposed by several authors for type checking of software product lines. The majority of work on family-based type checking is about creating variability-aware type systems (i.e., a domain-independent specification) and proving that, whenever a product line is type safe according to the type system, all derivable products are also well-typed. The rules of these type systems contain reachability checks (basically implications) making sure, among others, that every program element is defined in all products where it is referenced. Variability-aware type systems have been developed for composition-based implementation [Apel et al., 2010a,d; Delaware et al., 2009; Huang et al., 2007; Kim et al., 2008; Kolesnikov et al., 2013; Kuhlemann et al., 2009; Schröter et al., 2014; Thaker et al., 2007], composition-based design [Alférez et al., 2011], annotation-based implementation [Aversano et al., 2002; Chen and Erwig, 2014; Chen et al., 2014; Kästner et al., 2012a,b; Kenner et al., 2010; Kim et al., 2008; Le et al., 2013; Liebig et al., 2013; Teixeira et al., 2011], and annotation-based design [Czarnecki and Pietroszek, 2006; Heidenreich, 2009; Metzger et al., 2007].

Family-Based Static Analysis

Recently, researchers have proposed family-based static analyses for software product lines, in particular, intra-procedural [Brabrand et al., 2013; Kanning and Schulze, 2014; Liebig et al., 2013; Midtgaard et al., 2014] and inter-procedural [Bodden et al., 2013] data-flow analyses. Furthermore, static analyses have been proposed [Adelsberger et al., 2014; Nguyen et al., 2014a; Ribeiro et al., 2010; Sabouri and Khosravi, 2014; Tartler et al., 2011] that do not scale an existing static analysis known from single-system engineering, but rather focus on an analysis that is specific to product lines – to which we refer to as *family-specific analyses*. Interestingly, most approaches for family-based static analysis are designed for annotation-based implementations and domain-independent specifications. As an exception, Adelsberger et al. [2014] focus on composition-based implementations with feature-oriented programming, and Sabouri and Khosravi [2014] propose a family-wide specification.

Family-Based Model Checking

One distinguishing characteristic of approaches for family-based model checking is whether they operate directly on source code or on an abstraction of a system. The former is known as software model checking and we refer to the latter as abstract model checking. The majority of approaches for family-based model checking apply abstract model checking [Asirelli et al., 2012; Classen et al., 2014, 2013, 2010; Cordy et al., 2013a, 2012a,b,c, 2013b; Dubslaff et al., 2014; Fischbein et al., 2006; Greenyer et al., 2013; Gruler et al., 2008; Lauenroth et al., 2010; Sabouri and Khosravi, 2012, 2013a,b, 2014; Shi et al., 2014; ter Beek and de Vink, 2014; ter Beek et al., 2013]. In contrast, several authors proposed approaches for family-based software model checking [Apel et al., 2011, 2013d,d; Kästner et al., 2012c; Post and Sinz, 2008; Schaefer et al., 2010b]. Family-based model checking has been applied to composition-based [Apel et al., 2011, 2013d; Classen et al., 2014, 2013; Dubslaff et al., 2014; Greenyer et al., 2013; Sabouri and Khosravi, 2013a] and annotation-based [Asirelli et al., 2012; Classen et al., 2013, 2010; Cordy et al., 2013a, 2012a,b,c, 2013b; Fischbein et al., 2006; Gruler et al., 2008; Lauenroth et al., 2010; Post and Sinz, 2008; Sabouri and Khosravi, 2012, 2013b, 2014; Schaefer et al., 2010b; Shi et al., 2014; ter Beek and de Vink, 2014; ter Beek et al., 2013] product lines.

Besides the product line's source code or an abstraction thereof, family-based model checking requires a formalism to encode properties (i.e., specifications) to be checked. Existing formalisms and languages to specify properties, such as computation tree logic (CTL) and linear temporal logic (LTL), have been lifted to product lines as domain-independent [Post and Sinz, 2008; Sabouri and Khosravi, 2013a], family-wide [Cordy et al., 2012c; Fischbein et al., 2006; Greenyer et al., 2013; Gruler et al., 2008; Sabouri and Khosravi, 2012, 2013b, 2014; Schaefer et al., 2010b; Shi et al., 2014; ter Beek and de Vink, 2014; ter Beek et al., 2013], feature-based [Apel et al., 2011, 2013d; Classen et al., 2010; Dubslaff et al., 2014; Lauenroth et al., 2010], and family-based specifications [Asirelli et al., 2012; Classen et al., 2014, 2013; Cordy et al., 2013a, 2012a,b, 2013b].

Family-Based Theorem Proving

In our survey, we identified that there is no approach applying the family-based strategy to theorem proving. Based on this insight, we propose family-based theorem proving for product lines implemented with feature-oriented programming in Section 5.2.

3.4 Feature-Based Analyses

Software product lines may also be analyzed using a *feature-based* strategy. That is, all domain artifacts implementing a certain feature are analyzed in isolation (in bundles assigned to individual features) without considering other features or the feature model. The idea of feature-based analyses is to reduce the potentially exponential number of analysis tasks (i.e., for every valid feature combination) to a linear number of analysis tasks (i.e., for every feature) by accepting that the analysis *might* be incomplete. The assumption of feature-based analysis is that certain properties of a feature can be analyzed modularly, without reasoning about other features and their relationships. Similar to family-based strategies, feature-based strategies operate on domain artifacts instead of generated products. Contrary to family-based strategies, no feature model is needed as every feature is analyzed only in isolation. Feature-based analyses are sound and complete with respect to the base analysis, if the properties and the analyses are *compositional* with respect to the features (i.e., the analysis results cannot be invalidated by interactions of features).

Definition 3.5 (Feature-based analysis)**.** *An analysis of a software product line is feature-based, if it (a) operates only on domain artifacts and (b) software artifacts belonging to a feature are analyzed in isolation (i.e., knowledge about valid feature combinations is not used), and feature interactions are not considered.*

Example 3.6. *In the object-store example, we can analyze each of the three feature modules in isolation to some extent. First, we can parse each feature module in isolation to make sure that it conforms to the syntax and to create an abstract syntax tree for each feature module. For syntax checking, it is sufficient to consider each feature module in isolation, as syntactic correctness is independent of other features, and thus a compositional property. Second, the type checker uses the abstract syntax tree to infer which types and references can be resolved by a feature itself and which have to be provided by other features. As an example, all references to field* sealed *are internal and can be checked within the implementation of feature* AccessControl*, as illustrated in Listing 3.2. That is, there is no need to check this reference for every product. However, some of the references cut across feature boundaries and cannot be checked in a feature-based fashion. Well-typedness is not a compositional property. For example, references to the methods* read *and* readAll *of feature* AccessControl *cannot be resolved within the feature.*

Advantages and Disadvantages

Feature-based analyses have a strong disadvantage that we want to discuss first. A feature-based analysis can only detect issues *within* a certain feature and cannot reason

```
class Store {                                          feature module AccessControl
  private boolean sealed = false;
  Object read() {
    if (!sealed) { return original(); }
    else { throw new RuntimeException("Access denied!"); }
  }
  Object[] readAll() {
    if (!sealed) { return original(); }
    else { throw new RuntimeException("Access denied!"); }
  }
  void set(Object nvalue) {
    if (!sealed) { original(nvalue); }
    else { throw new RuntimeException("Access denied!"); }
  }
}
```

Listing 3.2: Feature-based type checking for field sealed.

about issues *across* features, because features are only analyzed in isolation. A well-known problem in this context are *feature interactions* [Calder et al., 2003]: several features work as expected in isolation, but lead to unexpected behavior in combination. A prominent example from telecommunication systems is that of the features *CallForwarding* and *CallWaiting* [Bowen et al., 1989]. While both features may work well in isolation, it is not clear what should happen if both features are selected and an incoming call arrives at a busy line: forwarding the incoming call or waiting for the other call to be finished. Hence, feature-based strategies must usually be combined with product-based or family-based strategies to cover feature interactions and to deal with non-compositional properties.

Nevertheless, feature-based strategies have advantages compared to product-based and family-based strategies. First, they analyze domain artifacts (similar to family-based strategies) instead of operating on generated software artifacts, and thus there are no redundant computations across products.

Second, the feature-based strategy supports open-world scenarios: It is not required that all features are known at analysis time. Furthermore, it is not required to have a feature model, which is typically not available in an open-world scenario. Nevertheless, a feature-based strategy can also be applied for closed-world scenarios, where all features and their valid combinations are known at analysis time.

Third, the effort to analyze a product line is minimal, when one or a small set of features are changed. In such cases, only changed features need to be re-analyzed in isolation, whereas with family-based and product-based strategies, we would need to re-analyze the whole product line or all affected products.

Fourth, the analysis of a software product line using a feature-based strategy is divided into smaller analysis tasks. Thus, a feature-based strategy is especially useful

for software analysis with extensive resource consumption (e.g., memory) and for large software product lines, for which some family-based analyses are not feasible.

Finally, changing only the feature model does not affect feature-based analysis at all. Hence, when the feature model evolves, we do not need to perform any feature-based analysis again, since features are only analyzed in isolation.

Feature-Based Approaches

As indicated previously, there are only few strict feature-based approaches. For example, parsing and syntax checking of software product lines with composition-based implementations, such as feature-oriented or aspect-oriented programs, are compositional analyses. While parsing is a necessary task for any static analysis, it is only discussed for non-modular feature implementations, such as conditional compilation [Kästner et al., 2011b], for which feature-based parsing is impossible. A further example for a simple feature-based analysis is to compute code metrics. For most software analyses, we need to combine feature-based analyses with other strategies.

It may seem odd that we defined a strategy which is not present in literature itself. Indeed, previous drafts of our classification were less restrictive for the feature-based strategy. In particular, we also included approaches that do parts of the analysis feature-based. However, it turned out that many approaches with very different characteristics were classified as feature-based, and it was difficult to assess their conceptual differences. Whereas many approaches claim to be modular or compositional, it is unclear what happens to those parts of the analysis which concern feature interactions (i.e., non-compositional properties). With our more strict classification, we identify how those approaches resolve feature interactions – which we discuss in the next section.

3.5   Combined Analysis Strategies

We have discussed product-based, family-based, and feature-based analyses as different strategies to scale software analyses from single-system engineering to software product lines. These three strategies form the basis of our classification, but they can also be combined resulting in four additional strategies. In this section, we discuss possible combinations even if some of them are not yet implemented.

Feature-Product-Based Analyses

A commonly proposed combined strategy, which we identified in the literature, is the feature-product-based strategy that consists of two phases. First, features are analyzed in isolation and, second, all properties not checked feature-based are analyzed for each product. The feature-based part can only analyze features locally and the product-based part checks that features work properly in combination. The key idea is to reduce analysis effort by checking as much as possible feature-locally.

Definition 3.7 (Feature-product-based analysis). *An analysis of a software product line is* feature-product-based, *if (a) it consists of a feature-based analysis followed by a product-based analysis, and (b) the analysis results of the feature-based analysis are used in the product-based analysis.*

Example 3.8. *In our object store, we could start to type-check all features in isolation. As shown in Listing 3.2 on Page 29, we can check that all intra-feature references are valid and create an interface for every feature. The interface contains all methods, fields, and classes that the feature provides and also those that are required. In the second step, we take these interfaces and iterate over every valid combination of features and check whether the interfaces are compatible for every valid configuration (i.e., everything that is required in some interface is provided by another interface). This way, we can save redundant checks for intra-feature references. Especially, if some features evolve, we can omit re-analyzing unchanged features in the feature-based analysis step.*

Feature-product-based strategies reduce redundant computations, compared to strict product-based strategies, but redundancies still occur for all analyses applied at the product level. For example, when some features evolve, other features need not to be re-analyzed, but all products containing any of the affected features need to be analyzed again whenever the feature interface changes. Considering that strict feature-based strategies are not sufficient for non-compositional properties, feature-product-based strategies seem to be a good compromise. Whether feature-product-based strategies are better than family-based strategies depends on the actual analysis, the number of products, how much can be checked feature-based, and whether evolution of the product line is an issue.

The feature-product-based strategy has been applied to scale type checking [Apel and Hutchins, 2010; Bettini et al., 2013, 2015; Klose and Ostermann, 2010; Kolesnikov et al., 2013], model checking [Blundell et al., 2004; Fisler and Krishnamurthi, 2001; Li et al., 2002, 2005; Liu et al., 2011; Nelson et al., 2001], and theorem proving [Batory and Börger, 2008; Damiani et al., 2012; Delaware et al., 2011, 2013] to software product lines. These approaches check composition-based implementations [Apel and Hutchins, 2010; Bettini et al., 2013, 2015; Damiani et al., 2012; Klose and Ostermann, 2010; Kolesnikov et al., 2013] and composition-based designs [Blundell et al., 2004; Fisler and Krishnamurthi, 2001; Li et al., 2002, 2005; Liu et al., 2011; Nelson et al., 2001] against domain-independent specifications [Apel and Hutchins, 2010; Bettini et al., 2013, 2015; Klose and Ostermann, 2010; Kolesnikov et al., 2013], family-wide specifications [Blundell et al., 2004; Fisler and Krishnamurthi, 2001; Liu et al., 2011; Nelson et al., 2001], and feature-based specifications [Damiani et al., 2012; Li et al., 2002, 2005].

Feature-Family-Based Analyses

A strategy similar to feature-product-based analysis is to combine feature-based and family-based analyses. The idea of feature-family-based analysis is to analyze features separately, and to analyze everything that could not be analyzed in isolation based on properties inferred from the feature-based analysis.

```
provides class Store {                                              interface of SingleStore
  provides Object Store.read();
  provides void Store.set(Object);   FM ⇒ (AccessControl ⇒ SingleStore ∨ MultiStore)
}
```

```
provides class Store {                                               interface of MultiStore
  provides Object Store.read();
  provides Object[] Store.readAll();
  provides void Store.set(Object);
}
```

```
requires class Store {                                             interface of AccessControl
  requires Object Store.read();
  requires Object[] Store.readAll();   FM ⇒ (AccessControl ⇒ MultiStore)
  requires void Store.set(Object);
}
```

Listing 3.3: Feature-family-based type checking with interfaces.

Definition 3.9 (Feature-family-based analysis). *An analysis of a software product line is* feature-family-based, *if (a) it consists of a feature-based analysis followed by a family-based analysis and (b) the analysis results of the feature-based analysis are used in the family-based analysis.*

Example 3.10. *In our object store, we can infer interfaces for each feature using feature-based type checking and check these interfaces for compatibility using family-based type checking. The interface of each feature defines the program elements it provides and the program elements it requires (cf. Listing 3.3). For example, feature Ac-cessControl requires a method* read, *which is provided either by feature SingleStore or feature MultiStore. However, method* readAll *required by feature AccessControl is not available in all products with feature AccessControl. Basically, we can create a propositional formula for each reference, which can be checked using a satisfiability solver, as described in Section 3.3.*

Feature-family-based analysis can be seen as an improvement of feature-product-based analysis, as redundant computations are eliminated entirely (i.e., redundancies are not only eliminated for feature-local analyses, but also for analyses across features). Furthermore, compared to a pure family-based analysis, it better supports the evolution of software product lines, in which usually only a small set of features evolves. Finally, a feature-family-based analysis combines open-world and closed-world scenarios. That is, while the feature-based analysis does not require to know all features and their valid combinations, we can post-pone all parts of the analysis requiring a closed world to the family-based analysis.

The feature-family-based strategy has been proposed for type checking [Damiani and Schaefer, 2012; Delaware et al., 2009] and theorem proving [Hähnle and Schaefer, 2012] of composition-based implementations [Damiani and Schaefer, 2012; Delaware et al.,

2009; Hähnle and Schaefer, 2012] with domain-independent [Damiani and Schaefer, 2012; Delaware et al., 2009] and feature-based specifications [Hähnle and Schaefer, 2012].

Family-Product-Based Analyses

A combination of family-based and product-based analyses may not seem useful at the first thought, because everything that can be analyzed product-based could already be analyzed family-based. Nevertheless, family-product-based analyses can be useful (a) if a product-based analysis is faster for particular parts of the analysis, (b) if there is a part of the analysis (e.g., certain safety properties) that is relevant for one product or a small set of products only, (c) if several software analyses are combined, and (d) if the analysis problem for a family-based analysis is too large to be solved with given resource limitations.

Definition 3.11 (Family-product-based analysis). *An analysis of a software product line is* family-product-based, *if (a) it consists of a (partial) family-based analysis followed by a product-based analysis and (b) the analysis results of the family-based analysis are reused in the product-based analysis.*
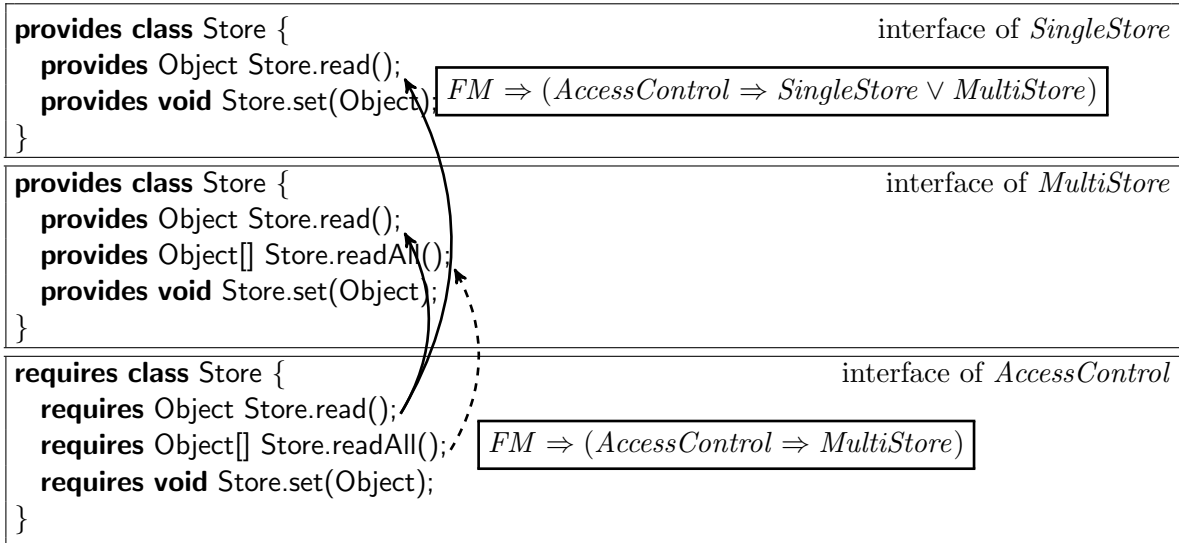
We have not found pure static approaches for this strategy. However, some approaches that combine family-based static analysis with product-based dynamic analyses for software product lines [Kim et al., 2011, 2010; Shi et al., 2012; Tartler et al., 2012]. In contrast to approaches for sampling discussed in Section 3.2, a family-product-based analysis incorporates not only the feature model, but also the source code of the product line during the family-based analysis step.

Feature-Family-Product-Based Analyses

It is also possible to combine all three analysis strategies. We can first analyze the features in isolation, then check whether the features are compatible in all valid combinations, and finally analyze products that have specific requirements.

Definition 3.12 (Feature-family-product-based analysis). *An analysis of a software product line is* feature-family-product-based, *if (a) it consists of a feature-based analysis followed by a family-product-based analysis, and (b) the analysis results of the feature-based analysis are used during family-product-based analysis.*

We have not found any feature-family-product-based strategy in the literature, but it might be useful to separate product-based from feature-based and family-based analyses, especially, if different software-analysis techniques are combined. It is future work, to analyze and discuss the feasibility of this strategy in more detail.

3.6   Research Agenda

Our aim is to bring the issue of systematic research on and application of product-line analysis to the attention of a broad community of researchers and practitioners. Our classification is intended to serve as an agenda for research on product-line analysis:

- What are the strengths and weaknesses of the individual strategies in practice?

- Is it meaningful to combine each strategy with each software analysis, and which combinations are useful and superior in what circumstances?

- What can we learn from strategies applied to one analysis when applying them to other analyses?

- Are there further novel analysis strategies?

- What characteristics of a given product line affect the efficiency of the individual analysis strategies?

- Is there a principle and possibly automated way to lift a given specification and analysis technique to product lines?

Based on the classification of existing approaches in the previous sections, we discuss underrepresented research areas and specific research questions that we uncovered in our survey: First, we summarize advantages and drawbacks of each strategy, and identify underrepresented analysis strategies. Second, we discuss how strategies have been evaluated quantitatively and report weaknesses of existing evaluations. Third, we discuss which analysis strategies have been combined with which specification and implementation strategies. Finally, we describe future challenges for type checking, static analysis, model checking, and theorem proving of software product lines.

Comparison of Analysis Strategies

In the previous sections, we have discussed three basic strategies and four combined strategies to scale software analysis to product lines. In Figure 3.1, we give an overview of how often each strategy was applied in the surveyed approaches and when. More than half of the approaches apply a family-based strategy, suggesting that this strategy to cope with software variability is well-known. However, we also found approaches for analysis in a product-line context that do not discuss how to cope with many, similar products. Almost a third of all approaches relies on the generation of *all* products (i.e., unoptimized product-based and feature-product-based strategy), which is infeasible for large product lines. None of the surveyed analysis approaches is solely feature-based, because analyzing features only in isolation is usually not sufficient (i.e., the properties of interest are not compositional). All combined strategies except for the feature-product-based strategy are underrepresented.
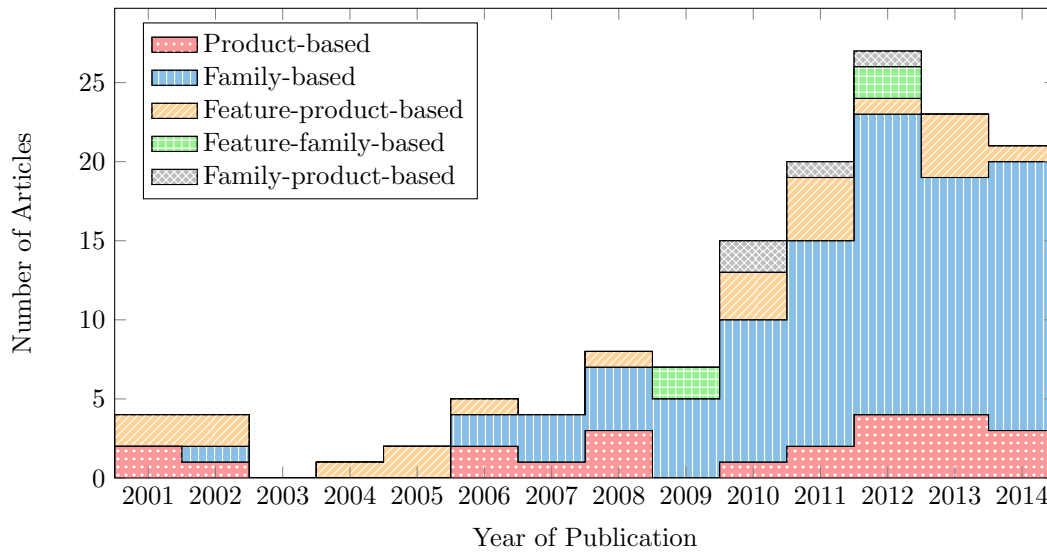
Figure 3.1: Frequency of analysis strategies addressed in the research literature.

In Table 3.1, we summarize the main advantages and disadvantages of all strategies. Each strategy enables the analysis of compositional properties. However, the feature-based strategy is the only strategy that does not support non-compositional properties. A further interesting characteristic is whether analysis results refer to domain artifacts or generated artifacts, because, for the latter, the developer needs to understand generated artifacts. For example, each strategy incorporating a product-based part inherently refers to generated artifacts. As feature-based and family-based strategies operate on domain artifacts, their results also refer to them. Nevertheless, with additional effort it is possible to aggregate analysis results from products.

A key characteristic of each strategy is to which extent redundant computations are avoided [Kolesnikov et al., 2013]. In the product-based strategy, we have redundant computations due to the similarities between products. In contrast, when analyzing a product line with the feature-based strategy, we avoid redundancies by considering domain artifacts in isolation, but we can only analyze compositional properties. The family-based strategy avoids redundancies for both, compositional and non-compositional properties. However, if some domain artifacts evolve in a product line that has been analyzed before, the family-based strategy usually requires redundant analyses. The redundant effort can be reduced by combining it with the feature-based strategy, because we can omit the analysis of domain artifacts for unchanged features.

The size of the analysis problem for a given product line is influenced by the analysis strategy. A particular strategy may conflict with resource limitations, while another does not. For example, even if we can model check each product in isolation on a given machine, it is possible that family-based model checking requires more main memory than actually available and is thus infeasible. In Table 3.1, we compare the expected problem size for each strategy with that of the product-based strategy. In general, we expect smaller problems for strategies incorporating a feature-based analysis step,

Analysis Strategy	Analysis of non-compositional properties	Analysis results refer to domain artifacts	Non-redundant computations for compositional properties	Non-redundant computations for non-compositional properties	Non-redundant computations for evolving product lines	Smaller analysis problems for compositional properties	Smaller analysis problems for non-compositional properties
Product-based	yes	no	no	no	no	—	—
Family-based	yes	yes	yes	yes	no	no	no
Feature-based	no	yes	yes	—	yes	yes	—
Feature-product-based	yes	yes/no[1]	yes	no	maybe[2]	yes	yes
Feature-family-based	yes	yes	yes	yes	maybe[2]	yes	maybe[3]
Family-product-based	yes	yes/no[1]	yes	yes	no	no	no
Feature-family-product-based	yes	yes/no[1]	yes	yes	maybe[2]	yes	maybe[3]

[1] Analysis results of product-based analysis step refers to products
[2] Avoids redundant computations when changed domain artifacts can be verified feature-based
[3] The family-based analysis problem may be larger or smaller than verifying a single product depending on how much feature-based part reduces the analysis problem

Table 3.1: Summary of advantages and disadvantages of analysis strategies.

because the analysis problem is split into an analysis of compositional properties for each feature and an analysis of non-compositional properties. However, as Table 3.1 indicates, avoiding redundant computations (e.g., with the family-based strategy) and minimizing the analysis problem (e.g., with the feature-product-based strategy) are conflicting goals. It seems that the feature-family-based strategy is a good trade-off, but empirical evaluations are needed to find the best strategy based on product-line characteristics.

Quantitative Evaluation of Analysis Strategies

Ideally, we would like to recommend the best strategy for a given software analysis based on static characteristics of a product line, such as the number of features, the number of products, or the size and cohesion of feature implementations. However, for such recommendations, we need reliable empirical evaluations assessing quantitative characteristics for each strategy and analysis. Whereas there are some evaluations, they are often not comparable to each other.

First, we found that, in almost all studies, a particular strategy is compared to an unoptimized product-based analysis [Apel et al., 2013d; Chen et al., 2014; Classen et al., 2014, 2013, 2010; Cordy et al., 2012a,c; Greenyer et al., 2013; Sabouri and Khosravi, 2012, 2013a,b; Schaefer et al., 2010b; Thüm et al., 2012] or to the analysis of a single product [Delaware et al., 2009; Gazzillo and Grimm, 2012; Kästner et al., 2012a,

2011b; Kenner et al., 2010; Post and Sinz, 2008]. The advantage of such a standard evaluation is that we can compare different approaches more easily; even if evaluations strongly depend on the size and kind of product line being analyzed. However, the unoptimized product-based strategy is often not an option in practice (e.g., for large product lines). Recently, researchers started comparing family-based with optimized product-based [Apel et al., 2013d; Liebig et al., 2013] and feature-product-based strategies [Kolesnikov et al., 2013]. However, there are still strategies that have not been compared with any other strategy. For example, researchers proposed feature-family-based analyses [Damiani and Schaefer, 2012; Delaware et al., 2009; Hähnle and Schaefer, 2012], but there is no empirical comparison with a family-based or feature-product-based strategy that assesses the potential of such a strategy.

Second, most studies only focus on time efficiency. However, memory consumption is especially important for product lines, because analyzing all products simultaneously may require significantly more resources than analyzing each product separately. Furthermore, there are different characteristics of product lines (e.g., number of faults) that influence time and memory efficiency of the analysis (e.g., model checking could be faster when the product line contains more faults). Hence, when comparing strategies, we should also incorporate product lines containing no faults, some faults, and many faults in source code and specification. We address this issue in Section 5.2.

Finally, there is no consensus on how to compare strategies empirically. The overall time for product-line analysis may include several analysis steps, but it is questionable what to compare if one strategy includes steps that the other does not include. For example, product-based type checking requires to retrieve all or a subset of all valid configurations from the feature model, to generate products, and the actual type checking of each product. In contrast, family-based type checking does not require to retrieve all valid configurations nor to generate products. Brabrand et al. [2013] and Kolesnikov et al. [2013] document the performance of each analysis step, while all other empirical comparisons ignore variability-model analysis and product generation [Apel et al., 2011, 2013d; Chen et al., 2014; Classen et al., 2014, 2013, 2010; Cordy et al., 2012a,c; Delaware et al., 2011; Gazzillo and Grimm, 2012; Kästner et al., 2012a, 2011b; Liebig et al., 2013; Post and Sinz, 2008; Sabouri and Khosravi, 2012, 2013b; Schaefer et al., 2010b; Thüm et al., 2012]. Especially, sampling may require a considerable amount of time [Liebig et al., 2013]. In summary, for empirical evaluations, the performance of each step should be documented to improve comparability.

Product-Line Implementation and Specification

In addition to the analysis strategy, we classified product-line analyses with respect to the underlying strategy for implementation and specification of variability. We distinguish between composition-based and annotation-based implementations, and between domain-independent, family-wide, product-based, feature-based, and family-based specifications. In Table 3.2, we give an overview of which analysis strategies have been applied to which kind of implementation and specification strategy, respectively.

Analysis strategy	**Implementation**	Composition-based	Annotation-based	**Specification**	Domain-independent	Family-wide	Product-based	Feature-based	Family-based
Unoptimized product-based		7	3		2	3	0	5	0
Optimized product-based		8	2		3	3	0	5	0
Family-based		19	43		31	14	0	7	8
Feature-based		0	0		0	0	0	0	0
Feature-product-based		17	0		8	5	0	6	0
Feature-family-based		3	0		2	0	0	1	0
Family-product-based		2	2		1	2	0	0	0
Feature-family-product-based		0	0		0	0	0	0	0
Total*		54	49		45	26	0	23	8

*The bottom row is not necessarily the sum of all above rows, because some specification approaches are used with several analysis strategies. Furthermore, some analysis approaches are available for both, annotation-based and composition-based implementations.

Table 3.2: Number of approaches for each combination of analysis, specification, and implementation strategy.

The majority of implementation and specification strategies discussed in our survey have actually been applied. Both composition-based and annotation-based implementations have been used with similar frequency in the literature. In contrast, most approaches built on domain-independent specifications. This is natural, as many approaches consider type checking or static analysis, for which specifications are often defined independently of a particular system. In addition, many other specifications are family-wide, which means that, while the implementation contains variability, the specification does not. About the same number of approaches rely on feature-based specifications, which support variability similar as with composition-based implementation. However, we found only eight approaches using family-based specification [Asirelli et al., 2012; Classen et al., 2014, 2013; Cordy et al., 2013a, 2012a,b, 2013b, 2014], and none with product-based specification. An open research question is how much variability is required in product-line specifications (e.g., whether feature-based specifications are sufficient [Apel et al., 2013c]) and whether there are differences in the variability of specifications depending on the underlying software analysis. Model checking is the only software analysis to which all specification strategies (except product-based specification) have already been applied (cf. Section 3.3). For type checking, domain-independent specification are sufficient, but other specification strategies shall be explored for static analysis and theorem proving.

While the strategies for implementation, specification, and analysis seem to be largely independent of each other, we discuss some findings based on our classification. First, product-based specifications are problematic not only from a reuse perspective, but also

	Composition-based	Annotation-based
Unoptimized product-based	Apel et al. [2008a]	Buchmann and Schwägerl [2012]
Optimized product-based	Jayaraman et al. [2007]	Liebig et al. [2013]
Family-based	Thaker et al. [2007], Kim et al. [2008], Kuhlemann et al. [2009], Apel et al. [2010a], Apel et al. [2010d], Alférez et al. [2011], Kolesnikov et al. [2013], Schröter et al. [2014]	Aversano et al. [2002], Czarnecki and Pietroszek [2006], Huang et al. [2007], Metzger et al. [2007], Kim et al. [2008], Post and Sinz [2008], Heidenreich [2009], Kenner et al. [2010], Teixeira et al. [2011], Kästner et al. [2012a], Kästner et al. [2012b], Le et al. [2013], Liebig et al. [2013], Chen and Erwig [2014], Chen et al. [2014]
Feature-product-based	Apel and Hutchins [2010], Klose and Ostermann [2010], Bettini et al. [2013], Istoan [2013], Kolesnikov et al. [2013], Bettini et al. [2015]	
Feature-family-based	Delaware et al. [2009], Damiani and Schaefer [2012]	

Table 3.3: Classification of approaches for product-line type checking.

for analysis efficiency, because we can hardly reuse verification effort if specifications are not reused at all. Hence, product-based specifications should be avoided whenever possible. Second, for annotation-based implementations or family-based specifications there is not a single approach including a feature-based analysis. Clearly, we cannot analyze a feature in isolation if its implementation or specification scattered in the product line. However, future research should investigate how to extract feature implementation and specification from an annotation-based implementation to enable modular analysis, for which emergent interfaces [Ribeiro et al., 2010] are a first step. Finally, product-line specifications are used in several approaches not covered in our survey (e.g., [Johnsen et al., 2012; Kim et al., 2013; Thüm et al., 2012]). The reason is that such specification approaches have not been proposed in the context of an analysis that operates statically. Consequently, to better understand the strategies for product-line specification, a survey dedicated to specification rather than analysis should be performed.

Product-Line Type Checking

In Table 3.3, we summarize the strategies that have been applied to type checking. We identified product-based, family-based, feature-product-based, and feature-family-based approaches, whereas the majority of work is on family-based type checking. While it is unclear whether any useful properties can be analyzed with feature-based type

	Composition-based	Annotation-based	Domain-independent	Family-wide	Feature-based
Unoptimized product-based	Klaeren et al. [2001], Scholz et al. [2011]				Klaeren et al. [2001], Scholz et al. [2011]
Optimized product-based	Katz [2006]	Liebig et al. [2013]	Liebig et al. [2013]	Liebig et al. [2013]	Katz [2006]
Family-based	Adelsberger et al. [2014]	Ribeiro et al. [2010], Bodden et al. [2013], Brabrand et al. [2013], Liebig et al. [2013], Kanning and Schulze [2014], Midtgaard et al. [2014], Nguyen et al. [2014a], Sabouri and Khosravi [2014]	Ribeiro et al. [2010], Liebig et al. [2013], Midtgaard et al. [2014], Nguyen et al. [2014a]	Bodden et al. [2013], Brabrand et al. [2013], Liebig et al. [2013], Sabouri and Khosravi [2014]	
Family-product-based	Kim et al. [2010], Kim et al. [2011]	Kim et al. [2011], Shi et al. [2012]	Shi et al. [2012]	Kim et al. [2010], Kim et al. [2011]	

Table 3.4: Classification of approaches for product-line static analysis.

checking, future research should propose and evaluate approaches pursuing a family-product-based and feature-family-product-based strategy.

For type checking, there are no empirical evaluations for feature-family-based type checking. This strategy should be compared to existing approaches for family-based type checking to assess its potential. In particular, it is not clear how much time is needed to analyze features in isolation compared to the overall analysis time. An open research questions is whether the feature-family-based strategy is faster than the family-based strategy for evolving product lines. Similarly, it is to be assessed empirically whether the feature-family-based strategy requires more or less memory.

Furthermore, there are two competing approaches for family-based type checking, called local and global approaches [Apel et al., 2010d; Huang et al., 2011]. The main difference is whether the whole product line is encoded as a single or a large number of satisfiability problems. However, empirical evaluations are missing that compare time and space efficiency of both approaches.

Product-Line Static Analysis

In Table 3.4, we give an overview of static analyses for software product lines. The majority of approaches has been published in the last four years. So far, only product-based, family-based, and family-product-based strategies have been considered, which naturally raises the question of whether other strategies can be applied to static analysis. Interestingly, the family-product-based strategy has been applied exclusively to static analysis. In particular, feature-product-based and feature-family-based strategies, as known from other analyses, have not yet been applied. It is an open research question whether static analyses can take advantage of compositional properties.

All approaches for family-based static analysis are based on implementations using preprocessors and domain-independent specifications. Thus, future research should evaluate whether it is possible to create family-based static analysis for composition-based implementations and how to define family-wide, feature-based, and family-based specifications for static analysis.

Family-based static analyses have been compared empirically with optimized [Liebig et al., 2013] and unoptimized [Bodden et al., 2013; Brabrand et al., 2013] product-based analyses. Comparisons include time efficiency [Bodden et al., 2013; Brabrand et al., 2013; Liebig et al., 2013], memory efficiency [Brabrand et al., 2013], and soundness [Bodden et al., 2013]. In particular, Bodden et al. [2013] measured that it is faster to ignore than to incorporate the feature model during static analysis. Further studies shall evaluate whether this is the case for all kinds of static analysis and explore the fundamental reasons. This is especially interesting, as opposite experience has been made with model checking [Classen et al., 2013].

Product-Line Model Checking

In Table 3.5, we present strategies applied to scale model checking to product lines. In 2001, the first approach for model checking has been proposed pursuing a feature-product-based strategy. However, since then, mainly family-based approaches have been developed and several unoptimized product-based approaches. Compared to type checking, there is not a single approach for feature-family-based model checking. Hence, the research question arises whether this strategy can be applied to model checking, and, if so, what are the benefits of such an approach. Similar research questions can be formulated for all other "missing" strategies.

As for type checking, most empirical evaluations compare family-based model checking with product-based model checking. For feature-product-based model checking there is only one evaluation using a product line with four products [Liu et al., 2011]. Further empirical evaluations are needed with larger product lines that also compare feature-product-based with family-based model checking.

Product-Line Theorem Proving

In Table 3.6, we summarize the strategies used for theorem proving. Compared to type checking and model checking, there are fewer approaches for theorem proving,

	Composition-based	Annotation-based	Domain-independent	Family-wide	Feature-based	Family-based
Unoptimized product-based	Ubayashi and Tamai [2002], Apel et al. [2010c], Bessling and Huhn [2014]	Kishi and Noda [2006], Fantechi and Gnesi [2008]		Ubayashi and Tamai [2002], Kishi and Noda [2006], Fantechi and Gnesi [2008]	Apel et al. [2010c], Bessling and Huhn [2014]	
Optimized product-based	Plath and Ryan [2001], Katz [2006], Cordy et al. [2012d], Apel et al. [2013d], Lochau et al. [2014]			Cordy et al. [2012d], Lochau et al. [2014]	Plath and Ryan [2001], Katz [2006], Apel et al. [2013d]	
Family-based	Apel et al. [2011], Sabouri and Khosravi [2012], Apel et al. [2013d], Classen et al. [2013], Greenyer et al. [2013], Sabouri and Khosravi [2013a], Classen et al. [2014], Dubslaff et al. [2014], Thüm et al. [2014]	Fischbein et al. [2006], Gruler et al. [2008], Post and Sinz [2008], Classen et al. [2010], Lauenroth et al. [2010], Schaefer et al. [2010b], Asirelli et al. [2012], Cordy et al. [2012a], Cordy et al. [2012b], Cordy et al. [2012c], Classen et al. [2013], Cordy et al. [2013a], Cordy et al. [2013b], Sabouri and Khosravi [2013], Cordy et al. [2013], Sabouri and Khosravi [2014], Shi et al. [2014], ter Beek and de Vink [2014]	Post and Sinz [2008], Sabouri and Khosravi [2013a]	Fischbein et al. [2006], Gruler et al. [2008], Schaefer et al. [2010b], Cordy et al. [2012c], Sabouri and Khosravi [2012], Greenyer et al. [2013], Sabouri and Khosravi [2013b], ter Beek et al. [2013], Sabouri and Khosravi [2014], Shi et al. [2014], ter Beek and de Vink [2014]	Classen et al. [2010], Lauenroth et al. [2010], Apel et al. [2011], Apel et al. [2013d], Dubslaff et al. [2014], Thüm et al. [2014]	Asirelli et al. [2012], Cordy et al. [2012a], Cordy et al. [2012b], Classen et al. [2013], Cordy et al. [2013a], Cordy et al. [2013b], Classen et al. [2014], Cordy et al. [2014]
Feature-product-based	Fisler and Krishnamurthi [2001], Nelson et al. [2001], Li et al. [2002], Blundell et al. [2004], Li et al. [2005], Liu et al. [2011], Istoan [2013]		Istoan [2013]	Fisler and Krishnamurthi [2001], Nelson et al. [2001], Blundell et al. [2004], Liu et al. [2011], Istoan [2013]	Li et al. [2002], Li et al. [2005], Istoan [2013]	

Table 3.5: Classification of approaches for product-line model checking.

	Composition-based	Domain-independent	Feature-based
Unoptimized product-based	Harhurin and Hartmann [2008]		Harhurin and Hartmann [2008]
Optimized product-based	Bruns et al. [2011], Hähnle et al. [2013]		Bruns et al. [2011], Hähnle et al. [2013]
Family-based	Thüm et al. [2012], Thüm et al. [2014]		Thüm et al. [2012], Thüm et al. [2014]
Feature-product-based	Batory and Börger [2008], Delaware et al. [2011], Thüm et al. [2011b], Damiani et al. [2012], Delaware et al. [2013]	Delaware et al. [2011], Delaware et al. [2013]	Batory and Börger [2008], Thüm et al. [2011b], Damiani et al. [2012]
Feature-family-based	Hähnle and Schaefer [2012]		Hähnle and Schaefer [2012]

Table 3.6: Classification of approaches for product-line theorem proving.

suggesting that this research field is underrepresented. Surprisingly, the family-based strategy has not been applied to theorem proving, whereas this strategy has been applied often to type checking and model checking. Based on these insights, we propose two new approaches in Chapter 5.

For theorem proving, there is a lack of reliable evaluations comparing the strategies to each other. There is only one empirical evaluation, besides our evaluations presented in Chapter 5. Delaware et al. [2011] measured the time needed for the feature-based and the product-based part in feature-product-based theorem proving.

3.7 Related Classifications and Surveys

We found classifications and surveys related to ours and describe commonalities and differences in the following.

Classifications for Quality Assurance in Software Product Lines

Pohl et al. [2005] discuss four strategies for product-line testing. In contrast to our classification, they discuss strategies incorporating tests at different levels including unit tests, integration tests, and system tests. The brute force strategy is similar to unoptimized product-based analysis, but tests are performed at all levels for all products. In contrast, for the pure application strategy only delivered products are tested in application engineering. The sample application strategy is equivalent to the sample-based strategy in our classification. Finally, with the commonality and reuse strategy artifacts common to all products are tested in domain engineering and then

all products are tested separately. These strategies have been defined for product-line testing and do not represent all strategies that we identified in our survey.

Similarly, Metzger [2007] and Lauenroth et al. [2010] discuss three strategies for quality assurance (e.g., model checking) of product lines, namely commonality strategy, sample strategy (similar to sample-based analysis), and comprehensive strategy (similar to unoptimized product-based analysis). The idea of the commonality strategy is to check artifacts that are common to all products. Similar to the family-based strategy, the commonality strategy uses the feature model and domain artifacts to retrieve the common artifacts. Similar to the feature-based strategy, it can only uncover certain faults for a given product line. The commonality strategy is not represented in our classification. However, we have not found any approaches applying this strategy.

Lutz [2007] classifies approaches for product-line verification and validation with respect to the software development life-cycle. In particular, he distinguishes requirements, safety requirements, architecture, design, and implementation. We made the experience that many approaches cannot uniquely be assigned to one of these classes. For example, most approaches for model checking are applicable to architecture, design, and implementation.

It is worthwhile to note that two extensions of our classification have already been proposed. In the first extension, we explore modeling product-line analyses by means of three dimensions: sampling, feature grouping, and variability encoding [von Rhein et al., 2013]. The difference to the taxonomy discussed in the previous sections is that there is a continuum along these dimensions. The dimension variability encoding ranges from a product-based to a family-based analysis and the dimension feature grouping ranges feature-based to product-based analyses. Orthogonal to these dimensions, each approach may use a certain degree of sampling ranging from a single product or feature to all products or features. The value of this extension is that it may lead to new approaches in the future. Furthermore, it is equipped with graphical and formal notations. However, besides classifying some example approaches, there is no literature survey applying this extension.

In the second extension, Benduhn [2014] builds on our classification and survey to distinguish product-line representations proposed in the research literature, such as product-line implementation, modeling, and specification. He unifies our classifications of implementations and specifications into a general classification to assess similarities and differences across representations. This classification was used to survey and classify approaches with a slightly different focus than in this chapter. First, analyses that do not require a specification for each product line were omitted (e.g., type checking). Second, he incorporates dynamic analyses and testing, which were not in our scope. Compared to this chapter, the focus is rather on how to specify and model the expected behavior, rather than to verify and analyze product lines.

Surveys on Quality Assurance in Software Product Lines

Benavides et al. [2010] survey automated analyses for feature models. These analyses consider only the feature model and can detect anomalies such as dead features or com-

pute statistics such as the number of products. In contrast, our focus is on approaches that operate on source code or models thereof. However, many of the approaches in our survey rely on techniques from this line of research to reason about variability (e.g., for the family-based strategy).

Furthermore, numerous surveys on product-line testing have been conducted in the last decade [Carmo Machado et al., 2014; Da Mota Silveira Neto et al., 2011; Engström and Runeson, 2011; Lee et al., 2012; Oster et al., 2011; Tevanlinna et al., 2004]. These surveys are complementary to ours, because we focus on approaches that operate statically and they focus on dynamic analysis and test execution. Nevertheless, our classification could also be applied to testing. While we started to apply our classification to testing approaches, it seems that most approaches for product-line testing are sample-based analyses. However, researchers recently proposed approaches for family-based testing [Kästner et al., 2012c; Kim et al., 2012, 2013; Nguyen et al., 2014a,b].

Montagud and Abrahão [2009] performed a systematic literature review on quality assessment of software product lines. They distinguish between quality assessment applied in domain engineering and application engineering. Etxeberria et al. [2008] presented a survey that additionally incorporates feature modeling, design, architecture, implementation, and testing. In contrast to both reviews, we focus only on product-line analysis that operate statically, our classification is more fine-grained, and we survey more approaches. Furthermore, we derived a research agenda based on our insights.

3.8 Summary

In software-product-line engineering, similar software products are built in an efficient and coordinated manner based on reusable artifacts. While there are efficient techniques to implement software product lines, current research seeks to scale software analyses, such as type checking, static analyses, model checking, and theorem proving, from single software products to entire software product lines. The field of product-line analysis is broad and diverse, and different approaches are often hard to compare.

We propose a classification of product-line analyses into three main analysis strategies: product-based, feature-based, and family-based analyses. These strategies indicate how the analysis handles software variability, and can be even combined, resulting in four further strategies: feature-product-based, feature-family-based, family-product-based, and feature-family-product-based analyses. Besides the analysis strategy, we classify approaches with respect to the implementation and specification strategy. We identified four specification strategies that have been applied in the literature: domain-independent, family-wide, feature-based, and family-based specifications.

Overall, we classified 137 existing analysis and specification approaches, gaining insights into the field of product-line analyses. First, whereas many approaches claim to be compositional, we distinguish feature-product-based and feature-family-based strategies to reveal how inherently non-compositional properties such as feature interactions are analyzed. Second, not all strategies have been applied to all software analyses.

For example, we have not found feature-product-based static analyses, feature-family-based static analyses, family-based theorem proving, and feature-family-based model checking. Third, we identified well-represented (e.g., family-based type checking, static analysis, and model checking) and underrepresented research areas (e.g., optimized product-based analyses and feature-family-based theorem proving). Finally, there is no compositional analysis for annotation-based product lines or family-based specifications.

Based on these insights, we formulated research questions to be addressed in future work. With the following two chapters, we overcome some limitations of prior work. In Chapter 4, we discuss how to specify product lines by means of contracts. Furthermore, we propose approaches for feature-product-based and family-based theorem proving in Chapter 5.

We hope this chapter can raise awareness of the importance and challenges of product-line analyses, initiate a discussion on the future of product-line analyses, motivate researchers to explore and practitioners to use product-line analysis methods, and help to form a community of researchers, tool builders, and users interested in product-line analyses. We refer interested readers to our website to follow the progress of our ongoing classification effort.[2]

---

[2]http://fosd.net/spl-strategies/

4. Feature-Oriented Contracts for Product-Line Specification

This chapter shares material with the FASE'12 paper "Applying Design by Contract to Feature-Oriented Programming" [Thüm et al., 2012]. Initial ideas have been presented at VAST'11 [Thüm et al., 2011b] and FOSD'11 [Scholz et al., 2011].

As indicated in the previous chapter, most verification techniques for product lines require a specification of the expected behavior of all products. In product-line verification, we analyze whether all products of the product line adhere to their specifications. While there exist many approaches to specify product lines, they are often just used as proof-of-concept for verification techniques and not justified empirically. To the best of our knowledge, this chapter presents the first systematic discussion and evaluation of *how to specify product lines*. Our results are a foundation for analyses of product lines that rely on specifications, such as formal verification, feature-interaction detection, or test-case generation. Indeed, we build on this foundation when we investigate product-line verification in the next chapter.

Our investigation of product-line specifications is *based on design by contract* for various reasons. First, contracts enable the formal specification of behavior and, thus, can be used for a wide range of verification techniques, such as theorem proving [Barnett et al., 2011; Beckert et al., 2007; Burdy et al., 2005; Hatcliff et al., 2012], model checking [Robby et al., 2006], static analysis [Burdy et al., 2005; Hatcliff et al., 2012], runtime assertion checking [Barnett et al., 2011; Burdy et al., 2005; Hatcliff et al., 2012; Meyer, 1988], and test-case generation [Burdy et al., 2005; Hatcliff et al., 2012]. Consequently, our findings inherently have many applications. Second, contracts help to identify the location of defects by means of blame assignment [Hatcliff et al., 2012; Meyer, 1988]. For example, a violation of a postcondition is the fault of a method itself and we should

correct the method implementation or the postcondition. We expect defect localization to be especially helpful when developing product lines with large development teams, in which no developer knows the complete code base. Third, design by contract is a means for specifying detailed designs. Once we understand the variability mechanisms required for specifications at code-level, we can use this knowledge to guide the development of product-line specification techniques for more abstract specifications (e.g., transition systems) or even abstraction mechanisms, such as model-based refinement methods (e.g., ASM [Börger and Stark, 2003] and Event-B [Abrial, 2010]).

We investigate contracts for product lines implemented by means of *feature-oriented programming*, to simplify the transfer of our results to other product-line implementation techniques. The reason is that feature-oriented programming contains only core variability mechanisms that can be encoded in many other implementation techniques. For instance, feature-oriented method refinement can be expressed using the around advice in aspect-oriented programming [Apel et al., 2008b], using method modifications in delta-oriented programming [Schaefer et al., 2010a], and using presence conditions in preprocessor-based product lines [Kästner et al., 2009a]. Consequently, when identifying variability patterns for feature-oriented method refinements, we can directly apply these patterns to other implementation techniques. A further reason for using a composition-based rather than an annotation-based technique is that it enables feature-based analysis techniques per-se and does not require a preceding family-based analysis step to extract modules (cf. Section 3.6).

While both, design by contract and feature-oriented programming, have been hot research topics for more than two decades, their combination is not straightforward and has not been addressed prior to this thesis. A key question is how to define and compose contracts when applying feature-oriented method refinements. For instance, we may assume behavioral subtyping [America, 1991; Dhara and Leavens, 1996; Hatcliff et al., 2012; Liskov and Wing, 1994; Meyer, 1988], which is often assumed for object-oriented inheritance, but this might be too restrictive for method refinements. In Section 4.1, we present a taxonomy of fundamental options for contract composition and then propose several mechanisms to implement these options in Section 4.2. In Section 4.3, we discuss how these contract-composition mechanisms can be extended for specification concepts beyond preconditions and postconditions (e.g., class invariants, assignable clauses, and specification cases). We describe our tool support for the specification of feature-oriented contracts in Section 4.4. Finally, we evaluate all contract-composition mechanisms empirically in Section 4.5 and discuss related work on contracts in aspect-oriented and delta-oriented programming in Section 4.6.

4.1   A Taxonomy for Contract Composition

Before actually integrating contracts into feature-oriented programming, we explore fundamental options for contract composition. Our consideration is more general than feature-oriented programming and can also be applied to contract composition for object-oriented method overriding, aspect-oriented around advice, delta-oriented

method modification, mixins, and traits. Whenever we compose two methods, the question is how to compose their contracts and which properties does such a composition establish. While we focus on composition of methods, we can even derive conclusions for annotation-based product lines, in which methods and their contracts may contain variability as well. First, we discuss interesting properties of contract-composition mechanisms in Section 4.1.1. Second, we discuss four fundamental options for contract composition based on these properties in Section 4.1.2.

4.1.1 Properties of Contract Composition

With *contract composition*, we refer to the process of retrieving a contract for a method given a list of contracts as input. Contract composition is motivated by the decomposition of methods on code level. Methods are decomposed into parts to achieve separation of concerns [Harrison and Ossher, 1993; Kiczales et al., 1997; Tarr et al., 1999] or to enable the automatic composition based on requirements [Apel et al., 2013a; Batory et al., 2004; Czarnecki and Eisenecker, 2000; Prehofer, 1997]. For both applications, the question arises how to decompose and compose contracts accordingly. That is, for each possible composition of methods, we are interested in the behavior in terms of a contract. While we may want to compose more than two contracts, our consideration is based on the composition of two contracts. This is sufficient, because a composition of more than two contracts can be simulated by several binary compositions. This simplification is in line with work on software composition, which is also often defined as a binary function [Apel et al., 2010b; Batory et al., 2004; Dhara and Leavens, 1996; Hatcliff et al., 2012; Höfner and Möller, 2009; Liskov and Wing, 1994; Meyer, 1988; Prehofer, 1997].

An interesting property is whether contract composition is commutative (i.e., whether the order of composed contracts matters). Contract composition is closely related to the composition of source code, because contracts are typically embedded in source code [Meyer, 1988]. Commutativity can facilitate comprehension, even if the composition of source code is often not commutative, because method refinements may refer to previous method implementations [Apel et al., 2010b; Höfner and Möller, 2009]. Usually, there is a special keyword to do so; for example, keyword super or Precursor in object-oriented method overriding [Bracha and Cook, 1990; Gosling et al., 2005; Meyer, 1988], keyword proceed (formerly runNext) in aspect-oriented around advice [Kiczales et al., 2001], and keyword original in feature-oriented [Apel et al., 2013b] and delta-oriented programming [Schaefer et al., 2011]. Hence, approaches enabling the composition of methods typically assume a partial order (i.e., a total order for each composition). For instance, delta modules in delta-oriented programming have to declare a partial order to all other modules refining the same methods [Schaefer et al., 2010a]. In feature-oriented programming, usually a total order on all features is assumed [Prehofer, 1997; Thüm et al., 2014b]. In aspect-oriented programming, aspect precedence can be defined and if the order is not unique, the aspect compiler chooses an order [Kiczales et al., 2001]. In object-oriented programming, an order is given by the inheritance hierarchy. Conse-
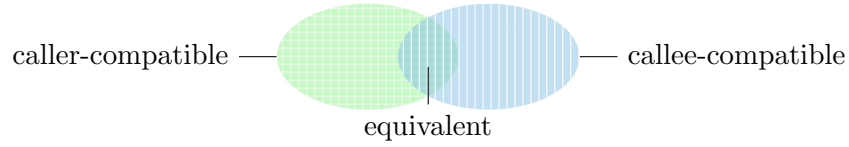
Figure 4.1: Compatibility of changed contracts for callers and callees.

quently, we can assume an order of composed contracts. We refer to the contract that is subject to refinement as *original contract* and its refinement as *refining contract*.

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

Definition 4.1. *Given two contracts $c_1 = \{\phi_1\}m_1\{\psi_1\}$ and $c_2 = \{\phi_2\}m_2\{\psi_2\}$.*

- *The contract c_2 is called* caller-compatible *with respect to c_1, if and only if $\phi_1 \models \phi_2$ and $\psi_2 \models \psi_1$, and* caller-incompatible *otherwise.*

- *The contract c_2 is called* callee-compatible *with respect to c_1, if and only if $\phi_2 \models \phi_1$ and $\psi_1 \models \psi_2$, and* callee-incompatible *otherwise.*

- *If and only if contract c_2 is both, callee-compatible and caller-compatible with respect to c_1, then c_2 is called* equivalent *to c_1, and* non-equivalent *otherwise.*

We illustrate these definitions by means of a Venn diagram in Figure 4.1. Assuming a fixed contract c_1, the Venn diagram illustrates the compatibility of all possible contracts c_2 with respect to c_1. Considering the caller view, it seems beneficial if the result of contract composition c_2 is caller-compatible, because all callers relying on c_1 can rely on c_2 instead. In contrast, callee-compatibility is desirable, because callees do not need to be aware of contract changes. However, requiring both properties enables only changes to contracts in which preconditions and postconditions remain equivalent (i.e., $\models \phi_1 \Leftrightarrow \phi_2$ and $\models \psi_1 \Leftrightarrow \psi_2$).

Given an original contract $c = \{\phi\}m\{\psi\}$ and a refining contract $c' = \{\phi'\}m'\{\psi'\}$, we denote the composed contract as $c'' = c' \bullet c = \{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\} = \{\phi''\}m' \bullet m\{\psi''\}$. A specific mechanism for contract composition defines how ϕ'' and ψ'' are derived from the contracts c and c'. We consider a *contract-composition mechanism M* as a total function $\bullet_M \colon C \times C \to C$ defined over the set C of all possible contracts. If it is clear from the context which contract-composition mechanism is meant, we write $\bullet$

Preservation property	Compatibility for	Compatibility to
original-caller-preserving	callers	original contract
refinement-caller-preserving	callers	refining contract
original-callee-preserving	callees	original contract
refinement-callee-preserving	callees	refining contract

Table 4.1: Compatibility of contracts for different preservation properties.

instead of $\bullet_M$ and thus overload the composition operator for method implementations (cf. Section 2.2.2). Clearly, the set C is specific to a certain specification language, whereas we discuss contract composition independently of a particular language. Still, we require that each contract $c \in C$ can be formulated as $c = \{\phi\}m\{\psi\}$, where ϕ and ψ are logical expressions being the precondition and postcondition, and m is the method to which the contract c applies to.

Based on caller-compatibility and callee-compatibility, we define preservation properties for contract-composition mechanisms that indicate to which extent it maintains the compatibility with the original and refining contract. Compared to our previous definitions of compatibility, we quantify over all possible contracts as input and classify the mechanism rather than single contracts. We define the following four *preservation properties* (cf. Table 4.1), which all facilitate some form of modular reasoning:

Definition 4.2. *Assume that a contract-composition mechanism m composes an original contract c with a refining contract c′ to the resulting contract c″ = c′ $\bullet_m$ c.*

- *The mechanism m is called* original-caller-preserving, *if for all $c, c' \in C$ the resulting contract c″ is caller-compatible with respect to c.*

- *The mechanism m is called* refinement-caller-preserving, *if for all $c, c' \in C$ the resulting contract c″ is caller-compatible with respect to c′.*

- *The mechanism m is called* original-callee-preserving, *if for all $c, c' \in C$ the resulting contract c″ is callee-compatible with respect to c.*

- *The mechanism m is called* refinement-callee-preserving, *if for all $c, c' \in C$ the resulting contract c″ is callee-compatible with respect to c′.*

In object-oriented programming, original-caller-preserving contract composition for inheritance is already known as subcontracting [Meyer, 1988] and behavioral subtyping [Dhara and Leavens, 1996; Hatcliff et al., 2012; Liskov and Wing, 1994]. Nevertheless, we introduce a new name for it as many composition mechanisms are orthogonal to object-oriented inheritance. For example, aspect-oriented programming, delta-oriented programming, and feature-oriented programming can be seen as extensions of object-oriented programming and do not aim to completely replace inheritance [Kiczales et al., 1997; Prehofer, 1997; Schaefer et al., 2010a]. As a consequence, when applying contracts to these techniques, we need a mechanism for composing contracts in inheritance

as well as a mechanism for aspects, delta modules, and feature modules. In particular, we might want to chose different mechanisms for modularization with inheritance and the modularization of cross-cutting concerns. For this reason, we decided to introduce original-caller-preserving composition as a new name.

Besides the four preservation properties, other interesting properties of contract composition are commutativity, associativity, and idempotence.[1] Contract composition is *commutative*, if the order of contracts in composition does not matter (i.e., $c' \bullet c \equiv c \bullet c'$). Contract composition is *associative*, if different parentheses in the composition of more than two contracts result in equivalent contracts (i.e., $c'' \bullet (c' \bullet c) \equiv (c'' \bullet c') \bullet c$). Contract composition is *idempotent*, if the composition of two identical contracts yields an equivalent contract (i.e., $c \bullet c \equiv c$). Commutativity, associativity, and idempotence are desirable properties for contract composition, because they may ease the understanding of contracts as the order, parentheses, and identical contracts do not influence resulting contracts.

4.1.2   Four Fundamental Options for Contract Composition

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

Theorem 4.3. *There is no contract-composition mechanism that is both, original-caller-preserving and refinement-callee-preserving.*

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

---

[1]Our discussion of properties is independent of the composition of implementations. That is, contract composition can be commutative, even if the composition of source code is typically not commutative [Apel et al., 2010b; Höfner et al., 2012].
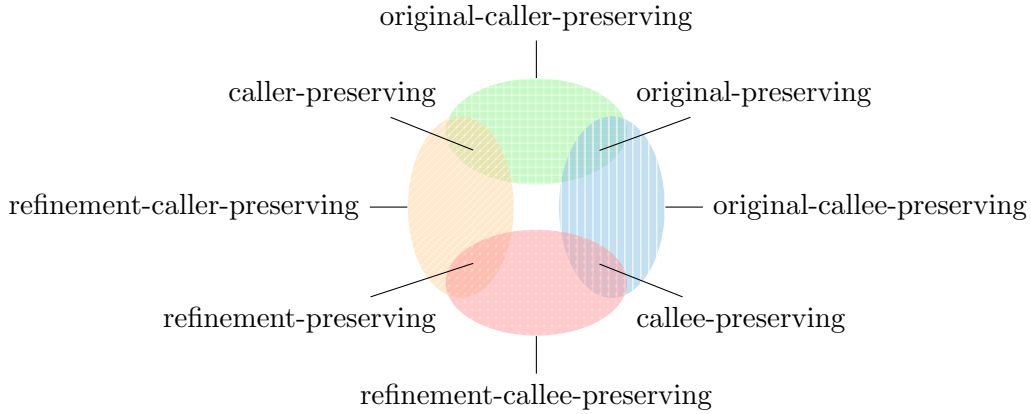
Figure 4.2: Contract-preservation properties indicate compatibility for callers and callees of original and refining contracts.

know that $\phi'' \models \phi$ and $\psi \models \psi''$. Because mechanism m is refinement-callee-preserving, we know that c'' is callee-compatible with c', and thus $\phi' \models \phi''$ and $\psi \models \psi''$. However, with $\phi' \models \phi''$ and $\phi'' \models \phi$ it follows that $\phi' \models \phi$, which is a restriction on the contracts that are composed. Thus the properties do not hold for all contracts c and c', which is a contradiction to our assumption that the mechanism is both, original-caller-preserving and refinement-callee-preserving. $\square$

Theorem 4.4. *There is no contract-composition mechanism that is both, original-callee-preserving and refinement-caller-preserving.*

Proof. Analogous to Theorem 4.3. $\square$

By means of Theorem 4.3, we know that a contract-composition mechanism cannot return a contract c'' that is caller-compatible with respect to c and callee-compatible with respect to c' for all $c, c' \in C$. Nevertheless, the proof indicates that if we would restrict our contracts $c = \{\phi\}m\{\psi\}$ and $c' = \{\phi'\}m'\{\psi'\}$ to fulfill $\phi' \models \phi$ and $\psi \models \psi'$, the resulting contract c'' could be caller-compatible with respect to c and callee-compatible with respect to c'. However, such restriction on the input contracts for contract composition are not in our scope and would hinder compositionality, because we would have to check that all possible compositions of contracts fulfill the restrictions on preconditions and postconditions.

With Theorem 4.3 and Theorem 4.4, we conclude that at most two of the four preservation properties can be fulfilled by a contract-composition mechanism. We illustrate the possible combinations of the preservation properties also using a Venn diagram in Figure 4.2. Overall, there are nine options for mechanisms with respect to the preservation properties. First, one option is to fulfill no preservation properties (white area in Figure 4.2). Second, there are four options to fulfill exactly one of these four properties (red, orange, green, and blue). Finally, fulfilling two out of four properties give rise to further four options (mixed colors). Theorem 4.3 rules out overlapping between green and red,

and Theorem 4.4 rules out overlapping between orange and blue. As explained above, fulfilling more of these preservation properties improves modular reasoning. Hence, of these nine options, the four options fulfilling two preservation properties seem to be most promising. For that reason, we refer to them as *fundamental options* for contract composition and introduce names for them:

Definition 4.5. *A contract-composition mechanism m is called* caller-preserving *if and only if it is original-caller-preserving and refinement-caller-preserving. A mechanism m is called* callee-preserving *if and only if it is original-callee-preserving and refinement-callee-preserving. A mechanism m is called* original-preserving *if and only if it is original-caller-preserving and original-callee-preserving. A mechanism m is called* refinement-preserving *if and only if it is refinement-caller-preserving and refinement-callee-preserving.*

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

4.2   Contract-Composition Mechanisms

Our formal taxonomy of contract composition and the properties discussed in the past section can generally be applied to any software composition supporting the composition of methods. In this section, we focus on feature-oriented programming as a particular technique for software composition. We already introduced feature-oriented programming in Section 2.2.2 and discuss how to integrate contracts into feature-oriented programming in the following. In particular, we propose six mechanisms for contract composition in feature-oriented programming, namely plain contracting, contract overriding, explicit contract refinement, conjunctive contract refinement, cumulative contract refinement, and consecutive contract refinement. These contract-composition mechanisms extend the composition of feature modules by support for contracts. All mechanisms are illustrated based on JAVA and JML, but are not restricted to them in principle. All examples used for illustration are excerpts of the product lines used in our evaluation and no fictive examples. As in the previous section, we concentrate on preconditions and postconditions for simplicity. Contract-based specification techniques beyond preconditions and postconditions are discussed in Section 4.3.

4.2.1 Plain Contracting

A simple mechanism to deal with contracts during feature-module composition is to apply the identity function to the original contract. That is, given an original contract c and a refining contract c', the result of contract composition is always the original contract c. We call this mechanism *plain contracting* [Thüm et al., 2012] and denote it as $\bullet_{pc}$ (i.e., $c' \bullet_{pc} c = c$). In plain contracting, the idea is to define a contract for each method, but to not allow their refinement. Instead of ignoring refining contracts during composition, the idea is rather to not define any contract refinements at all. Nevertheless, feature modules may contain method refinements if they establish the original contract.

Example 4.6. *In Listing 4.1, we show an excerpt of a product line that was developed by Wolfgang Scholz for feature-interaction detection [Scholz et al., 2011]. The product line* IntegerList *contains a method refinement of method* push*. In feature module* Base*, the method is introduced with a contract and simply inserts a given element at the end of the list. In feature module* Sorted*, the method is refined such that the list is sorted after each insertion. The method refinement adheres to the contract defined in feature module* Base *and is not refined in the optional feature module* Sorted.[2]

A rather technical design decision of contract composition, in general, and plain contracting, in particular, is how to handle methods without a contract during composition. In design by contract, the absence of a precondition means that there are no assumptions that the caller has to fulfill [Meyer, 1988], which is semantically equivalent to requires true. Analogously, the absence of a postcondition indicates that there is nothing the caller can rely on (i.e., ensures true). In the following, we call such a missing contract an *empty contract*, denoted as $c = \epsilon$. Given this semantics, the question for contract composition is whether or not such contracts are considered during composition. For instance, assume we want to compose the method m with methods m' and m'', of which m has no contract and the contracts of m' and m'' are c' and c'', respectively. If we ignore empty contracts during composition, the result for our example is $c'' \bullet_{pc} c' = c'$. However, if we treat the empty contract as a contract that is subject to composition, the result is $c'' \bullet_{pc} c' \bullet_{pc} c = \epsilon$. While both options are possible, they are largely independent of our discussion. We choose to ignore empty contracts during composition in the following, because a programmer can enforce the latter behavior by providing the trivial contract requires true; ensures true.

Properties

With respect to the properties discussed in Section 4.1.1, *plain contracting is original-preserving*, because the caller view and callee view of the original contract are both maintained. The reason is that caller and callee compatibility are reflexive. For any given contract c, c is caller-compatible and callee-compatible to c, due to the fact that

---

[2]We postpone a discussion on the role of class invariants in this example to Section 4.3.4.

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

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

Listing 4.1: *Plain contracting* in product line *IntegerList*: feature *Base* introduces a contract for method push, which is not refined in feature *Sorted*.

$\models$ is reflexive, too. Furthermore, plain contracting is idempotent as $c \bullet_{pc} c = c$ and associative as $(c'' \bullet_{pc} c') \bullet_{pc} c = c$ and $c'' \bullet_{pc} (c' \bullet_{pc} c) = c$. However, plain contracting is not commutative.

Advantages

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

about method calls, because the same contract holds for every possible combination of features.[3] This is beneficial since a programmer needs to know the contract for every called method (e.g., to find out whether the precondition is fulfilled at every position where the method is called).

Disadvantages

The downside of plain contracting is its restrictiveness. Method refinements may change the behavior only such that the original contract is maintained. For an example consider Listing 4.1 again; replacing the ascending sorting order by a descending order by means of a method refinement for method sort would not be allowed. In addition, even if the method refinement establishes the original contract, we cannot specify the changed behavior with the method refinement. As a consequence, callers cannot rely on the changed behavior. For instance, if we replace the instable sorting algorithms heap sort and quick sort by a stable algorithm, such as merge sort, we may want to express that callers can rely on stability, which is impossible with plain contracting.

4.2.2   Contract Overriding

Contract overriding is a contract-composition mechanism, which is complementary to plain contracting [Thüm et al., 2012]. In contract overriding, all methods and method refinements may provide a contract. During composition, the refining contract completely overrides the original contract (i.e., $c' \bullet_{co} c = c'$). Similar as for plain contracting, we assume that empty contracts are ignored during composition. Otherwise, we would need to repeat contracts for each method refinement, even if they do not require any changes to original contract.

Example 4.7. *In Listing 4.2, we give an example for contract overriding. The product line* GPL-scratch *was developed by André Weigelt to illustrate the need of different contract composition techniques in a single product line [Weigelt, 2013]. Feature module* Base *introduces a method* addEdge *that takes a given edge and inserts it into an existing graph. However, the edge need to be non-null and the nodes it connects must already exist in the graph. The feature module* MaxEdges *refines the method implementation such that only a specified number of edges can be added to the graph. That is, an edge is only inserted if the maximum number of edges will not be exceeded. The contract given in feature module* MaxEdges *is supposed to completely override the original contract.*

Properties

Contract overriding is a refinement-preserving contract-composition mechanism, because caller and callee view of the refining contract are maintained. As plain contracting, contract overriding is associative and idempotent, but not commutative. The argumentation for these properties is analogous to that for plain contracting.

---

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

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

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

Listing 4.2: *Contract overriding* in product line *GPL-scratch*: feature *MaxEdges* overrides the contract of feature *Base* (adapted from [Weigelt, 2013]).

Advantages

The main advantage of contract overriding over plain contracting is that contracts can be refined. Hence, when a method refinement provides some new guarantees, we can actually specify them in a refining contract and callers can rely on it. In addition, because we can arbitrarily refine contracts, also all method refinements are possible and do not have to adhere to the original contract. That all contract refinements and all method refinements are possible, provides flexibility particularly with respect to unanticipated changes.

Disadvantages

However, the flexibility of contract overriding also comes with disadvantages. Any contract may be subject to later refinement, which challenges callers. That is, *callers need to be aware of any contract* for a given method in order to determine the contract they can rely on. In particular, the composed contract heavily depends on the actual feature selection. In our example, precondition and postcondition of method addEdge depend on whether feature *MaxEdges* is selected or not. Hence, callers need to consider the valid feature combinations defined by the feature model, which may rule out certain combinations of contract refinements.

A further problem of contract overriding is that it may require to clone and adapt previous contracts. For example, the refined contract in Listing 4.2 repeats the complete precondition and postcondition of the original contract. In analogy to code clones [Roy et al., 2009], we refer to such cloned contracts as *specification clones*. Similar to code clones, we could distinguish several levels of specification clones, but such a discussion is out of our scope. The reason why we need to clone contracts is that contract overriding only supports to completely replace contracts without any mechanism to reuse existing contracts. Hence, the CPA (copy, paste, adapt) principle is the only option to refine contracts. We argue that specification clones have similar drawbacks as code clones. For instance, when updating a certain contract, we may forget to update clones of this contract and introduce inconsistencies. Hence, specification clones should be avoided whenever possible requiring more sophisticated mechanisms for contract composition.

Furthermore, if two or more features refine the same contract using contract overriding, we may get undesired contracts if both features are selected. This problem is already known from the implementation of feature modules as the optional feature problem [Kästner et al., 2009b; Liu et al., 2006], which we discussed in Section 3.1. A solution is to introduce *derivative contracts* (i.e., a contract that is only included if two or more features are selected). However, derivative contracts can cause further specification clones and may not scale for many contract refinements of the same method.

4.2.3 Explicit Contract Refinement

Explicit contract refinement [Thüm et al., 2012] is a contract-composition mechanism that tries to mitigate the issues of contract overriding with respect to specification clones and derivative contracts. Similar to contract overriding, explicit contract refinement permits contract refinements. That is, refining contracts override original contracts. However, refining contracts may refer to the original precondition and original postcondition in their precondition and postcondition, respectively. Similar to feature-oriented method refinement, we introduce keyword original in preconditions and postconditions, which are replaced by the original contract during composition. We define explicit contract refinement based on the composition of predicates: $\{\phi'\}m'\{\psi'\}\bullet_{ecr}\{\phi\}m\{\psi\} = \{\phi'\bullet\phi\}m'\bullet m\{\psi'\bullet\psi\}$, whereas $\phi'\bullet\phi$ is the result of replacing all occurrences of keyword original by ϕ in ϕ' and $\psi'\bullet\psi$ is defined analogously. The keyword is neither mandatory in preconditions nor in postcondition and may even appear several times in the same precondition or postcondition, respectively. In fact, contract overriding is a special case of explicit contract refinement where the keyword original is never used. The usage of keyword original has also been proposed for delta-oriented programming [Hähnle and Schaefer, 2012].

Example 4.8. *In Listing 4.3, we give an example for explicit contract refinement based on the previous example in Listing 4.2. The feature module* Base *is identical. However, instead of cloning precondition and postcondition of the original contract in feature module* MaxEdges*, we refer to them by means of keyword* original*. The result of composing both feature modules is exactly the same as for our example on contract overriding.*

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

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

Listing 4.3: *Explicit contract refinement* in product line *GPL-scratch*: feature *MaxEdges* refines a contract by referring to the original contract defined in feature *Base* (adapted from [Weigelt, 2013]).

Properties

Similar to contract overriding, explicit contract refinement is associative and not commutative. In contrast, explicit contract refinement is not idempotent, because replacing the keyword original may lead to different contracts. For example, precondition $\phi = \neg \text{original}$ is composed to $\phi \bullet \phi = \neg \text{original} \bullet \neg \text{original} = \neg \neg \text{original}$ with $\neg \phi \not\equiv \phi$.

According to our definition of preservation properties, explicit contract refinement does not establish any properties. First, the preservation properties regarding the original contract are not established, because the refining contract may arbitrarily replace the contract. For example, the result of composing an original contract $\{\text{true}\}m'\{\text{true}\}$ with a refining contract $\{\text{false}\}m'\{\text{false}\}$ is neither callee-compatible nor caller-compatible to the original contract, because true $\not\models$ false. Second, the preservation properties regarding the refining contract are not established, because keyword original is resolved at composition time and the definitions of preservation properties relate the refining contract (i.e., which may contain keyword original) with the result of composition (i.e., in which keyword original may be replaced). Similarly to the above example, the result of composing an original contract $\{\text{true}\}m'\{\text{true}\}$ with a refining contract $\{\text{original}\}m'\{\text{original}\}$ is neither callee-compatible nor caller-compatible to the refining contract, because true $\not\models$ original.

Advantages

Explicit contract refinement supports the refinement of contracts by the *same linguistic means* as method refinement, which is an intuitive approach for programmers familiar with feature modules, and thus could raise the acceptance of contracts in feature modules. That is, in feature-oriented programming, we may or may not refer to the original method implementation in a method refinement with keyword original. With explicit contract refinement, we may or may not refer to the original precondition or postcondition in a contract refinement with keyword original, too. Consequently, programmers may completely override, refine, or completely reuse the previous method implementation, precondition, and postcondition independently from each other.

Furthermore, explicit contract refinement overcomes some drawbacks of contract overriding. First, programmers can avoid some *specification clones*, because they have a linguistic means to refer to original preconditions and postconditions individually and do not have to clone and adapt contracts for all method refinements. Second, the need for *derivative contracts* is reduced as we can refer to a previous contract from whatever feature module this contract may be provided, and thus supporting compositional flexibility. Nevertheless, in some cases specification clones and derivative contracts may still be necessary (e.g., if we want to change just a small part within a precondition).

Disadvantages

As for contract overriding, *callers need to be aware of any contract* defined for a given method and its refinements. Even worse in explicit contract refinement, specification may even get more complex if several refinements for the same method contract exist and some, but not all refinements refer to the previous contracts. Furthermore, a new dimension of complexity arises due to the possibility to independently refer to preconditions and postconditions, and that it is even possible to negate preconditions or postconditions or to use them in some new logical context. Again, it may be hard for a programmer to retrieve the contract for a certain context, which, however, could be mitigated by means of tool support.

References to previous contracts introduce the possibility of dangling references. That is, we may use the keyword original in a contract, but during composition we detect that there is no original contract to which the keyword can point to. For example, we may define a method introduction including a contract in an optional feature and use the keyword original to refer to this contract from another optional method refinement. This is a particular instance of an unwanted feature interaction [Calder et al., 2003] (cf. Section 3.1). As typical for feature interactions, this dangling reference may only occur in some feature combinations and thus stay unnoticed until feature modules are composed for one of these combinations. However, we show how tool support can address this issue in Section 5.3.1.

While references of original method implementations is known from most approaches for software composition [Batory et al., 2004; Kiczales et al., 1997; Meyer, 1988; Prehofer, 1997], such as feature-oriented programming, aspect-oriented programming, and

object-oriented programming, it is uncommon for specifications. For instance, JML and SPEC# do not provide any keyword to refer to preconditions and postconditions of superclasses. Hence, it may be unintuitive for programmers familiar with existing contract languages to refer to other contracts. However, this is not a severe limitation and could also indicate missing language constructs in existing contract languages.

4.2.4   Conjunctive Contract Refinement

In contrast to explicit contract refinement, the next three contract-composition mechanisms that we discuss do not require any keywords to explicitly refer to previous contracts. We refer to these mechanisms also as *implicit contract refinement*, because how two given contracts are composed is only implicitly given by the mechanisms and not made explicit in the contract itself. One of these mechanisms for implicit contract refinement is *conjunctive contract refinement* [Thüm et al., 2012]. Given two contracts $c = \{\phi\}m\{\psi\}$ and $c' = \{\phi'\}m'\{\psi'\}$ their composition is determined by the conjunction of their preconditions and postconditions, respectively (i.e., $c' \bullet_{ConjCR} c = \{\phi' \wedge \phi\}m' \bullet m\{\psi' \wedge \psi\}$).

Example 4.9. *In Listing 4.4, we again show an excerpt of product line* GPL-scratch *to exemplify conjunctive contract refinement. The keyword* conjunctive_contract *indicates that the contracts for method* equals *are composed using conjunctive contract refinement. More details on this and other contract-composition keywords are postponed to Section 4.4.1. Each contract refinement contains a precondition and a postcondition that must be fulfilled in addition to all preconditions and postconditions defined in other feature modules. In particular, the optional feature module* Weighted *introduces a new field* weight*, for which method* equals *and its contract need to be refined accordingly, because edges are only considered equivalent if they have the same weight. A similar refinement is given in feature module* Directed*, which is considered alternative to feature module* Undirected *(not shown for brevity).*

Properties

Conjunctive contract refinement is associative, commutative, and idempotent, because it inherits these properties from the conjunction of predicates. However, it does not establish any preservation properties. The result of composition is neither caller-compatible with respect to c nor to c', because $\psi \nvDash \psi' \wedge \psi$ and $\psi' \nvDash \psi' \wedge \psi$. Similarly, the result of composition is neither callee-compatible with respect to c and c', because $\phi \nvDash \phi' \wedge \phi$ and $\phi' \nvDash \phi' \wedge \phi$. Nevertheless, the conjunction of postconditions is helpful for callers and the conjunction of preconditions is helpful for callees, but the combination is not sufficient for modular reasoning as discussed for the preservation properties in Section 4.1.1.

Advantages

Compared to explicit contract refinement, there is no specification effort with respect to providing the keyword original. At the same time, it is possible to avoid some

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

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

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

Listing 4.4: *Conjunctive contract refinement* in product line *GPL-scratch*: features *Weighted* and *Directed* refine a contract by adding a precondition and a postcondition to the original contract defined in feature *Base* (adapted from [Weigelt, 2013]).

specification clones and derivative contracts due to the fact that previous preconditions and postconditions are assumed to hold in all cases. Furthermore, resulting contracts are easy to understand as all preconditions and all postconditions are simply concatenated. That is, a later or potentially unknown feature module can only change the contract in this limited way.

Disadvantages

Nevertheless, the main disadvantage is the missing support for modular reasoning. Callers have to fulfill all preconditions and thus need to know all method refinements

in the refinement chain and their contracts in advance, which prohibits modular rea-
soning. Analogously, callees have to fulfill all postconditions. Both, precondition and
postcondition refinements make it impossible to understand, maintain, or reason about
a method call by considering one feature. Instead, programmers and verification tools
always need to consider all features of the refinement chain.[4]

A further disadvantage of conjunctive contract refinement is that only a limited form
of contract refinements can be expressed. In principle, contract refinements can only
add formulas to preconditions and postconditions in conjunction to existing ones. As a
result, we might have to remove some contracts to enable certain method refinements.
However, in the worst case, we may only be able to specify a small portion of the
product-line behavior, and thus only detect some errors of the product line. For exam-
ple, conjunctive contract refinement is too restrictive to specify the contract refinement
shown in Listing 4.2 on Page 58. Whether this is a severe restriction in practice will be
evaluated in Section 4.5.

4.2.5 Cumulative Contract Refinement

There is a further mechanism for implicit contract refinement, which we used for feature-
interaction detection [Scholz et al., 2011] and to which we refer to as *cumulative contract
refinement*. Compared to conjunctive contract refinement, the idea is to facilitate mod-
ular reasoning for callers similar to subcontracting in object orientation. Meyer [1988]
states that composed preconditions must be weaker or equal to original preconditions
and composed postconditions must be stronger or equal. He has proposed a simple lan-
guage rule that avoids checking the conformance using theorem proving: preconditions
are combined in a disjunction and postcondition in a conjunction (cf. Section 2.1.2).
Adopting this language rule to feature orientation leads us to the definition of cumula-
tive contract refinement as $\{\phi'\}m'\{\psi'\} \bullet_{CumCR} \{\phi\}m\{\psi\} = \{\phi' \vee \phi\}m' \bullet m\{\psi' \wedge \psi\}$. We
omit an example, as (a) we have not found a motivating use case during our evaluation
and (b) the only difference between cumulative contract refinement and conjunctive
contract refinement is the disjunction of preconditions.

Properties

Analogously to conjunctive contract refinement, cumulative contract refinement is as-
sociative, commutative, and idempotent, as it inherits these properties from disjunction
and conjunction. Furthermore, cumulative contract refinement is caller-preserving, as
the resulting contract c'' is caller-compatible with respect to the original contract c and
to the refining contract c'.

---

[4]As a special case, a refinement chain may only consist of one method implementation, if a method
is only introduced in one feature and not refined by others. However, programmers and verification
tools still need to consider all feature modules to find out whether this is the case.

Advantages

The main advantage of cumulative contract refinement compared to all previously discussed mechanisms is that it facilitates contract refinement *and* modular reasoning. Compared to plain contracting, it enables contract refinement by combining original contracts with new preconditions and postconditions. In contrast to contract overriding, explicit contract refinement, and conjunctive contract refinement, it enables modular reasoning for callers as discussed for caller-preserving mechanisms (cf. Section 4.1.2).

In addition, cumulative implicit refinement is beneficial for callers. A caller must fulfill any of the preconditions of selected features and can rely on all postconditions of selected features. If there are several preconditions available, the caller only needs to fulfill one of these.

Disadvantages

The beneficial characteristic for callers is, of course, a downside for callees. Using cumulative contract refinement, we can easily create contracts that are hard to fulfill for callees, or there might not even be a single implementation as the contract refinement is contradictory. In the end, this restrictiveness with respect to the addition of preconditions and postconditions may lead to the fact that many interesting properties of a given product-line implementation cannot be specified, which we empirically investigate in Section 4.5. The examples given in Listing 4.2 on Page 58 and Listing 4.4 on Page 63 cannot be expressed by means of cumulative contract refinement.

4.2.6 Consecutive Contract Refinement

In the third mechanism for implicit contract refinement, to which we refer to as *consecutive contract refinement*, we apply specification inheritance [Dhara and Leavens, 1996] known from object orientation to product lines [Thüm et al., 2012]. As discussed in Section 2.1.2, specification inheritance is an enhancement compared to subcontracting [Dhara and Leavens, 1996], and we aim to transfer this enhancement to product lines by the following definition. We define consecutive contract refinement as $\{\phi'\}m'\{\psi'\} \bullet_{ConsCR} \{\phi\}m\{\psi\} = \{\phi' \vee \phi\}m' \bullet m\{(old(\phi') \Rightarrow \psi') \wedge (old(\phi) \Rightarrow \psi)\}$, in which *old* evaluates a predicate in a postcondition as it would be evaluated before method execution (cf. Section 2.1.2).

Example 4.10. *We give an example for consecutive contract refinement based on product line* GPL-scratch *in Listing 4.5. The original method* sortEdges *takes a list of edges as input and returns a sorted list. The feature module* UniqueEdges *extends the method by additionally supporting a set of edges as input, which cannot contain duplicate values. Thus, when calling method* sortEdges *with a set as input, the result is strictly sorted (i.e., is sorted and does not contain duplicates). Given that feature* UniqueEdges *is selected, the caller has the choice which precondition to fulfill (i.e., passing a list or a set) and can rely on the respective postcondition. This example could not have been specified by means of cumulative contract refinement, because it is impossible for the method to fulfill both postconditions if only the precondition of feature* Base *is established.*

Properties

Consecutive contract refinement shares the properties with cumulative contract refinement – both mechanisms are commutative, associative, and idempotent. Commutativity is caused by the commutativity of conjunction and disjunction. Associativity is given due to the following equivalences:

$$
\{\phi''\}m''\{\psi''\} \bullet_{ConsCR} \left(\{\phi'\}m'\{\psi'\} \bullet_{ConsCR} \{\phi\}m\{\psi\}\right)
$$
$$
\equiv \{\phi''\}m''\{\psi''\} \bullet_{ConsCR} \left(\{\phi' \vee \phi\}m' \bullet m\{(old(\phi') \Rightarrow \psi') \wedge (old(\phi) \Rightarrow \psi)\}\right)
$$
$$
\equiv \{\phi'' \vee \phi' \vee \phi\}m'' \bullet m' \bullet m
$$
$$
\quad \{(old(\phi'') \Rightarrow \psi'') \wedge \left(old(\phi' \vee \phi) \Rightarrow (old(\phi') \Rightarrow \psi') \wedge (old(\phi) \Rightarrow \psi)\right)\}
$$
$$
\equiv \{\phi'' \vee \phi' \vee \phi\}m'' \bullet m' \bullet m
$$
$$
\quad \{(old(\phi'') \Rightarrow \psi'') \wedge (old(\phi' \vee \phi) \Rightarrow (old(\phi') \Rightarrow \psi')) \wedge \left(old(\phi' \vee \phi) \Rightarrow (old(\phi) \Rightarrow \psi)\right)\}
$$
$$
\equiv \{\phi'' \vee \phi' \vee \phi\}m'' \bullet m' \bullet m\{(old(\phi'') \Rightarrow \psi'') \wedge (old(\phi') \Rightarrow \psi') \wedge (old(\phi) \Rightarrow \psi)\}
$$
$$
\equiv \{\phi'' \vee \phi' \vee \phi\}m'' \bullet m' \bullet m
$$
$$
\quad \{(old(\phi'' \vee \phi') \Rightarrow (old(\phi'') \Rightarrow \psi'')) \wedge \left(old(\phi'' \vee \phi') \Rightarrow (old(\phi') \Rightarrow \psi')\right) \wedge (old(\phi) \Rightarrow \psi)\}
$$
$$
\equiv \{\phi'' \vee \phi' \vee \phi\}m'' \bullet m' \bullet m
$$
$$
\quad \{\left(old(\phi'' \vee \phi') \Rightarrow (old(\phi'') \Rightarrow \psi'') \wedge (old(\phi') \Rightarrow \psi')\right) \wedge (old(\phi) \Rightarrow \psi)\}
$$
$$
\equiv \left(\{\phi'' \vee \phi'\}m' \bullet m\{(old(\phi'') \Rightarrow \psi'') \wedge (old(\phi') \Rightarrow \psi')\}\right) \bullet_{ConsCR} \{\phi\}m\{\psi\}
$$
$$
\equiv \left(\{\phi''\}m''\{\psi''\} \bullet_{ConsCR} \{\phi'\}m'\{\psi'\}\right) \bullet_{ConsCR} \{\phi\}m\{\psi\}
$$

In particular, $old(\phi' \vee \phi) \Rightarrow (old(\phi') \Rightarrow \psi')$ is equivalent to $old(\phi') \Rightarrow \psi'$, as ψ' only needs to be fulfilled if $old(\phi')$ is fulfilled and $old(\phi') \models old(\phi' \vee \phi)$. Analogously, $old(\phi' \vee \phi) \Rightarrow (old(\phi) \Rightarrow \psi)$ is equivalent to $old(\phi) \Rightarrow \psi$. Furthermore, the commutativity of consecutive contract refinement is due to the commutativity of conjunction and disjunction, and idempotence is given because of the following equivalences:

$$
\{\phi\}m\{\psi\} \bullet_{ConsCR} \{\phi\}m\{\psi\}
$$
$$
\equiv \{\phi \vee \phi\}m \bullet m\{(old(\phi) \Rightarrow \psi) \wedge (old(\phi) \Rightarrow \psi)\}
$$
$$
\equiv \{\phi\}m \bullet m\{old(\phi) \Rightarrow \psi\}
$$
$$
\equiv \{\phi\}m \bullet m\{\psi\}
$$

Furthermore consecutive contract refinement is caller-preserving. Due to commutativity, we only show that the result of composition is caller-compatible with respect to the original contract. First, the preconditions align with the property (i.e., $\phi \models \phi' \vee \phi$). Second, for the postconditions we get that $old(\phi) \wedge (old(\phi') \Rightarrow \psi') \wedge (old(\phi) \Rightarrow \psi) \models \psi$, which is equivalent to $\psi \models \psi$, and thus fulfilled. Caller-compatibility with the refining contract is given due to commutativity.

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

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

Listing 4.5: *Consecutive contract refinement* in product line *GPL-scratch*: features *UniqueEdges* refines a contract by adding a new pair of precondition and postcondition to the original contract defined in feature *Base* (adapted from [Weigelt, 2013]; differences of contracts highlighted for convenience).

Advantages and Disadvantages

The advantages and disadvantages of consecutive contract refinement are the same as for cumulative contract refinement, except that callers cannot only provide one of the preconditions and rely on all postconditions. Consequently, it is not the burden of callees to fulfill all postconditions for any given precondition. Hence, we expect that consecutive contract refinement is superior to cumulative contract refinement, which we evaluate empirically in Section 4.5. In particular, we investigate empirically whether enforcing specification inheritance for feature-oriented method refinements is always feasible.

4.2.7 Comparison of Contract-Composition Mechanisms

So far, we discussed six mechanisms for contract composition that all have their advantages and disadvantages. We summarize the properties of these mechanisms in Table 4.2 and make several observations. All mechanisms are associative and, except for explicit contract refinement, also idempotent. Only conjunctive, cumulative, and

Contract-composition mechanism*	Preservation property	Associativity	Idempotence	Commutativity
$c' \bullet_{PC} c = \{\phi\}m' \bullet m\{\psi\}$	original-preserving	yes	yes	no
$c' \bullet_{CO} c = \{\phi'\}m' \bullet m\{\psi'\}$	refinement-preserving	yes	yes	no
$c' \bullet_{ECR} c = \{\phi' \bullet \phi\}m' \bullet m\{\psi' \bullet \psi\}$	none	yes	no	no
$c' \bullet_{ConjCR} c = \{\phi' \wedge \phi\}m' \bullet m\{\psi' \wedge \psi\}$	none	yes	yes	yes
$c' \bullet_{CumCR} c = \{\phi' \vee \phi\}m' \bullet m\{\psi' \wedge \psi\}$	caller-preserving	yes	yes	yes
$c' \bullet_{ConsCR} c = \{\phi' \vee \phi\}m' \bullet m\{\psi''\}$	caller-preserving	yes	yes	yes

$* \, c = \{\phi\}m\{\psi\}$, $c' = \{\phi'\}m'\{\psi'\}$, and $\psi'' = \big(old(\phi') \Rightarrow \psi'\big) \wedge \big(old(\phi) \Rightarrow \psi\big)$

Table 4.2: Overview on contract-composition mechanisms and their properties.

consecutive contract refinement are commutative. With respect to the previously defined preservation properties, plain contracting is original-preserving and contract overriding is refinement-preserving. Explicit contract refinement and conjunctive contract refinement do not fulfill any preservation properties. The remaining two mechanisms, cumulative and consecutive contract refinement, are both caller-preserving.

The discussed mechanisms can be distinguished by the supported contract refinements. Plain contracting completely forbids any contract refinement, whereas conjunctive and cumulative contract refinement enable the refinement in a limited way. Consecutive contract refinement subsumes cumulative contract refinement with respect to the supported refinements. Finally, contract overriding and explicit contract refinement facilitate arbitrary contract refinements and, consequently, also arbitrary method refinements.

A property that is conflicting with support for arbitrary contract refinements is modular reasoning for callers. Modular reasoning is possible with plain contracting as no refinement is available. The only other mechanisms enabling modular reasoning for callers are cumulative and consecutive contract refinement, but they only support contract refinement in a limited way.

Finally, for contract overriding, we discussed that the missing support to implicitly or explicitly reusing contracts of other features may lead to specification clones and derivative contracts. Explicit contract refinement improves over contract overriding by providing a means to explicitly refer to previous preconditions and postconditions and, thus, a means to avoid some specification clones and derivative contracts. In addition, explicit contract refinement relies on similar linguistic means as feature modules, but may lead to an additional source of errors in contracts due to dangling references for keyword original.

Based on our discussions, we cannot yet rule out mechanisms or even favor a single mechanism. Their are some indicators that consecutive contract refinement is superior

to cumulative contract refinement, and that explicit contract refinement is superior to contract overriding. Nevertheless, an empirical evaluation is required to judge about the practical relevance of these six mechanisms, which we postpone to Section 4.5.

4.3 Composition Beyond Pre- and Postconditions

For brevity, our considerations in Section 4.1 and Section 4.2 only focused on contracts consisting of a precondition and a postcondition. However, there are numerous advanced specification concepts based on the notion of contracts [Chalin et al., 2005; Hatcliff et al., 2012; Meyer, 1988]. In this section, we discuss some of the most relevant concepts and how our previous discussions relate to these concepts. In particular, we discuss specification cases, multiple preconditions and postconditions, class invariants, as well as pure methods.

4.3.1 Specification Cases

In the principle of design by contract, every contract consists of exactly one precondition and one postcondition [Meyer, 1988, 1992]. However, specification languages such as JML support the use of multiple *specification cases* for one method, by connecting them with keyword also [Chalin et al., 2005]. Roughly speaking, a method is specified with multiple contracts, which are connected by contract composition as with consecutive contract refinement. In fact, keyword also is just syntactic sugar and can be desugared whenever required, as described elsewhere [Chalin et al., 2005].

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

To avoid desugaring of keyword also, we can extend contract-composition mechanisms with support for specification cases. A trivial extension exists for plain contracting and contract overriding; instead of copying the original and refining precondition and postcondition, respectively, we can simply copy the complete original or refining contract during composition [Benduhn, 2012]. The extension of consecutive contract refinement is straightforward, because the composition is equivalent to specification cases anyway, and we can simply combine contracts from different feature modules by means of keyword also, too. Hence, consecutive contract refinement can be implemented by means of specification cases [Benduhn, 2012].

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

In contrast to these four contract-composition mechanism, it seems that desugaring cannot be avoided for conjunctive contract refinement and cumulative contract refinement. However, specification cases are even an elegant way for implementation of consecutive contract refinement, because it avoids the cloning of preconditions into postconditions during composition (cf. Section 4.2.6).

4.3.2   Multiple Preconditions and Postconditions

Besides the definition of multiple specification cases each consisting of a precondition and a postcondition, we may also define multiple preconditions and postconditions for a single specification case. Similar to specification cases, multiple preconditions and postconditions are just syntactic sugar and can be replaced as follows: Given the preconditions $\phi_1, \phi_2, \ldots, \phi_n$, we can combine them into one precondition by means of a conjunction (i.e., $\bigwedge_{1 \leq i \leq n} \phi_i$), and the same applies to postconditions [Beckert et al., 2007; Benduhn, 2012].

As for specification cases, we may apply desugaring with respect to multiple preconditions and postconditions before composition to reuse contract-composition mechanisms as presented above. However, desugaring is, again, not required for plain contracting and contract overriding, as they can simply copy the complete contract. When implementing consecutive contract refinement by means of specification cases (as discussed in Section 4.3.1), we can simply copy complete cases and, thus, desugaring is not required, too. For conjunctive contract refinement, we only compose preconditions and postconditions by means of conjunction, which can be implemented with multiple preconditions and postconditions. For cumulative contract refinement, the same is true for postconditions, but preconditions need to be desugared as described above prior to composition. Finally, explicit contract refinement poses similar challenges as for specification cases, because keyword original can either refer to the desugared precondition or postcondition, or refer to the jth precondition or postcondition of the ith specification case. For simplicity, we assume the former option.

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

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

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

Listing 4.6: *Pure-method refinement* in product line *ExamDB*: the contract of method
consistent contains a call to the pure method validStudent. Feature module *BackOut* re-
fines the contract of method consistent indirectly by refining method validStudent [Thüm
et al., 2012].

4.3.3   Pure Methods and Model Methods

As introduced in Section 2.1.1, contracts may contain calls to pure methods (i.e., meth-
ods that terminate and are side-effect free) [Beckert et al., 2007]. Pure methods require
attention during composition, as their refinement implicitly refines all those contracts
calling this method. That is, by refining one pure method, we can refine several con-
tracts indirectly at the same time. In the following, we refer to the refinement of pure
methods as *pure-method refinement*. Interestingly, pure-method refinement enables con-
tract composition by means of traditional feature-oriented method composition and does
not require new language concepts nor new mechanisms for contract composition.

Example 4.11. *In Listing 4.6, we give an example of pure-method refinement in a
feature-oriented database implementation for student exams. Class* ExamDatabase *stores
the results of student exams in array* students*, whereas a* null-*value refers to a free po-
sition in the array. The method* consistent *checks whether all students have at least
zero points. The method* validStudent *is used in the contract of method* consistent *and
is refined by a class refinement of feature module* BackOut*; this refinement allows stu-*

dents to back out from an exam. Hence, the contract of method consistent *is refined by changing the body of method* validStudent. *While our example just shows one method contract with one pure-method call, in principle, a pure method could be called from several contracts and a contract may contain calls to several pure methods.*

Similarly to the refinement of pure methods, we may refine model methods. A *model method* is a method introduced only for specification purposes and it can only be called within the specification and is invisible for the implementation [Chalin et al., 2005; Hatcliff et al., 2012]. With refining a model method, we can indirectly refine all contracts calling this model method as with pure methods. The advantage of refining model methods instead of pure methods is that we do not need to encode specifications into the underlying programming language. In contrast, the advantage of pure methods over model methods is that we can use the same method refinement in specification *and* implementation. Consequently, it depends on the situation whether pure methods or model methods should be refined. However, for our following considerations, it is sufficient to exemplarily focus on pure-method refinement.

All mechanisms discussed in Section 4.2 may or may not be enriched with pure-method refinement, resulting in twelve mechanisms overall. However, enabling pure-method refinement comes with the disadvantage that it breaks almost all properties. Compared to the properties without pure-method refinement (cf. Table 4.2 on Page 68), we loose all preservation properties, idempotence, and commutativity. The reason is that feature-oriented method refinement with keyword original is neither idempotent nor commutative [Apel et al., 2010b], and that contracts containing method calls may be arbitrarily changed by means of refining those pure methods. Consequently, it is a design decision whether or not to allow the refinement of pure methods. If the refinement of pure-methods is forbidden, tools could check for possible violations.

4.3.4   Class Invariants

In design by contract, besides assigning contracts to methods, we may also define conditions for classes by means of class invariants. Depending on the visibility (i.e., public in JML) of a given class invariant, it applies to all methods with the same or higher visibility [Leavens and Müller, 2007]. In particular, a class invariant is equivalent to adding the same condition to all preconditions and postconditions of methods that it applies to. Indeed, class invariants are sometimes just syntactic sugar. However, an exception is given in inheritance hierarchies in which not all subclasses are known (i.e., object-oriented, open systems). For example, an invariant defined for the class AbstractSet in the Java platform cannot be expressed by means of preconditions and postconditions, because we simply do not know all subclasses to which this invariant applies to. Nevertheless, for our considerations about the refinement of class invariants it is sufficient to consider them as syntactic sugar.

For each of our six contract-composition mechanisms, we derive a strategy for dealing with the refinement of class invariants. As preconditions and postconditions cannot

be refined at all in plain contracting, when desugaring class invariants, we cannot allow any refinement of invariants either. In contrast, we should be able to arbitrarily refine class invariants in contract overriding or explicit contract refinement, as preconditions and postconditions can be arbitrarily refined, too. Similar to preconditions and postconditions in explicit contract refinement, we could introduce keyword **original** in invariants to reuse the invariant that is subject to refinement. For all remaining mechanisms, invariants may also be changed arbitrarily. However, for conjunctive contract refinement and cumulative contract refinement, refining a class invariant from i to i' requires that the postcondition $i' \wedge i$ is satisfiable. For example, we cannot refine an invariant by its negation. Otherwise, the postcondition cannot be fulfilled by any method implementation. In summary, we could allow the refinement of class invariants except for plain contracting, whereas conjunctive contract refinement and cumulative contract refinement restrict possible refinements.

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

However, a restrictive refinement of class invariants does not require a unique identification. That is, if we aim to refine an existing invariant by adding further terms in a conjunction, we can simply introduce a new invariant instead. The reason is that multiple invariants in a single class are syntactic sugar for a single invariant being the conjunction of all of them. Hence, simply by adding new invariants in feature modules, we can implicitly refine invariants in this restrictive way, which is similar to conjunctive contract refinement. By introducing invariants in feature modules, we achieve variable class invariants customizable by the feature selection. In particular, an invariant defined in a certain feature module does only need to be established if the feature module is selected. We refer again to Listing 4.1 on Page 56 and Listing 4.4 on Page 63, in which class invariants are introduced in core features as well as optional features.

4.4 Tool Support for Specifying Feature-Oriented Contracts

In the former part of this chapter, we discussed numerous options for feature-oriented contract composition. This includes six mechanisms for contract composition, as well as design decisions how to deal with pure-method calls and class invariants. We aim to answer remaining questions by means of an empirical investigation of product-line

specifications in Section 4.5. However, a comprehensive empirical investigation requires
tool support for specifying feature-oriented contracts and for their composition. Next,
we describe the tool support that has been developed in the course of this thesis. In
Section 4.4.1, we discuss our extension of a tool for feature-oriented composition, namely
FEATUREHOUSE, with support for JML. Based on our FEATUREHOUSE extension, we
implemented tool support in Eclipse by extending FEATUREIDE, which we present in
Section 4.4.2.

4.4.1 Automating Contract Composition with FeatureHouse

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

As our goal is to build tool support for feature-oriented contracts based on *existing
tools*, we aim to extend an existing language for method contracts. We have chosen
to extend JML, because many verification tools are available for JML [Burdy et al.,
2005]. Furthermore, instead of building a new feature-module composer from scratch,
we rather extend an existing one; available composers are FEATUREC++ [Apel et al.,
2005], AHEAD [Batory, 2006], and FEATUREHOUSE [Apel et al., 2013b]. While FEA-
TUREC++ only supports the composition of feature modules written in a feature-
oriented extension of C++, AHEAD and FEATUREHOUSE support a variety of pro-
gramming languages. AHEAD supports Java 1.4, XML, and JavaCC, whereas FEA-
TUREHOUSE supports Java 1.5, C#, C, Haskell, JavaCC, Alloy, and UML. We decided
to extend FEATUREHOUSE as it already supports Java 1.5 and a new language can be
integrated by providing a new grammar annotated with information about composition
rules [Apel et al., 2013b].

We have developed a JML grammar according to the JML language levels 0-3 [Leavens
et al., 2013], based on the existing FEATUREHOUSE grammar for Java 1.5. The six
contract-composition mechanisms discussed in Section 4.2 are implemented by means
of a new command-line parameter of FEATUREHOUSE. That is, a user can choose *one
mechanism for a product line* and then compose feature modules of selected features
automatically. The main part of this tool support was developed in the course of a
bachelor's thesis [Benduhn, 2012].

However, in our empirical evaluation, we found that several mechanisms could even
be combined when specifying a single product line. Otherwise, a single method of a
product line may rule out certain composition mechanisms and certain preservation
properties. By means of keywords, we can choose a *contract-composition mechanism
for each method* individually.

contract c4   contract c3   contract c2   contract c1

$$\{\phi'''\}m\{\psi'''\} \bullet \{\phi''\}m\{\psi''\} \bullet \backslash consecutive\_contract\{\phi'\}m\{\psi'\} \bullet \{\phi\}m\{\psi\}$$

Consecutive Contract Refinement   Explicit Contract Refinement
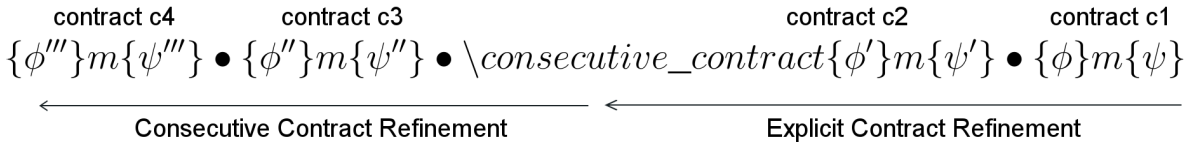
Figure 4.3: Overriding a contract-composition mechanism with another mechanism [Weigelt, 2013].

Before presenting keywords specifying contract composition, we explain some design decisions. First, we should consider *one mechanism as default* to save effort when specifying keywords. We argue that this should be explicit contract refinement or contract overriding, as they do not pose any restrictions on the possible method refinements in feature-oriented programming. In particular, it would be counter-intuitive if the default in a specification technique in feature-oriented programming is restrictive, while feature-oriented programming is not. Second, we argue that there is no need for separate keywords for contract overriding and explicit contract refinement. The difference of both is that explicit contract refinement allows for the usage of keyword original, but there is no reason to forbid this keyword. Hence, explicit contract refinement subsumes contract overriding as the developer may decide to not use keyword original at his choice.

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

We propose to define contract-composition keywords at the beginning of contracts. However, the ability of feature-oriented programming for method refinement naturally raises the question whether it should be possible to *override contract-composition keywords*. With overriding, we could use different contract-composition mechanisms for different refinements of the same method. We illustrate the overriding of keywords in Figure 4.3. Contract c_1 and c_2 are composed using explicit contract refinement as no keyword is specified in contract c_1. Keyword consecutive_contract defined in contract c_2 overrides the contract composition with consecutive contract refinement for later refinements (e.g., when applying the contract refinements c_3 and c_4) [Weigelt, 2013].

The overriding of contract-composition keywords provides more flexibility for refinement, but may *break preservation properties*. For instance, if we decide to use consecutive contract refinement for a given method and a subsequent method refinement changes the composition mechanism to explicit contract refinement. In this case, we lose the modular reasoning for method callers, as later feature modules may completely
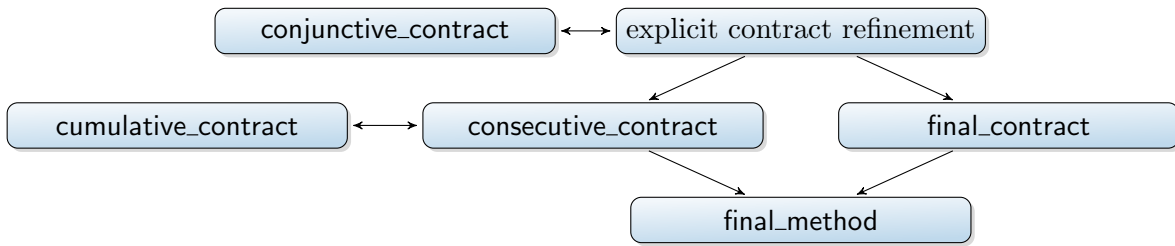
Figure 4.4: Contract-composition keywords and possible overriding that establishes preservation properties.

replace existing preconditions and postconditions. To avoid that keyword overriding can break preservation properties, we only permit overriding if guarantees are preserved.

In Figure 4.4, we illustrate in which cases overriding is permitted, whereas transitive arrows are omitted. First of all, keyword final_method can be used after every other mechanism, as it only indicates the end of the refinement chain and does not violate any properties. In contrast, keyword final_method cannot be followed by any other mechanism, as later method refinements are prohibited. Next, we can override keywords with mechanisms that have identical preservation properties (cf. Table 4.2 on Page 68). In particular, we can override cumulative contract refinement with consecutive contract refinement, and vice versa. The same holds for conjunctive contract refinement and explicit contract refinement. Finally, as explicit contract refinement and conjunctive contract refinement do not establish any of the above defined preservation properties, we can override them by all other mechanisms.

We refer to the contract composition with the aforementioned keywords and possible overridings as *keyword-based contract composition*. With our extension of FEATURE-HOUSE it is possible to use any of the contract-composition mechanisms discussed in Section 4.2 or keyword-based contract composition based on a command-line parameter.

4.4.2 Supporting Feature-Oriented Contracts in FeatureIDE

Prior to this thesis, the command-line tool FEATUREHOUSE has been integrated into ECLIPSE in the FEATUREIDE project [Kästner et al., 2009c; Thüm et al., 2014b], together with several other product-line implementation tools. FEATUREIDE is a framework for product-line implementation providing editors with syntax highlighting and content assistants as well as views specific to feature-oriented programming supporting the whole development process [Thüm et al., 2014b]. We integrated our FEATURE-HOUSE extension into FEATUREIDE (a) to provide a convenient use of contracts for student in lectures on product lines, (b) to ease the transfer of research results to practice, and (c) to implement error reporting for wrong usage of contracts based on the FEATUREIDE infrastructure. Our extensions are open-source and available as part of the FEATUREHOUSE and FEATUREIDE projects.[5]

---

[5]http://featureide.cs.ovgu.de

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

4.5 Empirical Evaluation of Feature-Oriented Contracts

The above sections raise several questions that we aim to answer by means of an empirical evaluation. In particular, to what extent do feature modules require variability in contracts and contract refinement? Are refinements of feature-oriented contracts typically original-preserving, refinement-preserving, caller-preserving, or callee-preserving? Which of the six contract-composition mechanism discussed in Section 4.2 is superior to others in practice? Do specification clones and derivatives contracts occur frequently with contract overriding? Is cumulative contract refinement too restrictive in practice? Does keyword-based contract composition give benefits compared to the six mechanisms in typical product lines? We give on overview on the subject product lines in Section 4.5.1. In Section 4.5.2, we share the insights gained with our case studies. Finally, we discuss possible threats to validity in Section 4.5.3.

4.5.1 Case Studies

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

The rationale behind *developing product lines from scratch* is that a given language for contracts may impose restrictions on the design. However, when developing a feature-oriented program from scratch, we can try to come up with a design that fits the language. In the end, we know whether a certain contract-composition mechanism is

feasible when building modules and their contracts from scratch. Overall, we have implemented and specified five feature-oriented product lines, which consist of data structures (*IntegerList*), algorithms (*UnionFind*, *StringMatcher*), and both (*BankAccount*, *GPL-scratch*).

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

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

In total, we conducted 14 case studies with a different product line each. According to our above mentioned tool support, all studied product lines are implemented in feature modules based on Java and feature-oriented contracts based on JML, but we expect similar results for other object-oriented languages and contract-based specification languages. We refer to Appendix A for a short description of each product line and statistics such as the number of features and classes. For our evaluation, it was not necessary to verify or test whether all or some products adhere to their specification. In the most case studies, we just created feature-oriented contracts as documentation. Nevertheless, how some of these product lines have been verified is explained in Chapter 5. All 14 subject product lines are publicly available at SPL2go.[6]

4.5.2   Results and Insights

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

---

[6]http://spl2go.cs.ovgu.de/
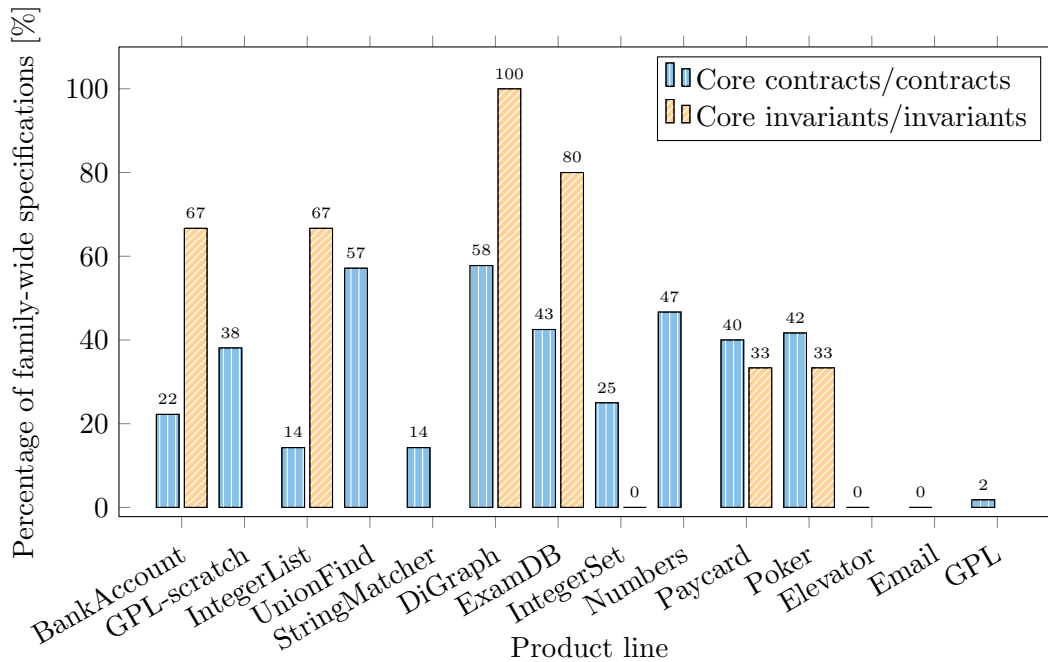
Figure 4.5: Percentage of family-wide specifications compared to all specifications.

Variability in Contracts

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

In Figure 4.5, we illustrate the percentage of core contracts and core invariants compared to all contracts and invariants, respectively. Missing numbers for some product lines indicate that invariants were not existent and thus the percentage is undefined (due to division by zero). Every product line contains contracts that are *not* defined in core features. Except for product line *DiGraph*, every product line with invariants also contains invariants not defined in core features. That is, the generated specification is typically similar between products, but not identical. In particular, for the product lines *Elevator* and *Email* there are products that do not share any specifications, because they do neither contain core contracts nor core invariants. The average over all product lines is that 29 % are core contracts and 54 % are core invariants, but we expect even smaller percentages for larger product lines.[7]

---

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

In Section 3.6, we identified as an open research question *whether family-wide specifica-tions are sufficient* for product lines. Our case studies indicate that family-wide speci-fications are not feasible, at least when specifying product lines by means of contracts. Hence, it seems that the code-level specification of variable code requires variable speci-fications, which are supported by all contract-composition mechanism discussed above. Even with plain contracting we can introduce contracts in optional feature modules to create variation in specifications.

Nevertheless, we may define a family-wide specification by means of feature-oriented contracts. That is, we can decompose a family-wide specification into several feature modules belonging to core features. Indeed, such a decomposition has been done in product line *Poker*. In Figure 4.6, we show an excerpt of the collaboration diagram, in which product-line specifications common to all products are split up to three core features. Reasons for such a decomposition are manifold. For example, the decomposi-tion separates concerns that different developers need to consider during evolution. In this sense, feature-oriented contracts are more general than defining contracts that all products must establish.

In Section 3.6, we raised a similar research question, namely *whether family-based spec-ifications are needed* or whether feature-based specifications are sufficient. That is, can we define the intended behavior of a product line by specifying each feature? Or do we need to explicitly define the behavior of feature combinations? The research question is similar to the question whether product lines can be implemented by means of feature modules. Liu et al. [2006] experienced that implementing a feature module for each feature has not been sufficient when they decomposed a legacy system into a product line, but they can easily be enriched by additional derivative modules (a.k.a. lifters [Pre-hofer, 1997]). A derivative module is a feature module that is added whenever two or more specific features are chosen.

In our case studies, we made similar experiences when decomposing contracts into feature-oriented contracts. For almost all programs that we migrated to a product line, we were able to specify all contract refinements by means of contract-composition mechanisms as discussed in Section 4.2 – without a need for derivative modules. As an exception, massive use of derivative modules turned out to be necessary for product line *ExamDB*. In particular, we had to create all theoretically possible derivative modules. For one core feature and three independent-optional features, we had to create three second-order derivatives and one third-order derivative (cf. Figure 4.7). The reason and possible solutions are discussed below. For creation strategies beyond migration (i.e., development from scratch and specification of existing product lines), no derivatives were needed except for *GPL*. However, *GPL* already contained ten derivatives for im-plementation purposes and specifying the product line with feature-oriented contracts did not require further derivatives. Overall, our experience is that feature-based specifi-cation is sufficient, because derivative modules could be completely avoided in our case studies by choosing a feasible contract-composition mechanism for each product line.
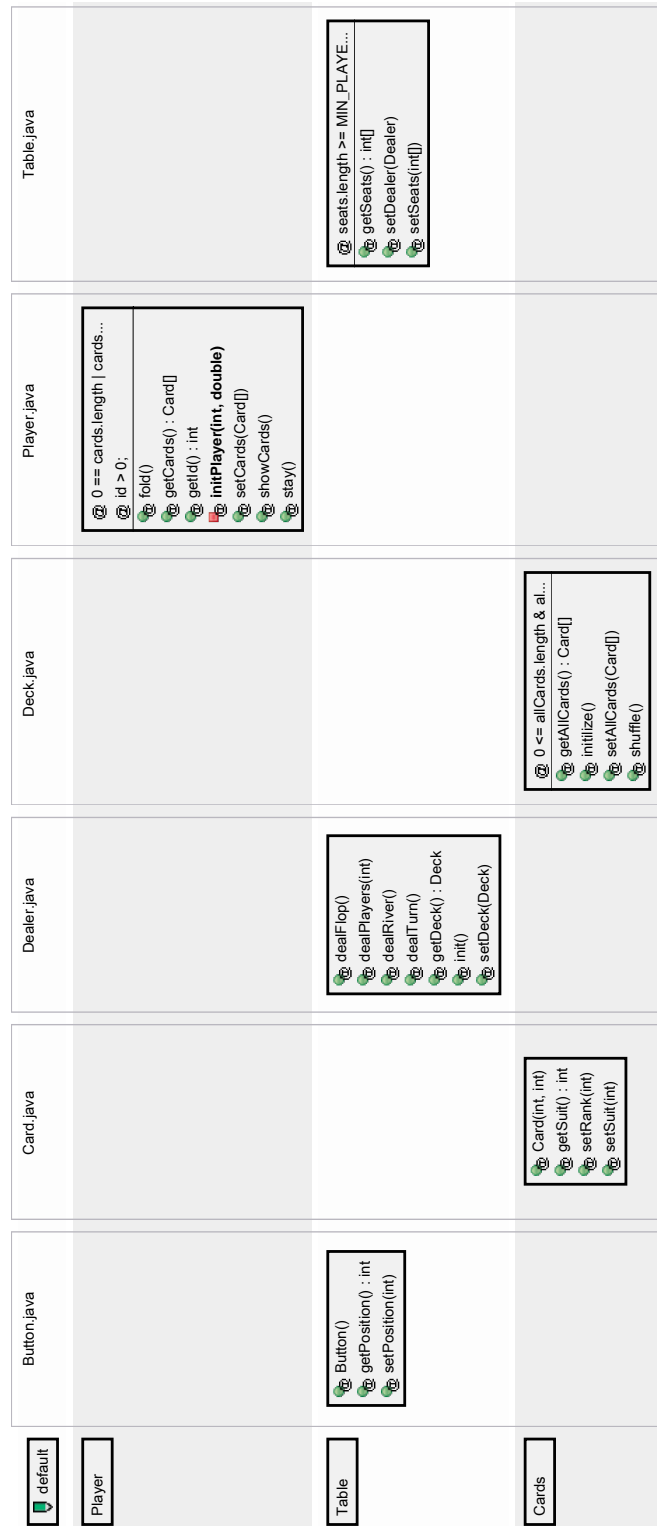
Figure 4.6: Collaboration diagram showing all core contracts and core invariants of product line *Poker*.
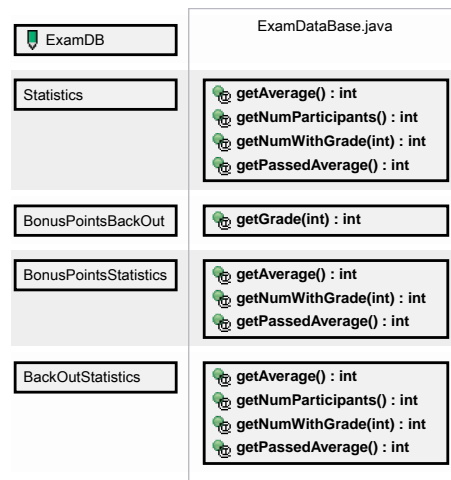
Figure 4.7: The derivative modules of product line *ExamDB* cover all combinations of the optional features *BonusPoints*, *BackOut*, and *Statistics*.

The Need for Contract Refinement

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

In Figure 4.8, we illustrate the percentage of contract refinements with respect to the overall number of contracts. We discover that between 0 % and 86 % of all contracts refine another contract in at least one product. No contract refinements were necessary only for *DiGraph* and *IntegerSet*. Product line *DiGraph* does not contain a single method refinement that could have required contract refinement. In other words, the features chosen for decomposition do not cross-cut method implementations. Nevertheless, method and contract refinement could be necessary when extracting further features or extending *DiGraph* with new features. The product line *IntegerSet* contains several method refinements that all adhere to the initially introduced contract. We already gave an example for a method refinement that does not require a contract refinement in Listing 4.1 on Page 56. Besides these two exceptions, all other product lines contain contract refinements. Hence, plain contracting is only applicable to all contracts in product lines *DiGraph* and *IntegerSet*.

In our subject product lines, several methods do have different contracts in different products. However, contract refinement is not the only option to achieve different contracts for the very same method; we experienced alternative introductions of contracts in three product lines, namely *GPL-scratch*, *IntegerSet*, and *GPL*. Interestingly, the three product lines have been created by means of a different strategy each (i.e., product line developed from scratch, specified program migrated to a product line, and
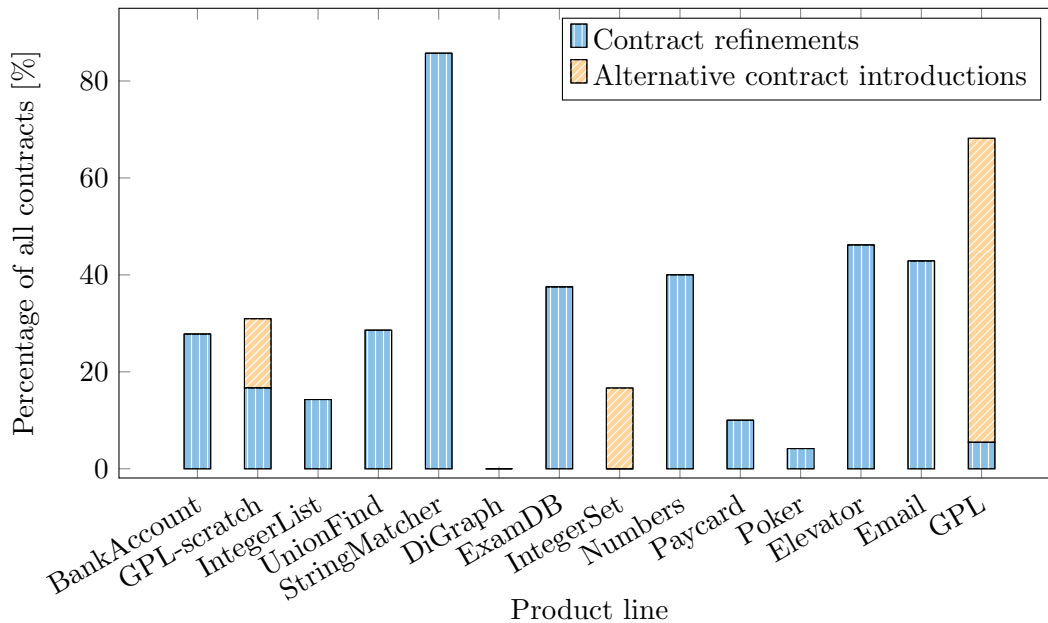
Figure 4.8: Percentage of contract refinements and alternative contract introductions compared to all contracts.

specification of an existing product line). Hence, alternative contract introductions are not only caused by an existing program or product-line design. Based on Figure 4.8, we make two observations. First, in product line *IntegerSet* all differing contracts are due to alternative contract introductions (i.e., there are no contract refinements). Second, in product line *GPL* more than half of the contracts are introduced in alternative features. Consequently, each contract-composition mechanism should also come with a strategy to deal with alternative contract introductions. Disallowing alternative contract introductions completely does not seem to be a valid option according to our experience with the case studies.

We already discussed that not all but some method refinements require contract refinements. That is, even though the implementation of a method is refined, the resulting method still adheres to the original contract (cf. plain contracting in Section 4.2.1). In Figure 4.9, we illustrate the percentage of method refinements that require the refinement of contracts. Similarly, we illustrate the percentage of alternative method introductions that require alternative contract introductions. Missing numbers indicate that the product line does not contain method refinements or alternative method introductions (i.e., the percentage is undefined). Both percentages range from 0 % to 100 % over all product lines. On the one hand, as indicated above, product line *IntegerSet* has method refinements, but no contract refinements. On the other hand, all method refinements of product lines *BankAccount*, *GPL-scratch*, and *StringMatcher* require contract refinements. Similarly, product line *Poker* has alternative method introductions but no alternative contract introductions, whereas in product line *GPL-scratch* every alternative method introductions is specified by means of an alternative contract. In average over all product lines, 47 % of the method refinements require contract refinements
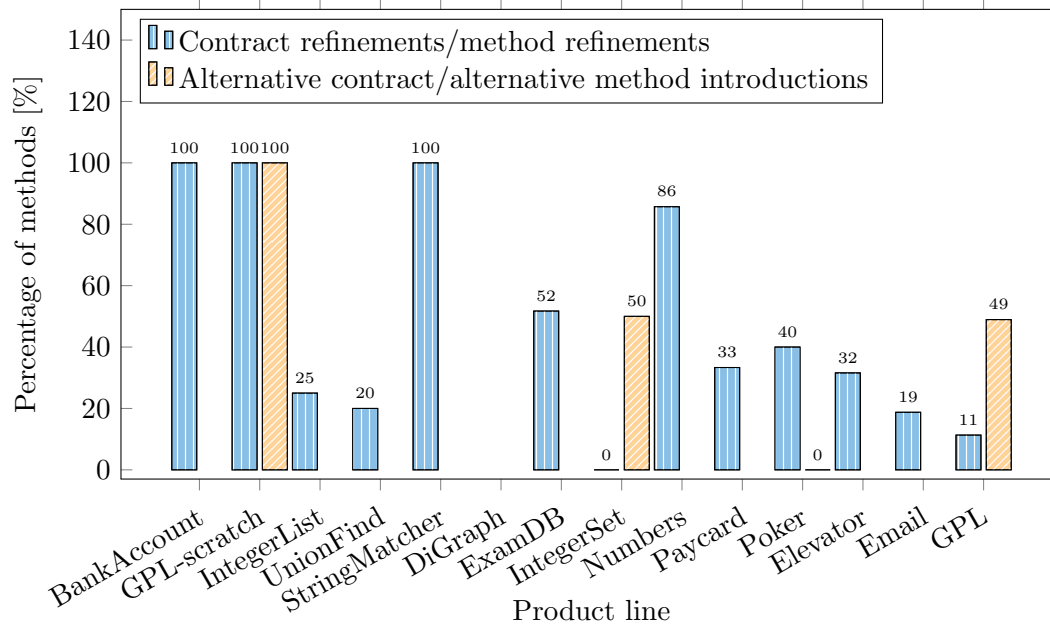
Figure 4.9: Percentage of contract refinements and alternative contract introductions compared to method refinements and alternative method introductions.

(i.e., plain contracting is sufficient for 53 % of the method refinements), and 50 % of the alternative method introductions require alternative contracts.

In contrast to method contracts, alternative introductions and refinements of class invariants were not needed in our case studies. However, there was an interesting case when decomposing *ExamDB* into a product line, which is shown in Listing 4.7. The original source code contained a large invariant that we had to decompose to the feature modules *ExamDB* and *BonusPoints*. The reason for decomposition was that field bonusPoints was moved to feature module *BonusPoints*, and thus the reference to the field within the invariant had to be moved accordingly. Otherwise, the absence of feature *BonusPoints* would have resulted in a dangling method reference in JML. In this case, fortunately, we could decompose the original invariant into two invariants. Hence, the refinement of invariants was not necessary. Nevertheless, other product lines may actually require invariant refinements.

Contract Preservation

In Section 4.1.1, we discussed several preservation properties that contract composition may establish. Hence, a question is to which percentage the contract refinements in our subject product lines preserve which properties. In the first diagram of Figure 4.10, we illustrate percentage of contract refinements for each preservation property.[8] That some bars sum up to more than 100 % is due to the fact that the same contract

---

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

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

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

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

Listing 4.7: Decomposition of a class invariant for product line *ExamDB*.

refinement may establish several preservation properties. In average, most contract refinements are original-caller-preserving (71 %) or refinement-caller-preserving (65 %). Only a small portion of all contract refinements are refinement-callee-preserving (17 %), original-callee-preserving (16 %), or do not preserve any of these four properties (16 %). Only three product lines contain contract refinements without preservation properties,
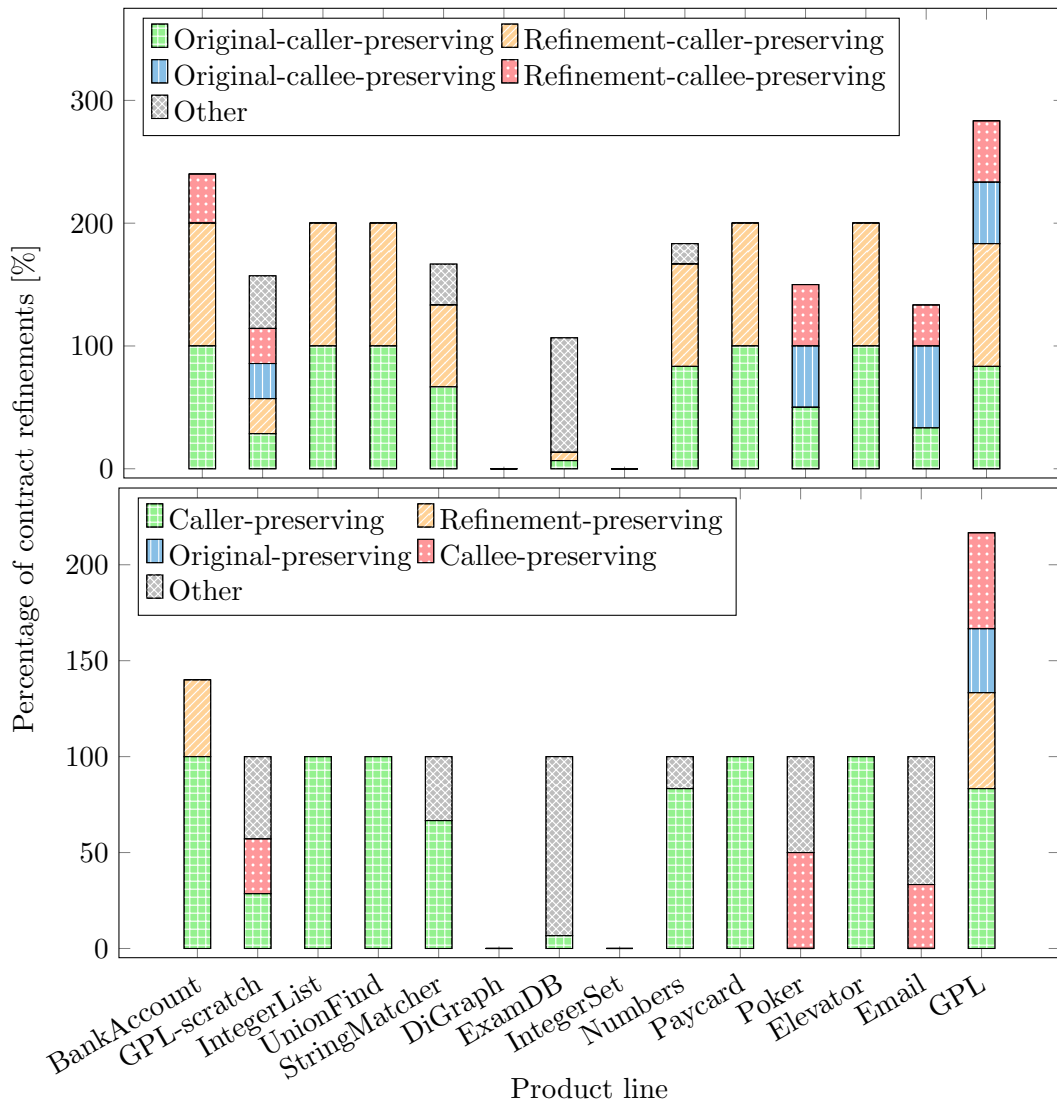
Figure 4.10: Preservation properties of contract refinements in all product lines.

namely *GPL-scratch*, *ExamDB*, and *Numbers*. For an example of such a contract refinement, we refer again to Listing 4.4 on Page 63. In this example, precondition and postcondition are strengthened at the same time using conjunctive contract refinement.

In the second diagram of Figure 4.10, we show the percentage of contract refinements that adhere to the fundamental preservation options discussed in Section 4.1.2. Most contract refinements are caller-preserving (64 %), and thus align with behavioral subtyping. A fourth of all contract refinements do not establish any of the fundamental options (25 %). Callee-preserving (13 %), refinement-preserving (8 %), and original-preserving (3 %) contract refinements are rather rare. Original-preserving contract refinements only occur in product line *GPL* and the reason is that a method with its contract is introduced in an optional feature; even though the contract refinement establishes the same contract, the contract has to be cloned. Otherwise the method would have no con-
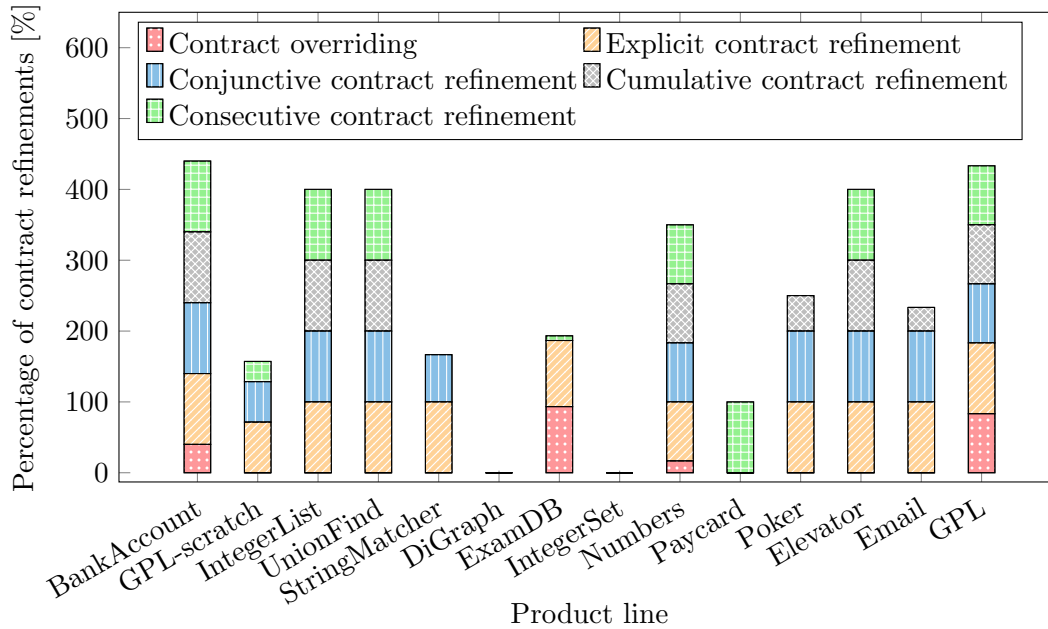
Figure 4.11: Applicability of contract-composition mechanisms.

tract in those products that do not contain the optional contract introduction. Those original-preserving contract refinements could be avoided by changing the design of the product line (i.e., introducing a new feature module for the contract introduction). This insight justifies our strategy to specify an existing product line, because the reason was that an existing product-line design was given, and it could have remained unnoticed with the other two strategies.

Our insight with these statistics is that caller-preserving mechanisms, such as consecutive contract refinement and cumulative contract refinement, are suitable for most contract refinements. For all other contract refinements, explicit contract refinement can be used, which does not establish any preservation properties. One may also use a callee-preserving mechanism for some contract refinements, which, however, was not part of our previous discussion. In Figure 4.11, we show the percentage of contract refinements that can be expressed with each contract-composition mechanism. We counted contract overriding as applicable to contract refinements if explicit contract refinement was applicable and keyword original could not be used. Furthermore, we counted mechanisms only as applicable if there was no other mechanism that could be used and avoided more specification clones.

Most contract refinements can be expressed with explicit contract refinement (87 % on average), followed by conjunctive contract refinement (74 %), consecutive contract refinement (58 %), and cumulative contract refinement (54 %). Only each fifth contract refinement can be expressed with contract overriding (19 %). In particular, explicit contract refinement is applicable in 55 % of the contract refinements by taking advantage of keyword original and 21 % without keyword original (i.e., contract overriding). The experience with our case studies confirms our suspicion that contract overriding is
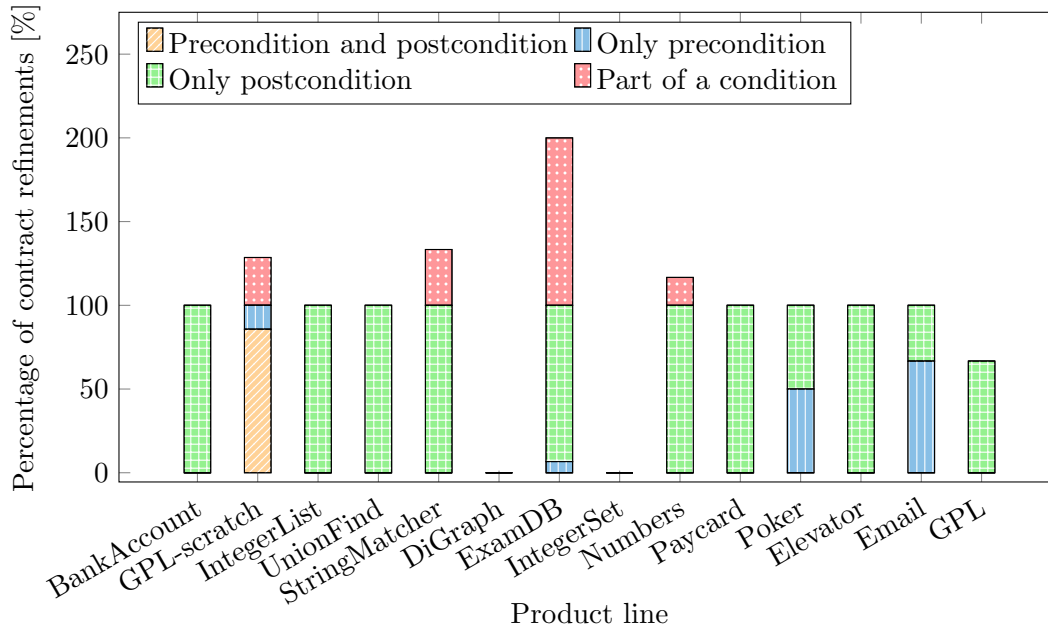
Figure 4.12: The granularity of contract refinements in all product lines.

superseded by explicit contract overriding. Similarly, there was not a single contract refinement to which cumulative contract refinement is applicable, while consecutive contract refinement is not applicable. Hence, our case studies suggest that contract overriding and cumulative contract refinement are not needed. Even if not shown in these diagrams, each case in which a method refinement does not require a contract refinement can be considered as original-preserving (cf. Figure 4.9), and thus plain contracting can be applied.

Granularity of Contract Refinement

A further interesting property of contract refinement is granularity. That is, whether only a precondition is refined, only a postcondition, or even just a part of a condition. In Figure 4.12, we give an overview on the granularity for the contract refinements in our subject product lines.[9] Most contract refinements only changed the postcondition while the precondition remained unchanged (79 % of contract refinements on average over all product lines). In contrast, sole refinement of a precondition was rather rare (11 %). Contract refinements that refine precondition *and* postcondition only occurred in product line *GPL-scratch* and seem to be rather atypical. The reason for the special position of *GPL-scratch* is most probably that it was designed to show the power of contract-composition mechanisms (cf. Appendix A).

That simultaneous refinement of precondition and postcondition does not occur in most product lines is rather surprising and does have consequences that we have not antic-

---

[9]As mentioned above, product line *GPL* contains identical contracts in contract refinements due to optional contract introductions, which do not fall into any of the categories of the diagram and are thus omitted.

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

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

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

Listing 4.8: Consecutive contract refinement in product line *Email* requires cloning of preconditions.

ipated in our previous discussion: First, when refining only the postcondition conjunctive contract refinement, cumulative contract refinement, and consecutive contract refinement collapse to the same semantics (i.e., postconditions are connected in a conjunction). Second, when only refining a precondition cumulative contract refinement and consecutive contract refinement collapse to the same semantics (i.e., preconditions are connected in a disjunction). Finally, it is relevant how contract-composition mechanisms deal with contract refinements in which either precondition or postcondition is refined. In particular, we found that conjunctive contract refinement can avoid cloning unchanged preconditions compared to cumulative contract refinement or consecutive contract refinement. We give an example of such cloning in Listing 4.8. The precondition has to be cloned, because non-existent preconditions are treated as requires true and the composition with the original precondition by means of a disjunction would invalidate all existing preconditions. However, it seems more reasonable to treat non-existent preconditions and postconditions in contract refinements of all contract-composition mechanisms as requires original and ensures original instead.

We noticed that specification clones were especially a problem in those product lines, in which only a part of a precondition or postcondition was refined. In particular, these product lines are *GPL-scratch*, *StringMatcher*, *ExamDB*, and *Numbers* (cf. Fig-

ure 4.12). For instance, in product line *ExamDB*, we had to clone and adapt contracts for all contract refinements, because they required fine-granular changes. All discussed mechanisms for contract composition only enable the reuse of complete preconditions and postconditions. Even worse, we had to create many derivative contracts due to the cloning; for four contract refinements in regular feature modules, we had to create 11 additional contract refinements in four derivatives modules. These derivative contracts massively aggravated the cloning. However, we found an elegant solution that avoids all clones and all derivatives based on pure-method refinement. We refer again to Listing 4.6 on Page 71, which is an excerpt of product line *ExamDB* illustrating the solution. Overall, we introduced two pure methods called from several contracts and refined the pure methods instead of refining each contract explicitly. Alternatively, we could have used model methods, but in this case we could even use these pure methods to reduce code clones. Hence, pure methods and model methods seem to be especially suited for fine-grained contract refinements. At the same time, it is possible to reduce accidental contract refinements by using keyword final_method for all pure methods that are not supposed to be refined.

Furthermore, we experienced several specification clones in alternative contract introductions. However, only a small percentage of them actually contain fine-granular differences, which could be avoided by means of pure methods and model methods. The majority of these alternative contract introductions were identical clones. We experienced the relevance of this problem especially in product line *GPL*, in which more than half of the contracts are introduced in alternative features and cloned (cf. Figure 4.8 on Page 83). A solution could be to specify those contracts in another feature module, which is available whenever one of the alternative features is available. In some cases, this could require to create a new feature module only for contracts. Another solution could be to investigate how contracts for alternative features should be specified, which was only rudimentary discussed in the previous sections.

4.5.3 Threats to Validity

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

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

The results of our case studies may depend on the purpose of specifying a product line with contracts. For most product lines, the main purpose of specifying contracts was

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

A further influence on our results might be that the case studies have been applied between 2010 and 2014 (cf. Appendix A), some even before we started to systematically discuss contract-composition mechanisms [Thüm et al., 2012]. That is, the product lines *IntegerList* and an initial version of *BankAccount* have been developed while creating a new contract-composition mechanism that fits the needs of the product line [Scholz et al., 2011; Thüm et al., 2011b]. However, both product lines have lead to similar statistics as other product lines. In particular, by means of a manual inspection in August 2014, we have retrieved statistics on all product lines at the same time to mitigate threats due to different knowledge on contract composition.

4.6 Related Work

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

Contracts in Feature-Oriented Programming

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

We started to investigate to use contracts for feature modules with the purpose of product-line verification [Scholz et al., 2011; Thüm et al., 2011b]. However, the verification techniques required a contract-composition mechanism to compose feature-oriented contracts into product specifications. Driven by the subject product lines, we

used cumulative contract refinement [Scholz et al., 2011] and contract overriding [Thüm et al., 2011b]. With the insights of the previous section, we know that consecutive contract refinement and explicit contract refinement should have been used instead. Then, we proposed further contract-composition mechanisms and systematically discussed their advantages and disadvantages [Thüm et al., 2012]. As proof-of-concept and to enable larger evaluations, we developed tool support for contract composition in FEATUREHOUSE and FEATUREIDE [Benduhn, 2012; Proksch and Krüger, 2014; Weigelt, 2013]. Based on explicit contract refinement, we proposed further verification techniques [Meinicke, 2013; Thüm et al., 2013; Thüm et al., 2014; Thüm et al., 2012], which we discuss in Chapter 5.

Contracts for Mixins

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

Contracts in Delta-Oriented Programming

Contracts have also been proposed for delta-oriented programming, which is similar to feature-oriented programming; delta modules are basically feature modules that can also remove classes and members [Schaefer et al., 2010a]. Bruns et al. [2011] propose a mechanism analog to contract overriding in feature modules, but also permit the removal of contracts in delta modules. Damiani et al. [2012] define contracts similar to explicit contract refinement. However, their contracts can reference preconditions and postconditions of any other contract by means of uninterpreted assertions. These

uninterpreted assertions can be seen as a generalization of keyword original, with which only the contract being subject to refinement can be referenced. With the abstract behavioral specification (ABS) language, Hähnle and Schaefer [2012] propose cumulative contract refinement for delta modules, whereas the composition is technically implemented by means of a restricted form of explicit contract refinement. Hähnle et al. [2013] propose a form of explicit contract refinement that supports to add, modify, and remove preconditions, postconditions, assignable clauses, and specification cases. According to our evaluation results, a single contract-composition mechanism is usually not sufficient and instead several mechanisms should be combined when specifying a product line.

Contracts in Aspect-Oriented Programming

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

Clifton and Leavens [2002] classify aspects into *observers* and *assistants*. An observer (a.k.a. spectator [Clifton, 2005]) is supposed to adhere to contracts of methods it advises, and thus is similar to plain contracting. In contrast, the specification of an assistant is composed similar to conjunctive contract refinement. For assistants, the effective assignable clause is the union of all assignable clauses defined for that method and its pieces of advice, and the condition in the signals clause is composed by means of a conjunction.

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

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

Molderez and Janssens [2012] propose ContractAJ, in which aspects have to adhere to the advice substitution principle. However, in 2014, Molderez and Janssens [2015] adapted ContractAJ to support two contract-composition mechanisms, which are similar to plain contracting and explicit contract refinement. To maintain modular reasoning, they propose to explicitly mention all pieces of advice that do not adhere to the advice substition principle with keyword @advisedBy. Their keyword proc is similar to keyword original and is actually the first time that previous contracts can be referenced explicitly in aspect-oriented programming. However, a problem is that all pieces of advice not mentioned in @advisedBy clause are oblivious to each other, and thus they may define a new contract but no other aspect can rely on it. In contrast, when composing contracts with any of our mechanisms, callers can rely on changed contracts.

There are several further approaches combining contracts with aspect-oriented programming [Griswold et al., 2006; Rebêlo et al., 2011, 2014; Rebêlo et al., 2013a; Rebêlo et al., 2008; Rebêlo et al., 2013b, 2008]. However, these approaches consider contracts as crosscutting concern that is modularized by means of aspects, which initiated a controversial discussion [Balzer et al., 2006; Rebêlo et al., 2014].

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

Contracts in Object-Oriented Programming

For object-oriented programming, several versions of behavioral subtyping and specification inheritance have been proposed [America, 1991; Dhara and Leavens, 1996;

Findler et al., 2001; Hatcliff et al., 2012; Liskov and Wing, 1994; Meyer, 1988], whereas newer versions are usually supposed to completely replace older ones. With cumulative contract refinement and consecutive contract refinement, we discussed two mechanisms being similar to behavioral subtyping as proposed by Meyer [1988] and specification inheritance as proposed by Dhara and Leavens [1996], respectively. For feature-oriented contracts, our case studies revealed that consecutive contract refinement supersedes cumulative contract refinement, which indicates that specification inheritance supersedes behavioral subtyping, too. The main difference to our work is that we allow programmers to use several mechanisms, because consecutive contract refinement does not apply to all feature-oriented contracts. Similarly, not all subclasses are necessarily behavioral subtypes, which is especially challenging when specifying existing designs posterior; either we cannot specify all properties of superclasses or we must change the design. Thus, having the option to use several contract-composition mechanisms could also be beneficial for object orientation. Interestingly, America [1991] has already proposed to distinguish between inheritance for code reuse and actual behavioral subtypes, but unfortunately his idea is not implemented in today's behavioral interface specification languages such as Eiffel, JML, and Spec#. Even if not establishing any of the preservation properties discussed above, conjunctive contract refinement is also applied to object orientation in the runtime assertion checker JMSAssert.[10]

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

Product-Line Specification with Contracts

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

For product-line development, Kästner et al. [2011a] distinguish between *open-world view* and *closed-world view*. In a closed-world view, we know all feature modules when

---

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

reasoning about them, whereas, in an open-world view, feature modules may be added that are unknown. In an open-world view, we need to reason about a set of feature modules without knowing all possible feature modules. In particular, Smaragdakis and Batory [2002] argue that feature-oriented programming is difficult to use in open, collaborative developments. However, our contract-composition mechanisms (e.g., consecutive contract refinement) may be used to facilitate reasoning for feature-oriented programming even in an open-world view. That is, we can assign contract-composition keywords to certain methods to enable modular reasoning.

4.7   Summary

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

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

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

Specifying the intended behavior of a product line is necessary for a variety of applications, such as formal verification, feature-interaction detection, and testing. How these applications can be achieved by means of feature-oriented contracts is discussed in the next chapter.

5. Feature-Oriented Contracts for Product-Line Verification

This chapter shares material with the VAST'11 paper "Proof Composition for Deductive Verification of Software Product Lines" [Thüm et al., 2011b], the FOSD'11 paper "Automatic Detection of Feature Interactions using the Java Modeling Language: An Experience Report" [Scholz et al., 2011], the GPCE'12 paper "Family-Based Deductive Verification of Software Product Lines" [Thüm et al., 2012], the MASPEGHI'13 paper "Subclack: Feature-Oriented Programming with Behavioral Feature Interfaces" [Thüm et al., 2013], and the SPLC'14 paper "Potential Synergies of Theorem Proving and Model Checking for Software Product Lines" [Thüm et al., 2014].

In the previous chapter, we systematically discussed how to specify the intended behavior of a product line by means of feature-oriented contracts. In this chapter, we show how feature-oriented contracts can be utilized for product-line verification. Our focus is on deductive verification techniques, because we identified it as underrepresented research area (cf. Chapter 3). In particular, for the first time, we compose human-written proof scripts by the same syntactic means as source code and specifications, as presented in Section 5.1. The decomposition of source code and specifications into feature modules allows us to reduce the verification effort by providing proof scripts for feature modules and to compose them for each product.

Furthermore, we found that there was not a single family-based approach for deductive product-line verification prior to this thesis (cf. Section 3.6). We share our insights with family-based theorem proving in Section 5.2. In addition, we exploit an advantage of feature-oriented contracts over other specification techniques; feature-oriented contracts enable us to apply several verification techniques to the very same product-line implementation and specification. We are the first to apply model checking to a product-line

specification in terms of contracts, and to empirically compare product-line theorem proving and model checking.

In Section 5.3, we discuss further, preliminary results with respect to contract-based product-line verification. In detail, we show how to detect syntax and composition problems for feature-oriented contracts. Then, we share our experiences of applying static analyses to feature modules specified by means of feature-oriented contracts. Finally, we discuss how to locate defect feature modules with behavioral feature interfaces.

5.1   Feature-Product-Based Theorem Proving

Compared to product-based analyses, the feature-product-based analysis strategy aims to reduce the analysis effort by performing part of the analysis on features in isolation and to reuse these results in the product-based part. In contrast to existing approaches for feature-product-based theorem proving (cf. Section 3.5), we propose the first approach applying the principle of uniformity to machine-readable proof scripts. Batory et al. [2004] introduced the *principle of uniformity* stating that all artifacts, such as source code, documentation, specification, and test cases, should be generated in a similar manner. That is, as feature modules and feature-oriented contracts are composed by means of superimposition, we apply superimposition also to proof scripts – to which we refer to as *proof composition*. In particular, proofs are modularized into feature modules and are automatically composed based on a selection of features similar to source code and contracts.

The underlying assumption of proof composition is that the creation of proof scripts is way more expensive than only checking the correctness of an existing proof script. This is the case especially for interactive theorem proving, in which proof scripts are created by a human and then checked by a machine [Bertot and Castéran, 2004]. That proof checking is easier than proof finding is also assumed for *proof-carrying code* [Necula, 1997] and in fact, proof composition can be seen as an instance of proof-carrying code. Each feature module comes with a proof and these proofs can be checked after product derivation and before product execution.

In Section 5.1.1, we give a brief introduction how interactive theorem proving can be applied to individual products according to a product-based strategy. In Section 5.1.2, we present proof composition for feature-product-based interactive theorem proving. While our considerations are based on WHY/KRAKATOA [Filliâtre and Marché, 2007] and COQ [Bertot and Castéran, 2004], we are confident that proof composition can also be applied to other proof-obligation generators and proof assistants. In Section 5.1.3, we share our insights of an empirical comparison between proof composition and product-based interactive theorem proving.

5.1.1   Product-Based Interactive Theorem Proving

When using interactive theorem proving for software verification, a verification tool is required to automatically generate proof obligations (i.e., theorems) for a given a

```
 1  class Account {                                      feature module BankAccount
 2      //@ invariant balance_non_negative: balance>=0;
 3      int balance = 0;
 4      //@ ensures balance == 0;
 5      Account() {}
 6      [...]
 7  }
 8  class Account {                                          feature module DailyLimit
 9      final static int DAILY_LIMIT = −1000;
10      //@ invariant withdraw_in_limit: withdraw >= DAILY_LIMIT;
11      int withdraw = 0;
12      [...]
13  }
```

Listing 5.1: Excerpt of product line *BankAccount* to illustrate proof composition.

program and its specification [Filliâtre and Marché, 2007]. These proof obligations are then proven interactively by a human with several proof commands – to which we refer to as *proof script*. The proof script is then checked for correctness by a further tool, such as a proof assistant. Our examples and our evaluation are based on the verification platform WHY/KRAKATOA [Filliâtre and Marché, 2007] and the proof assistant COQ [Bertot and Castéran, 2004]. WHY is a verification platform providing a language for proof obligations. KRAKATOA is a plug-in for WHY that parses a JML-annotated Java program and produces proof obligations in the WHY language. Using WHY, proof obligations can be exported to several automated theorem provers and proof assistants. We selected the proof assistant COQ, because of our expertise in theorem proving with COQ from previous work [Kästner et al., 2012a; Thüm, 2010].

In the following, we illustrate proof composition using a tiny excerpt of product line *BankAccount* (cf. Appendix A). We show two feature modules for features *BankAccount* and *DailyLimit* in Listing 5.1, which consist of one class fragment for class Account each. In feature module *BankAccount*, field balance is introduced with invariant balance_non_negative and a constructor is defined with a postcondition. Similarly, the fields withdraw and DAILY_LIMIT are introduced with invariant withdraw_in_limit in feature module *DailyLimit*. Even though this example appears to be overly simplistic, it already illustrates an interesting feature interaction: depending on whether feature *DailyLimit* is selected or not, the constructor defined in feature module *BankAccount* has to establish invariant withdraw_in_limit or not.

A proof obligation generated by WHY typically consists of several premises and one conclusion. A human is supposed to write proof commands to simplify and transform the conclusion into formulas being equivalent to the premises. In Listing 5.2, we give an example for the proof obligation that the constructor of class Account in configuration {*BankAccount*} establishes all invariants. In particular, only invariant balance_non_negative is available in this configuration. A separate proof obligation is

```
1   (*Why goal*) Lemma cons_Account_safety_po_1 :
2   forall (this_1: (pointer Object)),
3   forall (Account_balance: (memory Object int32)),
4   forall (Object_alloc_table:(alloc_table Object)),
5   forall (HW_1: (valid_struct_Account this_1 0 0 Object_alloc_table)),
6   forall (result: int32),
7   forall (HW_2: (integer_of_int32 result) = 0),
8   forall (Account_balance0: (memory Object int32)),
9   forall (HW_3: Account_balance0 = (store Account_balance this_1 result)),
10  (* JC_17 *) (balance_non_negative this_1 Account_balance0).
11  Proof.
12  intuition.
13  unfold balance_non_negative.
14  replace Account_balance0 with (store Account_balance this_1 result).
15  rewrite select_store_eq; trivial.
16  omega.
17  Save.
```

Listing 5.2: Proof that constructor of class Account establishes invariants for configuration {*BankAccount*} [Thüm et al., 2011b].

generated by WHY stating that the constructor fulfills its postcondition, which is not shown here.

The proof obligation in Listing 5.2 contains a name in Line 1, which is generated by WHY to identify the proof in case that we regenerate proof obligations after changing the source code. Lines 2–9 show premises that can be used to prove the conclusion in Line 10. For this proof obligation, we wrote the proof script surrounded by Proof and Save, which has been checked with COQ v8.3. For our discussion of proof composition, it is not necessary to understand these proof commands in detail. Instead, we refer interested readers to dedicated literature on interactive theorem proving with COQ [Bertot and Castéran, 2004; Coq Development Team, 2010].

To illustrate our motivation to propose proof composition, we show the same proof obligation for a different configuration in Listing 5.3. In this new configuration, the features *BankAccount* and *DailyLimit* have been selected. Due to the similarity of the source code and specification of both configurations, proof obligation and proof are quite similar. However, there are three main differences. First, new premises were added at the Lines 4 and 11–14 because of the new fields. Second, there is a new conclusion to prove in Line 16 stating that invariant withdraw_in_limit is established. Third, the Lines 23–28 contain new proof steps, which we wrote to prove the additional conclusion.

An unoptimized product-based strategy can be applied to interactive theorem proving as follows. For each valid configuration, we compose selected feature modules including their feature-oriented contracts. Then, we use WHY/KRAKATOA to generate proof obligations and write proof scripts for each product individually. However, as our

```
1   (∗Why goal∗) Lemma cons_Account_safety_po_1 :
2   forall (this_3: (pointer Object)),
3   forall (Account_balance: (memory Object int32)),
4   forall (Account_withdraw: (memory Object int32)),
5   forall (Object_alloc_table:(alloc_table Object)),
6   forall (HW_1: (valid_struct_Account this_3 0 0 Object_alloc_table)),
7   forall (result: int32),
8   forall (HW_2: (integer_of_int32 result) = 0),
9   forall (Account_balance0: (memory Object int32)),
10  forall (HW_3: Account_balance0 = (store Account_balance this_3 result)),
11  forall (result0: int32),
12  forall (HW_4: (integer_of_int32 result0) = 0),
13  forall (Account_withdraw0:(memory Object int32)),
14  forall (HW_5: Account_withdraw0 = (store Account_withdraw this_3 result0)),
15  (∗ JC_45 ∗) ((balance_non_negative this_3 Account_balance0)
16  ∧ (withdraw_in_limit this_3 Account_withdraw0)).
17  Proof.
18  intuition.
19  unfold balance_non_negative.
20  replace Account_balance0 with (store Account_balance this_3 result).
21  rewrite select_store_eq; trivial.
22  omega.
23  unfold withdraw_in_limit.
24  replace Account_withdraw0 with (store Account_withdraw this_3 result0).
25  rewrite select_store_eq; trivial.
26  replace (integer_of_int32 result0) with 0.
27  unfold Account_DAILY_LIMIT.
28  rewrite int32_coerce; omega.
29  Save.
```

Listing 5.3: Proof that constructor of class **Account** establishes invariants in configuration {*BankAccount, DailyLimit*}, whereas major differences to Listing 5.2 are highlighted [Thüm et al., 2011b].

previous example indicates, parts of proofs are identical for different products, leading to redundant, human effort. In contrast, the idea of proof composition is to modularize proof scripts with respect to features, and to compose them for a given configuration together with feature modules and contracts.

5.1.2   Proof Composition for Interactive Theorem Proving

Proof composition has two main ingredients: partial proofs and their composition. A partial proof consists of a name and a proof script. We give an example for partial proofs of feature modules *BankAccount* and *DailyLimit* in Listing 5.4. The name is necessary for two reasons. First, we use the name to know which partial proofs have to be composed, similar to class and method names in feature modules. Second, the name

```
1  (∗Why goal∗) Lemma cons_Account_safety_po_1 :            feature module BankAccount
2  Proof.
3  intuition.
4  unfold balance_non_negative.
5  replace Account_balance0 with (store Account_balance this_1 result).
6  rewrite select_store_eq; trivial.
7  omega.
8  Save.
```

```
9  (∗Why goal∗) Lemma cons_Account_safety_po_1 :            feature module DailyLimit
10 Proof.
11 unfold withdraw_in_limit.
12 replace Account_withdraw0 with (store Account_withdraw this_3 result0).
13 rewrite select_store_eq; trivial.
14 replace (integer_of_int32 result0) with 0.
15 unfold Account_DAILY_LIMIT.
16 rewrite int32_coerce; omega.
17 Save.
```

Listing 5.4: Partial proofs that constructor of class Account establishes invariants.

is needed by WHY to identify where the proof obligation need to be generated (i.e., which proof obligation belongs to which composed proof). Partial proofs are composed by concatenating their proof scripts in the same order as defined for the feature modules. Then, WHY is used to generate the according proof obligation. The conceptual result of composing these partial proofs and calling WHY is shown in Listing 5.3.

With partial proofs, we exploit WHY's support for software evolution. Given that we generated proof obligations and wrote proofs for a given program, we may want to change the program and verify it again. In this case, WHY makes sure that existing proofs remain untouched while proof obligations are updated according to the program changes. For our partial proofs, WHY identifies proofs by their name and generates the required proof obligations.

A central question is whether simple concatenation of partial proofs is sufficient for refinements of feature modules and feature-oriented contracts. To investigate this question, we systematically discuss all possible cases of refinements that may belong to a single feature. For each case, we consider corresponding changes to proof obligations. Then, we discuss the consequences of these changes to proof obligations to the proof script. In our discussion, we abstract from technical issues, which we discuss afterward. In detail, we distinguish between the following, typical refinements of feature modules:

- *Adding a new field.* We get no new proof obligations for any class. A new premise that represents the initialization of the field is added to the proof obligations for constructors (cf. Case 2 below).

- *Adding a new invariant.* We need to prove for every method and constructor of the class that the invariant is fulfilled after execution. This changes proof obligations concerning constructors and existing methods (Case 3). New proof obligations are generated in case there were no invariants before (Case 1).

- *Adding a new method with or without a contract.* No existing proof obligations are changed. However, if the class contains invariants or if the method has a contract, we get new proof obligations that the method fulfills the existing invariants or the newly added contract (Case 1).

- *Refining a method with consecutive contract refinement.* No existing proof obligations are changed, because we only allow to refine a method according to consecutive contract refinement. We get new proof obligations to show that the refined method fulfills the existing invariants and the contract refinement (Case 1).

We assume that all contract refinements are defined by means of consecutive contract refinement to increase the potential of proof reuse. In case of arbitrary contract refinements, we would need to adapt all those proof scripts relying on the original contract (i.e., the contract that is subject to the refinement). In the worst case, it could mean that not a single proof can be reused. However, it is out of our scope to evaluate the reuse potential beyond consecutive contract refinement and a possible target for future work. We refer interested readers to related work on abstract contracts with similar goals [Bubel et al., 2014].

These four changes to feature modules induce changes to proof obligations that can be classified into the following three cases. For each change of proof obligations, we elaborate on the consequent changes to proof scripts:

Case 1 *New proof obligations.* For every newly created proof obligation, we can write a new partial proof for the new feature. Proof composition simply copies it to the proof of the composed program.

Case 2 *New premises at proof obligations.* A new premise does not imply a change to the proof steps.

Case 3 *New cases at proof obligations.* The changed proof obligations contain a new conclusion for which we need additional proof steps at the end of the proof. This can be achieved as illustrated in Listing 5.3 and Listing 5.4 (i.e., creating a new partial proof including these additional proof steps). Proof composition will concatenate the partial proofs according to the selection of features.

Technical Issues of Proof Composition

The previous case distinction shows that proof composition is generally possible. Next, we want to discuss some technical problems we faced with WHY/KRAKATOA and give suggestions for solutions.

First, whenever composing a module introducing an invariant, it should be added below all other invariants. This ordering simplifies proof composition, since new cases always appear at the end of proof obligations and new proof steps can always be added below existing proof steps. For example, when composing the feature modules shown in Listing 5.1, the invariants of feature *DailyLimit* should be added below the invariants of feature *BankAccount*.

Second, WHY and KRAKATOA generate names for premises, assignments, nested expressions, and for the current object. These names are needed to reference these entities in proof steps. The problem is that name generation is very fragile in terms of changes to the source code. For example, variable this_1 in Listing 5.2 is named this_3 in Listing 5.3. That is, it depends on the feature selection and the reference in the partial proof for feature *BankAccount* in Listing 5.4 has to be renamed during composition. We suggest to minimize references to generated names, which is possible to a certain degree. Remaining references can mostly be updated automatically based on the generated proof obligation.

Third, during feature-module composition, refined methods are usually renamed and the keyword original is replaced by a call to that renamed method [Apel et al., 2013b]. Since the name of a proof obligation is generated using a method's name, we also need to rename the partial proofs, such that they still match to the proof obligation. Fortunately, the renaming of methods is predictable and renaming partial proofs can be done automatically when composing proofs.

Finally, WHY requires all proof obligations to be alphabetically ordered. However, proof composition can easily take the order into account when composing partial proofs. Alternatively, a postprocessing step after composition could order all proofs.

5.1.3 Evaluation with Why/Krakatoa and Coq

The practicability of proof composition is not yet clear from our previous discussions. One open question is whether it actually saves proof effort compared to product-based interactive theorem proving. Another questions is whether the above discussed cases are sufficient and how proof composition can deal with feature interactions. We aim to answer these questions by verifying a small version of the product line *BankAccount* with five features and 12 program variants (cf. Appendix A). Verifying larger or several product lines was not feasible in the course of this thesis, as formal verification with COQ already requires heavy user interaction even for small programs.

We created hand-written partial proofs for every feature module, in which we omitted the proof obligation including all premises and the conclusion. Then, we generated every program variant including its specification in JML and composed the partial proofs of the features included in the configuration. We used KRAKATOA and WHY to generate the proof obligations into our composed proofs and checked the correctness of the proofs using COQ for all 12 program variants.

As proof composition is based on interactive theorem proving, measuring the time for proof checking does not seem valid for evaluation. Indeed, proof composition does
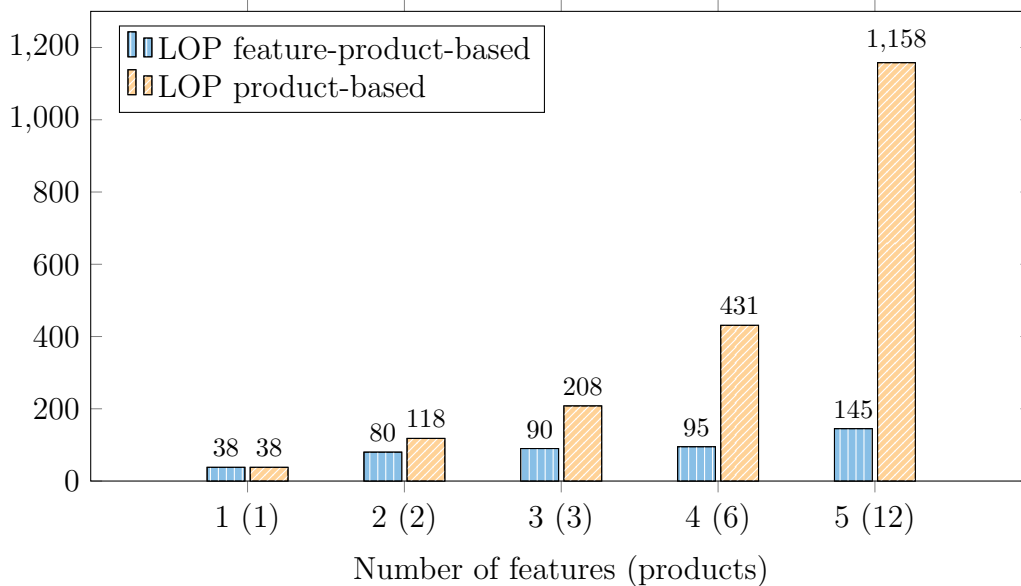
Figure 5.1: Lines of proof for proof composition and product-based theorem proving.

not simplify proof checking at all; still each program of the product line needs to be generated to check the validity of the proofs. Additionally, the time to check proofs is negligible compared to the time a human being needs to come up with the proof script. Hence, we estimate the effort for writing partial proofs for each feature and compare it with the effort of writing proofs for each program variant. We assume that the effort for proof writing is related to the number of proof steps, and so we introduce the measure *lines of proof* (LOP) that counts the number of proof steps, in analogy to the measure lines of code. For example, the LOP for Listing 5.2 and Listing 5.3 are 5 and 11, respectively.

In Figure 5.1, we show the measure LOP of all partial proofs and compare it with the LOP of all program variants. In particular, we wrote partial proofs for all five feature modules, which sum up to 145 LOP. In contrast, the result of composing these partial proofs for each program variant results in a total of 1,158 LOP. That is, proof composition reduces the LOP by 88 %. To illustrate the scalability of proof composition even with our small product line, we decided to measure both values also when removing some of the features, leading us to product lines with one, two, three, four, and five features. The diagram illustrates that we can already save proof effort for two products and that the potential to reuse proofs increases with the number of features and products. For an overview on the reuse potential for each feature, we refer to the original publication on proof composition [Thüm et al., 2011b], in which we also discuss when derivative modules are required due to feature interactions.

Discussion and Experiences

Using proof composition, we were able to save time in verifying the correctness of all program variants in our case study. In order to generalize these results, further case

studies are needed. Case studies may evaluate the potential of proof composition for
other domains or larger code bases. In particular, it is unclear whether proof composi-
tion can generate proofs automatically for all program variants in practice. Problems
can arise with feature interactions (i.e., we need to prove something only if two features
are contained in the same configuration).

Our approach is designed for languages and tools that rely on superimposition [Apel
et al., 2013b], which raises the question of whether proof composition can be used
with other product-line implementation techniques such as delta-oriented program-
ming [Schaefer et al., 2010a] or preprocessors [Liebig et al., 2010]. For delta-oriented
programming, we would further need to be able to remove proofs or parts of proofs
(e.g., if we remove a field including its class invariant). Hence, a mechanism would be
required to refer to parts of proofs. When implementing product lines using preproces-
sors, we could imagine that preprocessor macros are used in proof documents as well.
Proof composition seems to fit well in this scenario and even easier than with feature
modules, as methods are not renamed.

We discovered several technical problems using Krakatoa and Why, but we assume that
similar problems will arise for other verification tools. The reason is, that we need to
refer to assignments, expressions, and instances in proofs for which names have to be
generated. These generated names may change when adding new members to a class.

Nevertheless, we have doubts on the practicability of proof composition for three rea-
sons. First, the question is how to come up with partial proofs. The tool support by a
proof assistant such as COQ is crucial for interactive theorem proving, but it cannot deal
with partial proofs. For our evaluation, we interactively proved the program variant
with all features and decomposed this proof into features, which could get complicated
for larger product lines. Second, a rather technical issue is that the generator of proof
obligations and the composition tool for proofs highly depend on each other. That is,
changes to one tool will most probably also require changes to the other tool, which
could turn out infeasible in practice. Finally, consecutive contract refinement is not ap-
plicable to a considerable amount of contract refinements (cf. Section 4.5) and arbitrary
contract refinements are likely to reduce the proof reuse with proof composition.

5.2 Family-Based Theorem Proving and Model Checking

We present how to verify product lines with feature-oriented contracts by means of
family-based analyses with two advantages over proof composition. First, contract
refinements do not need to be expressed with consecutive contract refinement. In-
stead, our approach is based on explicit contract refinement, which does not require
feature modules to apply behavioral subtyping (cf. Section 4.5). Second, we avoid the
generation of products and their redundant analysis during proof checking (i.e., the
product-based phase of proof composition).

As discussed in Section 3.3, family-based analysis can be implemented by making analyses for single systems variability-aware or by transforming compile-time into runtime variability. We focus on the latter, which is known as *configuration lifting* [Post and Sinz, 2008] or *variability encoding* [Apel et al., 2013d], because existing analysis tools can be reused as-is. The novelty of our approach is that we are the first to extend variability encoding for specifications (i.e., contracts). Furthermore, we use variability encoding to scale theorem proving and model checking to product lines according to the open research challenges identified in our survey (cf. Section 3.3). For theorem proving, we present the first approach pursuing a family-based strategy. For product-line model checking, contracts have not yet been used for specification.

Applying variability encoding to feature-oriented contracts enables an empirical comparison of family-based theorem proving and model checking. Such a comparison has not yet been performed for product lines, but can expose whether one verification technique is superior for earlier or later stages in the development process. That is, efficiency and effectiveness may be influenced by the number of defects in the product line. However, existing evaluations of product-line verification techniques have either verified a product line with [Apel et al., 2011, 2013d; Classen et al., 2011] or without [Bodden et al., 2013; Brabrand et al., 2013; Liebig et al., 2013; Post and Sinz, 2008; Thüm et al., 2011b] defects, or have not compared these verification times [Kästner et al., 2012a,c; Kolesnikov et al., 2013; Thaker et al., 2007]. We consider the number of defects as independent variable and introduce artificial defects by means of mutation techniques as known from mutation testing [Jia and Harman, 2011].

In Section 5.2.1, we illustrate how to transform compile-time into runtime variability by means of variability encoding. In particular, we apply variability encoding to feature models, feature modules, and feature-oriented contracts. In Section 5.2.2, we present our tool support to automate variability encoding and the tool chains for family-based theorem proving and model checking. Finally, we share the results of our evaluation in Section 5.2.3.

5.2.1 Variability Encoding for Metaproduct Generation

Variability encoding is the process of transforming a product-line's compile-time variability into runtime variability with the purpose of reusing analyses from single-system engineering [Apel et al., 2013d]. Variability encoding takes several artifacts of the product line as input. First, the feature model is encoded to make sure that only valid feature combinations are considered during analysis. Second, the product-line implementation (i.e., feature modules) is transformed into runtime variability. Finally, a step that we introduce in this thesis is to also encode the product-line specification (i.e., feature-oriented contracts). The result of variability encoding is a *metaproduct* consisting of a metaprogram simulating all program variants and a metaspecification simulating all product specifications. Both, metaprogram and metaspecification are ideally expressed in existing programming and specification languages from single-system engineering, to enable the reuse of existing tools for analysis.
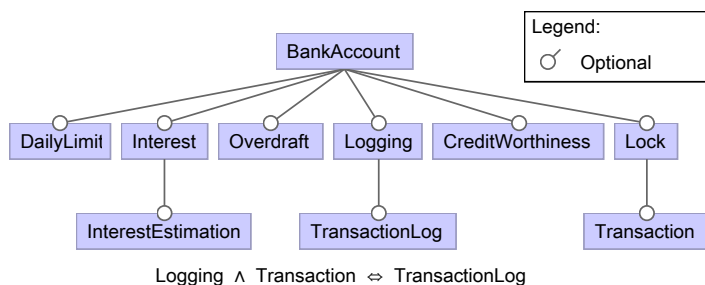
Figure 5.2: Feature model for bank account software [Thüm et al., 2014].

Variability Encoding for Feature Models

The feature model is encoded into the metaproduct to simulate exactly those feature selections that are valid. A boolean class variable is created for each feature, whereas the variable assignment **true** indicates that the feature is selected and **false** that it is not. Verification tools are configured to treat those *feature variables* as not initialized to consider all combinations. The dependencies between features are translated into a propositional formula into the host language (e.g., Java in our case). The translation of feature models into propositional formulas is well-known [Batory, 2005; Thüm et al., 2011a]. This formula is then used to prohibit any execution of non-valid feature selections.

In Java, the point where the program starts may not be unique. For example, a Java program may have several main methods in different classes. Alternatively, we may also want to verify a Java library and basically every static method or constructor may be the entry point of the program. Our solution is to use Java's class loading for initialization of feature variables. We add a class called **FeatureModel** containing all feature variables. When this class is accessed for the first time, the class is loaded and random values are assigned to each feature variable, to make sure that the values are arbitrary. Another option would be to prompt the user for providing values. Hence, verification tools cannot rely on particular values and *all* feature combinations are verified.

Example 5.1. *In Figure 5.2, we give an example feature model for a variable bank account software that we use to illustrate variability encoding. In Listing 5.5, we show a class that encodes the bank account feature model. Feature variables are modeled as static fields and initialized in a static constructor using random values. The dependencies between features are modeled using a propositional formula (cf. Section 2.2.1), which is encoded as a Java expression (Lines 11–13). If the random initialization is invalid according to the feature model, the program is terminated to avoid verification of invalid feature combinations (Line 8). If the initialization is valid, we can assume the feature model as an invariant, because the feature variables are declared as* final *(Line 2).*

Variability Encoding for Feature Modules

Based on variability encoding of the feature model, we discuss how to compose feature modules into the metaproduct. Compared to the generation of products from feature

```
1  class FeatureModel {                                              metaproduct
2    //@ static invariant fm();
3    final static boolean bankAccount, dailyLimit, interest, interestEstimation,
4      overdraft, logging, transactionLog, creditWorthiness, lock, transaction;
5    static {
6      bankAccount = random();
7      //initialization of other variables
8      if (!fm()) System.exit(1);
9    }
10   /*@ pure @*/ static boolean fm() {
11     return bankAccount && (!(dailyLimit || interest || overdraft || logging
12       || creditWorthiness || lock) || bankAccount) && (!interestEstimation || interest)
13       && (!transactionLog || logging) && (!transaction || lock);
14   }
15   static boolean random() { return Math.random() < 0.5; }
16 }
```

Listing 5.5: Variability encoding of the feature model given in Figure 5.2 (adapted from [Thüm et al., 2012]).

modules, variability encoding of feature modules has two differences. First, whereas a configuration is taken as input for product generation, variability encoding is used to compose *all* feature modules instead of a selection. Second, the composition of feature modules is based on feature variables to simulate the behavior of all products. That is, branching statements are added to switch between different implementations depending on the configuration at runtime.

The transformation of feature modules into a metaproduct (a.k.a. product simulator) was proposed by Apel et al. [2011], which we adapt for our purpose. All feature modules of the product line are composed by merging class introductions with their respective class refinements. The crucial case is how to handle method refinements, because the method body of a method and its refinements depend on the configuration. In a nutshell, we generate a distinct method for every method introduction and method refinement, where methods that are subject to refinement are renamed to distinguish them in the resulting metaproduct. At the beginning of each method refinement, we add a branching statement to check at runtime, whether the corresponding feature of the refinement is selected or not. If the feature is selected, we continue with the implementation introduced by this feature, and if not, we call the original method (i.e., the previous method refinement or the method introduction). Similarly, variability encoding can be applied to constructors and fields. We refer interested readers to a dedicated bachelor's thesis [Meinicke, 2013].

Example 5.2. *We illustrate variability encoding of feature modules by means of our running example. In Listing 5.6, we show excerpts of three feature modules of product line* BankAccount *(ignore feature-oriented contracts for now). Feature module* BankAccount *introduces class* Account *with a method and two fields. In our excerpt, feature*

```
1   class Account {                                          feature module BankAccount
2     final static int OVERDRAFT_LIMIT = 0;
3     //@ invariant balance >= OVERDRAFT_LIMIT;
4     int balance = 0;
5     /*@ ensures (!\result ==> balance == \old(balance))
6       @ && (\result ==> balance == \old(balance) + x);  @*/
7     boolean update(int x) {
8       int newBalance = balance + x;
9       if (newBalance < OVERDRAFT_LIMIT) return false;
10      balance = newBalance;
11      return true;
12    }
13  }
```

```
14  class Account {                                              feature module Overdraft
15    final static int OVERDRAFT_LIMIT = −5000;
16  }
```

```
17  class Account {                                             feature module DailyLimit
18    final static int DAILY_LIMIT = −1000;
19    //@ invariant withdraw >= DAILY_LIMIT;
20    int withdraw = 0;
21    /*@ ensures \original && (!\result ==> withdraw == \old(withdraw))
22      @ && (\result ==> withdraw<=\old(withdraw)); @*/
23    boolean update(int x) {
24      int newWithdraw = withdraw;
25      if (x < 0) { newWithdraw += x; if (newWithdraw < DAILY_LIMIT) return false; }
26      if (!original(x)) return false;
27      withdraw = newWithdraw;
28      return true;
29    }
30  }
```

Listing 5.6: Class **Account** with feature-oriented contracts and two class refinements (adapted from [Thüm et al., 2012]).

module Overdraft *only refines the value of a field, and feature module* DailyLimit *refines method* update *and adds two fields to class* Account. *The result of applying variability encoding to these three feature modules is shown in* Listing 5.7. *In the metaproduct, class* Account *contains the fields* balance, withdraw, OVERDRAFT_LIMIT, *and* DAILY_LIMIT, *as defined in the respective feature modules. Method* update, *defined in feature module* BankAccount, *is renamed to* update$$BankAccount *(Lines 7–12). In the refinement of method* update *(Lines 24–31), defined originally in feature module* DailyLimit, *a new branching statement is added as first statement. If feature* DailyLimit *is not selected, the original method is called and then the method returns. Otherwise, the method is executed as defined in feature module* DailyLimit.

```
1  class Account {                                             metaproduct
2    final static int OVERDRAFT_LIMIT = FeatureModel.overdraft ? −5000 : 0;
3    //@ invariant balance >= OVERDRAFT_LIMIT;
4    int balance = 0;
5    /*@ ensures (!\result ==> balance == \old(balance))
6      @ && (\result ==> balance == \old(balance) + x); @*/
7    boolean /*@ helper @*/ update$$BankAccount(int x) {
8      int newBalance = balance + x;
9      if (newBalance < OVERDRAFT_LIMIT) return false;
10     balance = newBalance;
11     return true;
12   }
13   final static int DAILY_LIMIT = −1000;
14   //@ invariant FeatureModel.dailyLimit ==> withdraw >= DAILY_LIMIT;
15   int withdraw = 0;
16   /*@ ensures !FeatureModel.dailyLimit==>
17     @ (!\result ==> balance == \old(balance))
18     @ && (\result ==> balance == \old(balance)+x);
19     @ ensures FeatureModel.dailyLimit ==>
20     @ ((!\result ==> balance == \old(balance))
21     @ && (\result ==> balance == \old(balance)+x))
22     @ && (!\result ==> withdraw == \old(withdraw))
23     @ && (\result ==> withdraw <= \old(withdraw)); @*/
24   boolean update(int x) {
25     if (!FeatureModel.dailyLimit) return update$$BankAccount(x);
26     int newWithdraw = withdraw;
27     if (x < 0) { newWithdraw += x; if (newWithdraw < DAILY_LIMIT) return false; }
28     if (!update$$BankAccount(x)) return false;
29     withdraw = newWithdraw;
30     return true;
31   }
32 }
```

Listing 5.7: Metaproduct for class Account as defined in Listing 5.6 [Thüm et al., 2012].

Similarly to Kästner et al. [2012a], we assume type uniformity for all feature modules. That is, (a) all valid combinations of feature modules are well-typed *and* (b) the composition of *all* feature modules is well-typed. The first condition is necessary as only well-typed programs can be verified. The second condition is necessary as mutually exclusive features may introduce incompatible classes or class refinements, which may cause type errors in the metaproduct. For example, two mutually exclusive feature modules may introduce a field to a certain class with the same name but of incompatible types. In this case, the metaproduct is ill-typed, even though every valid product is well-typed. The problem can be solved by renaming based on a variability-aware type system [Apel et al., 2010a]. However, mutually exclusive features introducing incom-

patible types are rare in feature-oriented product lines [Apel et al., 2013b], and did not occur in our evaluation.

Furthermore, we assume that there is no *name shadowing* in the resulting metaproduct. Name shadowing can occur when a field of a class has the same name as a local variable (lexical shadowing) or if a field is defined in a class and its superclass (inheritance-based shadowing). For an example consider Listing 5.6 again: if method update in feature module *BankAccount* would declare a local variable balance, then the statement in Line 10 would access the local variable instead of field balance. With name shadowing, a certain statement can have different meanings depending on the configuration, but only one meaning in the metaproduct. That is, the metaproduct does not cover the behavior of all products. Fortunately, name shadowing can be detected statically. Nevertheless, it is out of our scope to handle name shadowing in metaproducts correctly.

Variability Encoding for Feature-Oriented Contracts

With variability encoding of the feature model and all feature modules, we can generate a metaprogram that simulates the actual behavior of all products. As we want to verify a product line with respect to its feature-oriented contracts, we need to apply variability encoding to feature-oriented contracts, too. The result is a metaspecification describing the intended behavior of all products. We propose a composition of feature-oriented contracts resulting the metaprogram specified by means of contracts, such that existing tools for contract-based verification can be reused for the metaproduct.

Contracts and invariants are encoded into the metaspecification by introducing implications with feature variables. Each precondition and postcondition (for short condition) c defined in a feature f is rewritten as $f \Rightarrow c$ to ensure that the condition is checked only if feature f is selected. Similar to the encoding of feature modules (cf. Line 25 in Listing 5.7), we need to specify the behavior if f is not selected. Given the previous condition c' in the refinement chain, we also add the condition $\neg f \Rightarrow c'$. As invariants do not need to be refined (cf. Section 4.3.4 and Section 4.5), it is sufficient to rewrite each invariant i in feature f as $f \Rightarrow i$. A simple optimization is to keep each specification as-is that is specified in a core feature (i.e., a feature that is included in every product), because its specification must be fulfilled by every product.

Beyond these implications, we need to replace all occurrences of keyword original in the contracts to reuse existing verification tools from single-system engineering. We replace every occurrence of original by the previous precondition or postcondition in the refinement chain. The order of replacement is important; it is necessary to start with method introductions and then to continue with every method refinement in the refinement chain to successfully remove all occurrences of the keyword.

Example 5.3. *The metaproduct shown in Listing 5.7 already contains a metaspecification. The invariant from feature module* DailyLimit *is transformed into an implication stating that it is established only if feature* DailyLimit *is selected (Line 14). In contrast, the invariant from feature module* BankAccount *is not transformed into an implication,*

because feature BankAccount *is a core feature and its specification must be fulfilled by all products (Line 3). For the same reason, the contract of the method introduction* update *is copied as-is to* update$$BankAccount*. The only change to the contract is that this method is annotated with the keyword* helper *to indicate that it does not need to fulfill class invariants. Note that all renamed methods must be marked as helper methods, because they are not intended to fulfill all class invariants (a detailed discussion follows in Section 5.3.3).*

The transformation for the contract of method refinement update*, defined in feature module* DailyLimit*, is more complex. We generate two postconditions stating the behavior for products containing the feature* DailyLimit *and those not containing it. If feature* DailyLimit *is not selected, the method guarantees the original postcondition as defined in feature module* BankAccount*. Hence, we copy the postcondition from method* update$$BankAccount *(Lines 16–18). Otherwise, if feature* DailyLimit *is selected, the method guarantees the contract as defined in feature module* DailyLimit*, in which we need to replace keyword* original *by the postcondition from method* update$$BankAccount *(Lines 19–23).*

5.2.2 Tool Support for Variability Encoding

With this thesis, we overcome limitations of existing tool support for variability encoding and family-based verification. Post and Sinz [2008] have proposed variability encoding, but manually assembled metaproducts, which is laborious and error-prone. Apel et al. [2011] presented the first tool support for variability encoding, which has been used also in later research projects [Apel et al., 2013d; Kästner et al., 2012c]. Their implementation of variability encoding is based on FEATUREHOUSE [Apel et al., 2013b] and available for feature modules in Java and C. However, this functionality was only available in the command-line interface of FEATUREHOUSE and did not support variability encoding of contracts. Furthermore, there was no tool support for family-based theorem proving and family-based model checking with contracts, as we identified in our survey on analysis tools [Meinicke et al., 2014].[1]

We extended variability encoding in FEATUREHOUSE for the composition of feature-oriented contracts into a metaspecification. We have chosen FEATUREHOUSE for two reasons. First, we already implemented the composition of feature-oriented contracts into product specifications in FEATUREHOUSE (cf. Section 4.4.1). Second, variability encoding of feature models and feature modules was already supported by FEATURE-HOUSE. Our extension is available in the master of the FEATUREHOUSE repository.[2]

In line with our integration of FEATUREHOUSE into FEATUREIDE (cf. Section 4.4.2), we extended the user interface of FEATUREIDE to make variability encoding available in Eclipse. We illustrate the interplay of FEATUREIDE and FEATUREHOUSE in Figure 5.3. FEATUREIDE passes feature modules and feature-oriented contracts

---

[1]http://fosd.de/tools
[2]https://github.com/joliebig/featurehouse
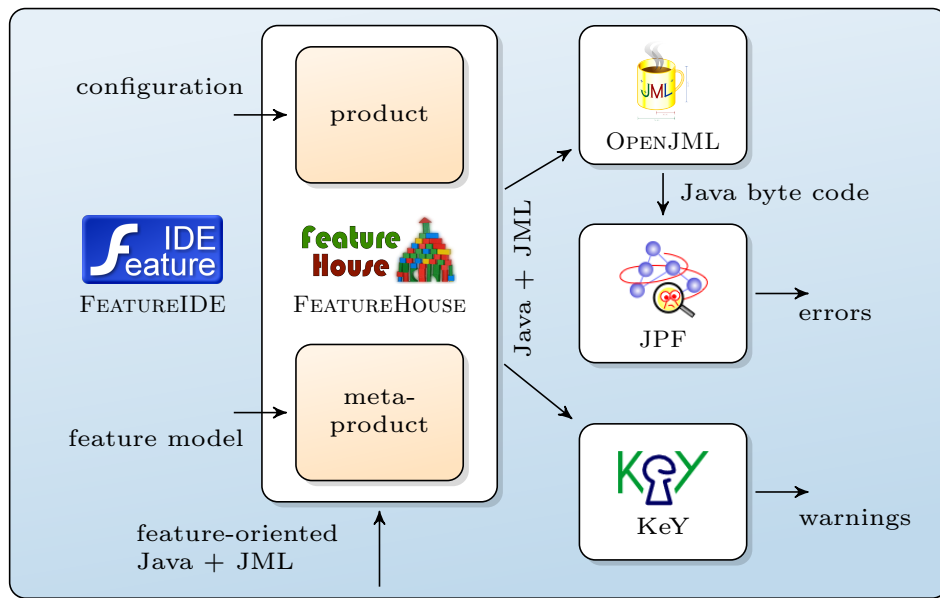
Figure 5.3: Product-line verification with feature-oriented contracts in the FeatureIDE framework [Thüm et al., 2014].

into FEATUREHOUSE. In addition, the configuration is passed for product genera-
tion and the feature model for metaproduct generation. In each case, the output is
a JML-annotated Java program (cf. Section 2.1.1), which can, in principle, be pro-
cessed and verified by means of any existing JML tool. However, different verification
techniques can require small adaptations of the metaproduct generation, as outlined
elsewhere [Meinicke, 2013; Thüm et al., 2014]. Nevertheless, we were able to reuse the
theorem prover KeY [Ahrendt et al., 2014; Beckert et al., 2007] and the software model
checker JPF [Havelund and Pressburger, 2000] as-is for product-line verification. As
JPF has no built-in support for contracts, we use OPENJML [Cok, 2011] to translate
contracts of the metaproduct into runtime assertions before model checking.

Our extension of FEATUREIDE enables the user to switch between product gener-
ation and metaproduct generation by means of a context menu in the Package Ex-
plorer. While error propagation was available for product generation before [Thüm
et al., 2014b], we implemented error propagation for the metaproduct. That is, if any
analysis tool, such as a compiler or verification tool, finds a defect in the metaproduct,
it will create an error marker in the metaproduct (i.e., in generated code). With error
propagation, we refer to the process of identifying the initial location in one of the
feature modules that lead to the error in the metaproduct. We integrated our extension
of FEATUREIDE into the master of the FEATUREIDE repository.[3]

---

[3]https://github.com/tthuem/FeatureIDE

5.2.3 Evaluation with Theorem Proving and Model Checking

Given our tool support in FEATUREIDE, it is possible to implement feature modules, specify feature-oriented contracts, and verify them using theorem proving and model checking by means of variability encoding. This gives rise to a number of questions. What are the benefits of combining theorem proving and model checking? Which verification technique is a programmer supposed to use when? How do the verification techniques scale depending on the number of features or defects? In the following, we describe our experiment to explore potential synergies of combining techniques, the results of the experiment, and threats to validity. We refer interested readers to our website containing screencasts, source code, and raw data for reproduction purposes.[4]

Experiment Subjects and Set-Up

In our experiment, we use product line *BankAccount* (cf. Appendix A), which we completely verified with KeY and JPF. In contrast to the tool support described above, we use MonKeY [Thüm et al., 2012], an extension of KeY that we created to provide a batch mode for automatic verification. For model checking, we use JPF-BDD [Apel et al., 2013d], an extension of JPF for product-line verification that symbolically encodes feature variables in a binary decision diagram for better performance. We deliberately introduced defects by means of mutations in feature modules and feature-oriented contracts, respectively. The goal of mutations is to simulate different phases of development (i.e., mature and less mature product lines). We measured the verification time and effectiveness of KeY and JPF for the product line containing no defects, some defects, and many defects.

Compared to Section 5.1.2, we extended product line *BankAccount* by four new features, to ten features overall [Meinicke, 2013; Thüm et al., 2014]. The feature model of the product line has already been presented in Figure 5.2, and a simplified version of three feature modules and their contracts have been shown in Listing 5.6. Overall, the product line consists of four classes, ten class refinements, 17 unique methods with a contract each, six class invariants, eight method refinements, and six contract refinements. Quantifiers and model methods were not necessary for specifying product line *BankAccount*. The test scenarios that are necessary for software model checking [Thüm et al., 2014] are composed along with the feature modules and achieve a method coverage of 100.0 %, an instruction coverage of 91.6 %, and a branch coverage of 72.2 % for the largest product. Based on this product line, we simulate different product-line sizes by successively removing existing features. The resulting product lines have between 2 and 10 features (2, 4, 6, 12, 24, 36, 36, 72, and 144 products).

While the goal of product-line verification is a defect-free product line, verification tools are used to detect defects on the way towards a verified product line. Consequently, an interesting characteristic for evaluating verification techniques is how they perform for product lines containing many, some, or no defects. While extending the bank account product line, we introduced defects, but they are too few to make any general

---

[4]http://fosd.de/spl-contracts

Source/Target	Target/Source	In Java	In JML	Sum
false	true	27	1	27
*	/	12	0	12
-	+	7	8	15
+=	-=	4	0	4
<	<=	7	5	12
>	>=	1	12	13
&&	||	0	11	11
==>	<==>	0	27	27
==	!=	0	37	37
Sum		58	101	159

Table 5.1: Mutations applied to feature modules and feature-oriented contracts of product line *BankAccount* [Thüm et al., 2014].

statements. Hence, we decided to automatically introduce defects as known from mutation testing [Jia and Harman, 2011]. We mutate feature modules before variability encoding to simulate realistic defects. Table 5.1 shows the considered mutation operators as well as the number of occurrences for our product line. These operators are typical for mutation testing [Jia and Harman, 2011]. As common in mutation testing, we use string replacements, as they are applicable to feature modules and contracts. With regular expressions, we identified possible positions for mutations and randomly selected mutations in our experiment.

We computed all experiments on a lab computer with Intel Core i5 CPU with 3.33 GHz, 8 GB RAM, and Windows 7. In all runs, we measured the time for verification with KeY and JPF. We created separate runs for no defects, one defect, and so on, until reaching ten defects, whereas each run was repeated 20 times with different, randomized mutations each to avoid computation bias and bias due to mutations. We stopped both tools after the first defect had been identified, because this time is more critical for developers than the overall verification time; a developer can investigate the first defect already and need to start verification again after fixing the defect anyway. In particular, we stopped KeY if a proof obligation could not be proven automatically. In general, an open proof obligation does not necessarily indicate a defect; the proof obligation may require user interaction. However, we inspected all open proof obligations for each single mutation and they all indicate a defect.

Effectiveness

We measured effectiveness as how often a verifier finds at least one defect, independent of whether the product line contains one, two, or more defects. In particular, we consider a verifier as *effective*, even if it finds less defects than the product line contains. The rational behind this decision is that developers typically work on one defect at a time and then verify the product line again.
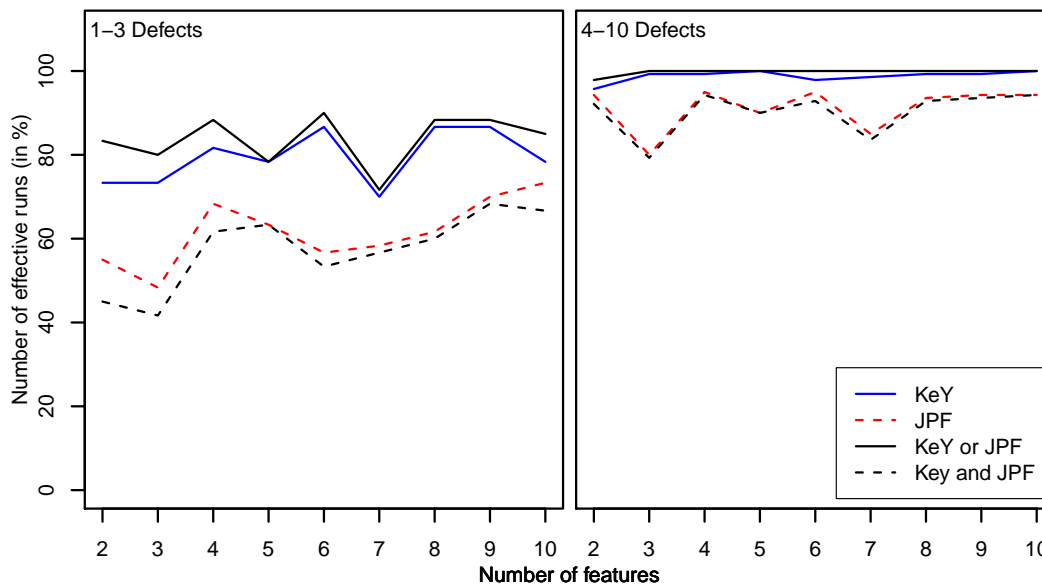
Figure 5.4: Effectiveness for finding a defect in product lines with some and many defects [Thüm et al., 2014].

In Figure 5.4, we show the effectiveness of theorem proving and model checking for some defects (left diagram) and many defects (right diagram). With regard to the size of our subject product line, we already consider more than three defects as many. Each run is either effective or not, and we show the percentage of runs in which KeY (solid, blue line) and JPF (dashed, red line) were effective. The x-axis represents the number of features of the product line in each run. Furthermore, we show the percentage of runs in which both, KeY and JPF, were effective (dashed, black line, named "KeY and JPF") and the percentage in which at least one of them was effective (solid, black line, named "KeY or JPF"). We computed the percentage for each number of defects separately and show mean values in the diagram (e.g., for one, two, and three defects in the left diagram).

These diagrams lead to the following observations. First, both verification techniques are more effective, when more defects are introduced, because it is more likely that they detect one defect. Second, JPF is, in general, less effective than KeY. A code inspection in these cases revealed that our test scenarios were not sufficient to detect several defects. Third, JPF detects some defects that KeY does not. One reason is that the test scenarios can expose defects in addition to the contracts. Another reason is that KeY does not check precondition violations of called methods when a method body is inlined instead of applying its contract. Fourth, the effectiveness varies for different sizes of product lines, which is caused by our rather small product line with only few mutations (58 mutations in Java and 101 mutations in JML, cf. Table 5.1). Finally, using both verifiers leads to better effectiveness compared to each of them. This is especially the case if there are only few defects in the product line. That the combined effectiveness is not 100 % is due to the fact that some mutations can simply not be detected with the given contracts and test scenarios. Overall, we found that
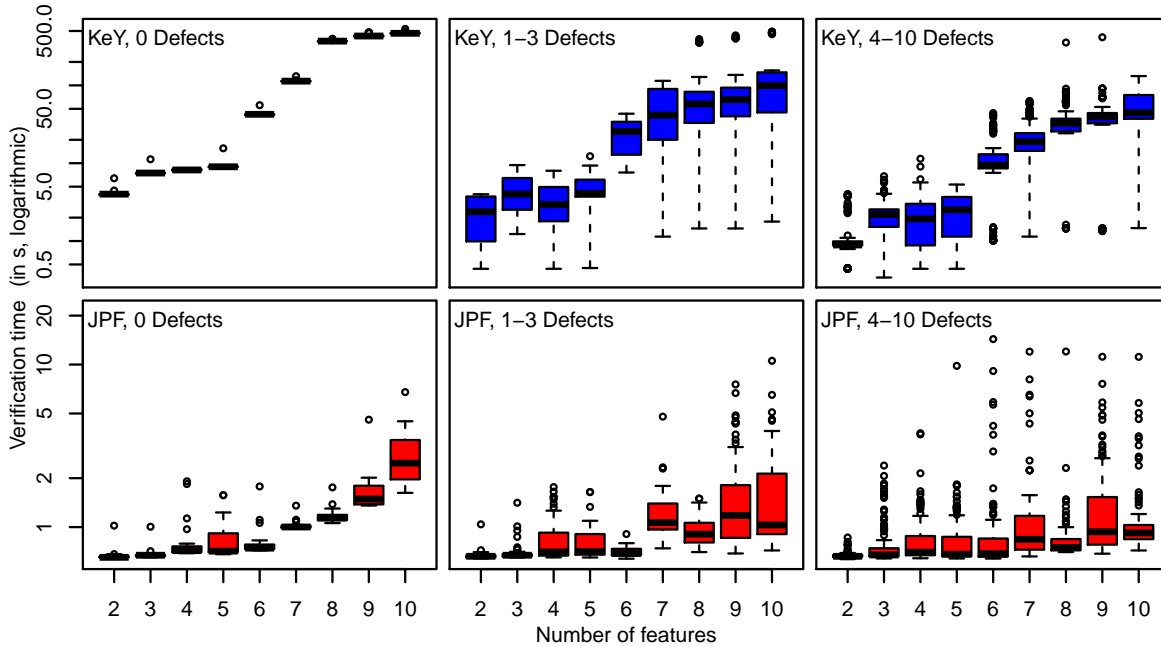
Figure 5.5: Performance for finding the first defect in product lines with no, some, and many defects [Thüm et al., 2014].

the combination of theorem proving and model checking improves the effectiveness, especially if there are only a few defects (i.e., in later development stages).

Performance

We assess the performance of theorem proving and model checking as the time needed to detect the first defect. If no defects were found, we consider the time to completely verify the product line. For JPF, we measured the time until the first runtime assertion was violated. For KeY, we measured the time until the first proof obligation could not be proven automatically. In Figure 5.5, we present the performance of KeY and JPF in box plots (created with default parameters of R). Note that the y-axes are logarithmic. Compared to effectiveness, we also consider the product line without defects.

Again, we make several observations. First, the most obvious result is that JPF is significantly faster than KeY (39.5 times in average). However, this result heavily depends on the test scenarios and should not be overestimated. Second, the verification time grows with the size of the product line, which is due to the larger code base. Third, the deviation of verification is larger for some and many defects than for no defects. The reason is that the computation bias is much smaller than the bias introduced by mutations; the left diagrams only show computation bias as we repeatedly verified the source code without mutations. Finally, the average verification time of both verifiers reduces when more defects are added; for KeY from 163.4 s to 50.7 s and 19.3 s, and for JPF from 1.19 s to 1.06 s and 1.01 s, for some and many defects, respectively. Thus, the speed-up from no to many defects is 8.5 for KeY and 1.2 for JPF. The
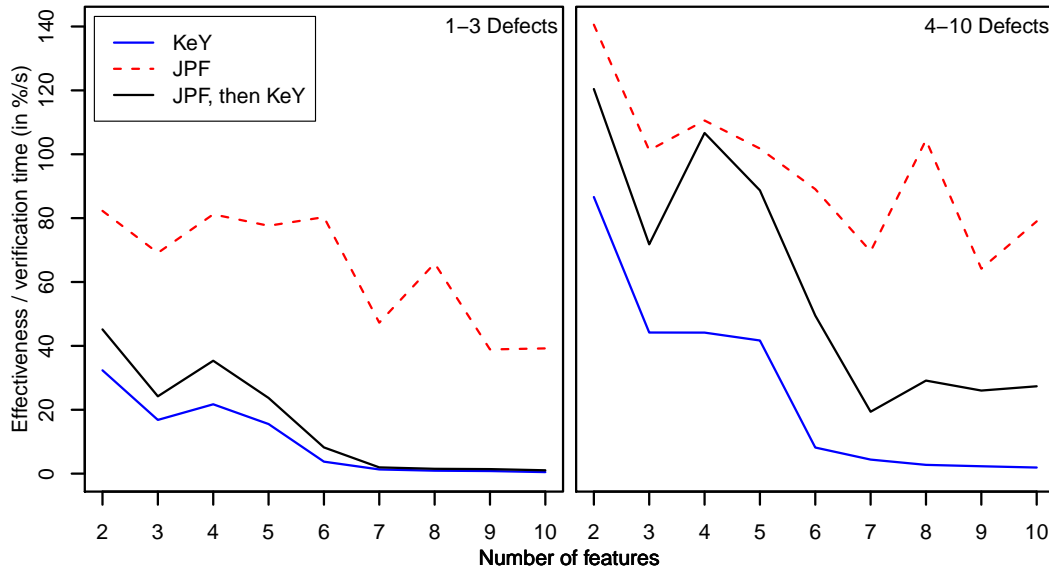
Figure 5.6: Efficiency as the ratio of effectiveness and performance (larger values are better) [Thüm et al., 2014].

better average performance is caused by the fact that both verifiers are aborted once they detect a defect. In contrast, larger verification times in some cases indicate that verifiers have, depending on the mutation, some extra effort. In those cases, JPF executes more statements and KeY needs to consider further proof rules than for the defect-free product line.

Efficiency

Based on the effectiveness and performance, we measured efficiency as the ratio of both. However, with our insight that JPF was faster than KeY, we propose to combine both verifier as follows: the product line is checked with JPF first and, depending on the result, KeY is only used if no defects were identified by JPF, resulting in the same effectiveness as always running both verifiers (i.e., "KeY or JPF" in Figure 5.4). Another reason for using JPF first is that it always indicates actual defects, compared to an open proof obligation in KeY. We show our results in Figure 5.6 (the new strategy is named Synergy), whereas larger values indicate higher efficiency. In the combination, KeY was utilized in 17.6 % of all runs and consumed 90.2 % of the verification time. However, KeY indicated additional defects in 71.3 % of those runs, in which JPF did not find a defect.

Based on the diagrams in Figure 5.6, we make the following observations. First, each verifier and the combination of both is more efficient for many defects than for some defects. The reason is that they are more effective and require less time. Second, JPF is more efficient for all sizes of product lines as well as some and many defects, because of the better performance. Third, the combination of both verifiers is less efficient than JPF but more efficient than KeY. This is an interesting result, as the combination

achieves a better effectiveness than each verifier individually, but the efficiency is still better than KeY. The reason is that we do only verify a product line with KeY, if no defects were found with JPF. In summary, combining KeY and JPF seems beneficial, as it increases the effectiveness compared to both in isolation and the efficiency compared to KeY.

Discussion

Besides our empirical evaluation, we discuss fundamental differences of theorem proving and model checking in the following, such as inherent guarantees, support for incomplete product lines, defect identification, and tool support.

Software model checking does not provide the same *guarantees* as theorem proving. With theorem proving, we verify methods for all given inputs, whereas software model checking is usually based on test scenarios, which are then executed symbolically. A test scenario is more than a simple test case, as it may consider a set of values at the same time. For example, method update in Listing 5.6 on Page 110 may be called with any positive integer value, which can be handled symbolically in JPF (with extensions dedicated to symbolic execution). In principle, we can write a test scenario exposing each possible defect, but defects are typically not known in advance. Test scenarios are often incomplete and it is up to the developer to write meaningful tests.

In principle, theorem proving enables verification with *incomplete implementations* and model checking with *incomplete specifications*. In theorem proving, we can verify methods separately by only relying on contracts of called methods. By doing this, we can even verify product lines that are not completely implemented. In contrast, model checking requires that for each test scenario all called methods are implemented, but they do not need to have contracts. Modular verification with theorem proving may be impractical, because contracts of all called methods must be strong enough, which comes with the price of high specification effort. In particular, some contracts in product line *BankAccount* are too weak, and thus we configured KeY to use the implementation of called methods (a.k.a. method inlining).

A further characteristic difference is how the programmer can *identify defects*. Using JPF, the result is a trace that is essentially a counter example. With KeY, we can inspect the proof, in general, and the unclosed goal, in particular. Both, the trace and the proof, can be large and hard-to-understand. However, an advantage of combining theorem proving and model checking is that we can use both to locate the defect. Nevertheless, a benefit of contracts in both cases is that the violated method contract is already identified, and we only need to identify whether the defect is in the specification or implementation.

A rather technical detail is that each JML *tool* implements a different set of keywords and a slightly different semantics, which requires some effort to use them in concert. For example, OpenJML reports invalid use of access modifiers (e.g., private, public) in contracts that KeY does not check. A further example is that OpenJML creates

runtime assertions reporting every violation of a precondition, whereas KeY does only check precondition violations if a method call is treated by applying its contract (e.g., not for method inlining). In addition, each tool usually only supports a certain subset of Java (e.g., OPENJML does not support threads and KeY does not support floating numbers).

Threats to Internal Validity

We measured the verification time for KeY and JPF until the first contract cannot be proven and the first assertion is thrown, respectively. While this time strongly depends on the order of test scenarios and proof order, we randomly generated a large number of mutations for each case to increase our confidence in the results. In general, an unproven contract in KeY does not necessarily mean that there is a defect; the theorem prover may need additional support, such as providing loop invariants [Barnett et al., 2011; Beckert et al., 2007; Burdy et al., 2005]. However, we inspected all unproven contracts for single mutations and they all indicated a defect.

The mutations that we applied to the product line may not be considered as defects in all cases. To avoid this problem, first, our implementation makes sure that each position in the code is only mutated once, to avoid that two mutations compensate each other. Second, some of the mutations cannot be detected with the given method contracts, as contracts typically do not encode the behavior completely. However, this is independent of the verification technique and should not influence our comparison.

The verification with JPF is influenced by the test scenarios, which are not required for KeY. Whether our test scenarios are meaningful for a comparison with KeY is questionable. As JPF with our test scenarios has a similar effectiveness as KeY indicates that our test scenarios are reasonable. Nevertheless, other test cases may lead to changes in effectiveness, performance, and efficiency of JPF.

The performance and efficiency of KeY could be better than measured in our experiment. The reason is that KeY can store proofs and check them after changes rather than finding a new proof. A common experience is that proof checking is magnitudes faster than proof finding [Beckert et al., 2007]. In our experiment, we have not used the ability to store proofs, because it is questionable how to simulate two subsequent versions of a product line with mutations. Simply taking the product line with and without mutations does not seem realistic and it is future work to incorporate proof checking into the comparison. For model checking, there are similar strategies that save effort during evolution [Strichman and Godlin, 2008].

Threats to External Validity

It is questionable to which extent our results can be generalized to larger product lines (i.e., more features and larger feature implementations). While experiments with larger product lines would be more valuable, already verifying product line *BankAccount* with several tools was a considerable effort. Furthermore, each verification tool does only

support a certain subset of Java and JML, which rules out many large product lines. Nevertheless, according to experience reported in Section 4.5, our subject product line including its contracts has typical characteristics (e.g., with respect to the mapping between features and classes). In addition, we simulate different sizes of product lines each bringing us to the same conclusions.

Our mutations may not represent real defects in product lines. As it was necessary for our evaluation to automatically generate defects into feature-oriented Java code and JML specifications, we decided to use mutation techniques. We used typical string replacement operators from mutation testing [Jia and Harman, 2011] to mutate feature modules and their contracts directly. More sophisticated operators operating on abstract syntax trees may provide more realistic defects, because a real defect may consist of not only slightly wrong parts, but also missing parts or wrong orders of statements. Furthermore, the generated mutations may not represent typical defects caused by feature interactions. The automatic generation of representative feature-interaction defects is non-trivial and should be investigated in future research.

Other verification tools for theorem proving and model checking may lead to different results, which should be evaluated in further studies. We have chosen KeY and JPF as both tools have been used by many other researchers before and in particular also for product-line verification [Apel et al., 2011, 2013d; Bruns et al., 2011; Kästner et al., 2012c].

Our comparison is based on fully automated verification, but in practice, both, theorem proving and model checking may depend on user input. Theorem proving may require to provide loop invariants or to guide the proof interactively. Besides creating test scenarios, model checking is automatic itself, but may require tuning of parameters to avoid the state explosion. Hence, further experiments are needed to assess the effort when evolving a product line.

5.3   Further Experiences

Besides proof composition and variability encoding, we explored further facets of the utilization of feature-oriented contracts for product-line verification. In the following, we give a brief overview on further challenges and discuss possible solutions, even if they have not yet been implemented and evaluated in-depth for various reasons. Nevertheless, we hope that their discussion provides a broader overview on research challenges of product-line verification with feature-oriented contracts. In particular, we discuss type checking for feature-oriented contracts in Section 5.3.1, feature-interaction detection in Section 5.3.2, and blame assignment in Section 5.3.3.

5.3.1   Type Safety of Feature-Oriented Contracts

With proof composition and variability encoding, we aim to verify that feature modules are correct with respect to feature-oriented contracts. Proof composition requires the generation of proof obligations for each product by means of a verification tool, such as

```
class Account {                                feature module BankAccount
   //@ invariant balance >= OVERDRAFT_LIMIT;
   int balance = 0;
}
```

```
class Account {                                   feature module Overdraft
   final static int OVERDRAFT_LIMIT = −5000;
}
```

```
class Account {                                             metaproduct
   //@ invariant balance >= OVERDRAFT_LIMIT;
   int balance = 0;
   final static int OVERDRAFT_LIMIT = −5000;
}
```

Listing 5.8: Example for a type error in a feature-oriented contract.

WHY. Besides generating proof obligations, each product is parsed and type checked in WHY. In contrast, variability encoding generates the metaproduct instead of each product. When verifying the metaproduct, verification tools, such as KeY, parse and type check the metaproduct, but it is not ensured that all products are type safe.

Example 5.4. *In Listing 5.8, we give an example for a type error that cannot be detected by type checking the metaproduct. For illustration purposes, we adapted the example in Listing 5.6 on Page 110. The invariant defined in feature modules* BankAccount *references a field of feature module* Overdraft*. When generating a product containing feature* BankAccount*, but not feature* Overdraft*, JML tools detect an undefined field access. With Java compilers, the problem cannot be detected, because it only concerns the type safety of JML contracts, which are written in comments. When verifying the metaproduct with a JML tool, this problem remains unnoticed, too, because all members are included into the metaproduct.*

As discussed in Section 5.2.1, a necessary step before applying analyses to the metaproduct is a type check of the product line. Efficient type checking of feature modules, in particular, and product-line implementations, in general, have been hot research topics and are well-understood (cf. Section 3.6). However, there is not a single approach for type checking product-line contracts, beyond product-based type checking. That is, we can generate all products and type check their contracts separately, as we do for proof composition.

An implementation of feature-based or family-based type checking of feature-oriented contracts seems to be desirable from a practical perspective. Such tool support can be achieved by either making existing type checkers for contracts variability-aware or by extending existing product-line type checkers with support for contracts. While this involves a large engineering effort in both cases, we do not expect many new insights from a research perspective. We implemented some analyses in FEATUREHOUSE and
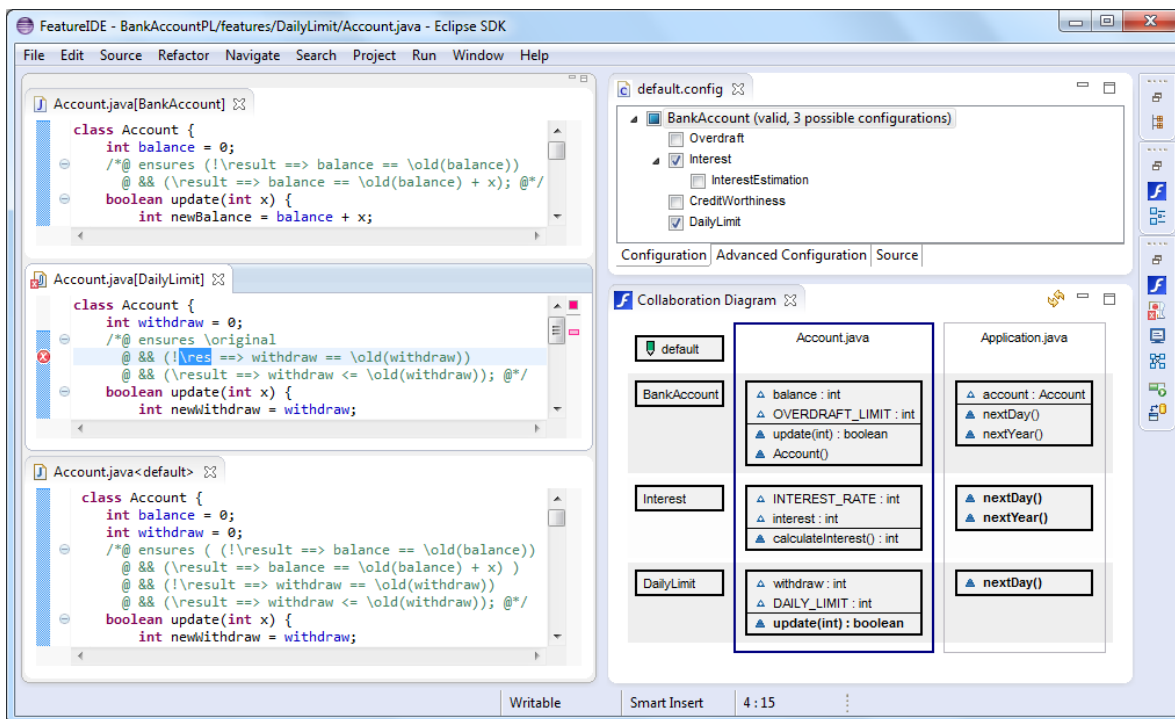
Figure 5.7: A syntax error in feature-oriented contracts detected by means of feature-based parsing.

FEATUREIDE that could be part of such a type checker for feature-oriented contracts. We give a brief overview on those analyses:

- With our FEATUREHOUSE extension, we parse each feature module including its feature-oriented contracts separately [Benduhn, 2012]. Thus, this can be considered as feature-based syntax checking. In principle, if all feature-oriented contracts are syntax-conform, also all generated products should not contain syntax errors. In Figure 5.7, the string \res is identified to be an invalid keyword and reported by FEATUREIDE using an error marker.

- In Section 4.4.1, we proposed the use of contract-composition keywords as an alternative to using the same contract composition for all contracts of a given product line. If contract-composition keywords are used even when FEATUREIDE is configured for using just one contract-composition mechanisms, then a warning is generated as the result of a feature-based analysis [Proksch and Krüger, 2014].

- We implemented several family-based analyses to detect wrong usage of contract-composition keywords [Proksch and Krüger, 2014]. First, if keyword final_method or keyword final_contract are followed by a method refinement or contract refinement, respectively, an error marker is created to emphasize the violation of the keyword. Second, we detect whether keyword original is used beyond explicit
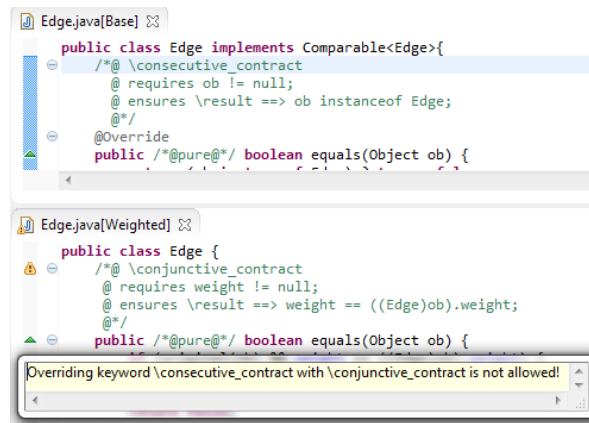
Figure 5.8: Warning in FeatureIDE for wrong overriding of contract-composition keywords [Weigelt, 2013].

contract refinement. Third, we warn users for prohibited keyword overriding according to Figure 4.4 on Page 76, and give an example for such a warning in Figure 5.8. In each case, the analysis problem is reduced to a satisfiability problem based on the FEATUREIDE infrastructure [Proksch and Krüger, 2014].

These analyses helped to identify some errors in our product lines with feature-oriented contracts. Nevertheless, we expect these product lines to contain more type errors in feature-oriented contracts that could be found by means of family-based type checking.

5.3.2 Static Analysis for Feature-Interaction Detection

Beyond theorem proving and model checking, feature-oriented contracts can also be checked by means of static analyses. As pointed out in Section 2.1.3, static analyses typically have a better scalability at the price of being *unsound or incomplete*. That is, a static analysis may miss defects and at the same time falsely signals defects. We experimented with the verifier ESC/JAVA2 [Cok and Kiniry, 2005], which is an extension of the extended static checker ESC/JAVA [Flanagan et al., 2002] with support for JML. ESC/JAVA2 takes a JML-annotated Java program as input and generates verification conditions that can be verified by an existing theorem prover. Our experiments are based on the automated theorem prover SIMPLIFY [Detlefs et al., 2005].

We verified product line *IntegerList* (cf. Appendix A) with deliberately introduced feature interactions [Scholz et al., 2011]. The product line consists of a core feature and four optional features giving rise to 16 products in total. None of these 16 products contains type errors, neither due to feature modules nor due to feature-oriented contracts (i.e., no syntactical feature interactions occur). In contrast, product line *IntegerList* has two *semantic feature interactions* that can only be found by means of a product-line specification. Each feature interaction affects four products. Due to the small size of the product line, we could anticipate these feature interactions in a manual inspection of the product-line implementation and specification.

We applied ESC/Java2 to all 16 products in a product-based fashion, as the subject product line is sufficiently small. With the default parameters of ESC/Java2, we found only one of two feature interactions. A manual inspection revealed that one interaction was not found due to the unsoundness of ESC/Java2. In particular, loops are verified by loop unrolling but the default value is that only the first loop run is checked. Changing the according parameter to check also the second run results in the detection of both feature interactions. However, changing the default values lead to a false alarm, too. This false alarm was due to the fact that some verification condition were too large for Simplify and could be avoided by providing further specifications (i.e., assert and assume statements). We refer interested readers to the original work for more details on the experiments [Scholz et al., 2011].

Our experiences with static analysis of feature-oriented contracts for feature-interaction detection are twofold. We were able to verify the product line with ESC/Java2 and, in the end, have not missed feature interactions and do not get false alarms. However, the problem with feature interactions is that they are typically unknown and then the question arises which parameters of ESC/Java2 should be used for verification. In contrast, software model checkers might miss feature interactions but have no false alarms and theorem proving might produce false alarms but do not miss feature interactions. It is an open challenge to compare the scalability and practicability of static analyses for product lines with that of model checking and theorem proving. Our initial experiences indicate that static analysis runs faster, but involves more effort for programmers to identify false alarms from actual feature interactions and defects.

5.3.3 Blame Assignment with Behavioral Feature Interfaces

One of the benefits of design by contract is blame assignment [Meyer, 1992]. That is, if the precondition of a method is not established, blame is assigned to the caller. Analogously, if the postcondition of a method is not established, blame is assigned to the callee. Hence, blame assignment is a means to locate defects in source code. In particular, blame assignment can identify the method that violated a given contract.

Feature-oriented contracts enable a kind of blame assignment for feature modules. If a method from one feature module calls a method from another feature module, blame assignment helps to locate the faulty feature module. However, not in all cases the faulty feature module can be identified, because method implementations may cross cut feature boundaries. If a method has several refinements, it is not sufficient to know that the method implementation is faulty. Instead, we also want to know the feature module that contributed a faulty method refinement. We propose *behavioral feature interfaces* [Thüm et al., 2013], which apply feature-oriented contracts also within refinement chains of methods, as we illustrate in the following example.

Example 5.5. *In Listing 5.9, we show three slightly modified feature modules of product line* BankAccount. *All three feature modules contribute to method* update *in class* Account. *We deliberately introduced a bug into one of these three feature modules. When composing all three feature modules into a product, we are likely to detect the*

```
class Account {                                    feature module BankAccount
  int balance = 0;
  /*@ requires balance + x >= 0;
    @ ensures balance == \old(balance) + x && \result == balance; @*/
  int update(int x) { balance += x; return balance; }
}
```

```
class Account {                                     feature module DailyLimit
  final int DAILY_LIMIT = −1000;
  int withdrawToday = 0;
  /*@ requires \original && withdrawToday + x >= DAILY_LIMIT;
    @ ensures \original; @*/
  int update(int x) { if (x < 0) withdrawToday += x; return original(−x); }
}
```

```
class Account {                                        feature module History
  int lastTransaction = 0;
  /*@ requires \original;
    @ ensures \original && lastTransaction == x; @*/
  int update(int x) { lastTransaction = x; return original(x); }
}
```

Listing 5.9: A defect in method update and its two refinements [Thüm et al., 2013].

defect by means of runtime assertions or verification. However, identifying the feature module with a defective contribution to method update *is hardly possible. With behavioral feature interfaces, we do not only compose contracts for method refinements, but we also generate contracts for each method introduction and method refinement separately, as illustrated in Listing 5.10. Not only the final method refinement is specified with a contract, but also all renamed methods, such as method* update$$BankAccount *and method* update$$DailyLimit. *For statement* new Account().update(42), *the first violation by means of a runtime assertion is found for statement* update$$BankAccount(-x), *which is originally defined in feature module* DailyLimit. *The method call violates the precondition* balance + x >= 0 *defined in feature module* BankAccount. *Consequently, we assign blame the method refinement* update *defined in feature module* DailyLimit, *in which statement* original(-x) *could be corrected to* original(x).*

The solution illustrated in the example above does not establish behavioral feature interfaces for class invariants, as discussed elsewhere in more detail [Thüm et al., 2013]. For short, class invariants apply only to previous method refinements, which can be achieved by creating an inheritance hierarchy for all class refinements as in Lightweight Feature Java [Delaware et al., 2009], FEATUREC++ [Apel et al., 2005], and MIXIN of the AHEAD tool suite [Batory, 2006]. In contrast to these formalizations and tools, methods being subject to later refinements need to be renamed. Otherwise, all method refinements would need to establish behavioral subtyping, which is too restrictive (cf. Section 4.5). An interesting side-effect of our application of contracts to feature mod-

```
class Account {                                        behavioral feature interfaces
  int balance = 0;
  final int DAILY_LIMIT = −1000;
  int withdrawToday = 0;
  /*@ requires balance + x >= 0;
   @ ensures balance == \old(balance) + x && \result == balance; @*/
  private int update$$BankAccount(int x) { balance += x; return balance; }
  /*@ requires balance + x >= 0 && withdrawToday + x >= DAILY_LIMIT;
   @ ensures balance == \old(balance) + x && \result == balance; @*/
  private int update$$DailyLimit(int x) { if (x < 0) withdrawToday += x;
    return update$$BankAccount(−x); }
  /*@ requires balance + x >= 0 && withdrawToday + x >= DAILY_LIMIT;
   @ ensures balance == \old(balance) + x && \result == balance &&
   @ lastTransaction == x; @*/
  int update(int x) { lastTransaction = x; return update$$DailyLimit(x); }
}
```

Listing 5.10: Blame assignment for method update with behavioral feature interfaces.

ules is that we identified semantic differences between different realizations of feature modules in products, which have been considered equivalent before [Thüm et al., 2013].

We illustrated behavioral feature interfaces for product generation, but they can also be established in the metaproduct. In Listing 5.7 on Page 111, we already showed contracts for renamed methods in Lines 5–6. Nevertheless, the same problem with class invariants and their visibility raises, and our solution for product generation [Thüm et al., 2013] may be applied metaproducts, too. In summary, to identify faulty feature modules, behavioral feature interfaces should be established for all analyses of feature-oriented contracts, such as proof composition and variability encoding.

5.4   Related Work

We already gave a comprehensive overview on product-line verification in Chapter 3. In the following, we discuss product-line verification by means of contracts in more detail.

Contracts for Product-Line Verification

Before we started to work on this thesis, contracts have not been used for product-line verification. Since then, several researchers investigated contracts for product-line verification, whereas all approaches are based on theorem proving. Bruns et al. [2011] and Hähnle et al. [2013] propose optimized product-based theorem proving based on KeY. Bruns et al. [2011] reduce the effort for products by means of slicing techniques, whereas Hähnle et al. [2013] split product verification into a first phase with abstract contracts and a second phase with concrete contracts. With proof composition, we presented another technique for proof reuse. Instead of searching for reuse potential

after product generation, we systematically compose partial proofs defined for each feature. Damiani et al. [2012] discuss feature-product-based theorem proving. The difference to proof composition is that features are verified by means of uninterpreted assertions. Proofs that could not be closed in the feature-based phase, are proven for each product separately. In contrast to proof composition, the product-based phase is likely to require user interaction. Hähnle and Schaefer [2012] propose feature-family-based theorem proving, in which features are verified in isolation and then their valid combinations are verified in a family-based fashion. The advantage of their approach over variability encoding is that part of the verification problem is already solved at feature level. The advantage of variability encoding, as presented in this thesis, is that features do not have to establish behavioral subtyping.

Product-Line Theorem Proving

Product-line theorem proving has its roots in type-soundness proofs for product lines of programming languages. Indeed, proof composition is inspired by feature-product-based theorem proving as proposed by Batory and Börger [2008]. They modularize the Java grammar, theorems on type soundness, and natural language proofs into feature modules. Nevertheless, a human needs to reason about all products when understanding the proofs. Later, Delaware et al. [2009, 2011, 2013] verified type-soundness proofs with the proof assistant Coq in a feature-product-based manner. The difference to proof composition is that they manually modularize theorems, while we generate theorems with the program verification framework Why. Furthermore, the product-based phase of proof composition does not require any user interaction, whereas theorems and proofs have to be manually assembled and extended for the type-soundness proofs.

Product-Line Model Checking

As discussed in Chapter 3, the majority of approaches for product-line model checking are applied to an abstraction of the product line, whereas we apply software model checking to the implementation of a product line. All existing approaches to model check a product-line implementation are family-based and rely on variability encoding [Apel et al., 2011, 2013d; Kästner et al., 2012c; Post and Sinz, 2008], as we do. By means of variability encoding, product lines are verified with existing model checkers from single-system engineering, such as CBMC [Post and Sinz, 2008], CPAchecker [Apel et al., 2011, 2013d], and JPF [Apel et al., 2013d; Kästner et al., 2012c]. The main difference to our work is that we extend variability encoding to specifications (i.e., contracts), and thus are the first to apply model checking on product-line contracts. Furthermore, we analyze the effect of defects on the effectiveness and efficiency of product-line model checking and theorem proving for the first time.

Product-Line Type Checking

An assumption of variability encoding is that the product line is type safe. In Chapter 3, we give a detailed overview on product-line type checking. While several strategies have been applied to composition-based and annotation-based product-line implementations and designs, none of these approaches applies type checking to specifications. We discussed type checking for feature-oriented contracts and sketched possible approaches.

Combination of Verification Techniques

Other researchers propose a combination of verification techniques. Liebig et al. [2013] combine type checking and dataflow analysis to find defects in Linux. However, both techniques are used for different kinds of errors, whereas we focus on synergies for the same kind of error. Others use Event-B for product-line verification, which has support for theorem proving and model checking [Gondal et al., 2011; Poppleton, 2007, 2008; Sorge et al., 2010]. However, they do neither discuss nor evaluate the benefit of that combination. Besides product lines, several combinations of theorem proving and model checking have been proposed [Abrial, 2010; Dybjer et al., 2004; Halpern and Vardi, 1991; Ismail et al., 2013; Owre et al., 1996], but they all inherently require to create new or change existing verifiers, whereas, with variability encoding, we used each verifier as-is. Nevertheless, more sophisticated combinations of theorem proving and model checking could be considered for product lines and single systems in future work.

5.5   Summary

We explored several analysis strategies and verification techniques for the verification of product lines based on feature-oriented contracts. With proof composition and variability encoding, we propose two alternative techniques for proof reuse, which promote the underrepresented research area of product-line theorem proving. In proof composition, we are the first to apply the concept of proof-carrying code to feature modules. Composing proofs together with source code and specification can drastically reduce the effort of proof writing. Nevertheless, redundant effort for proof checking is necessary in our feature-product-based theorem proving.

We used variability encoding to overcome redundant proof checking and to combine several verification techniques to the very same properties. One interesting outcome is that the combination of theorem proving and model checking is more effective and efficient than using each verification technique in isolation. Furthermore, product-line theorem proving and model checking are both more effective and efficient if the product line contains more defects.

Beyond proof composition and variability encoding, we discussed the role of other verification techniques for feature-oriented contracts and gained several insights. First, existing type checking approaches for product lines need and can be extended to type check feature-oriented contracts. Second, static analyses for feature-oriented contracts seem to scale better than theorem proving and model checking, but suffer from incompleteness and unsoundness. Finally, each verification approach for feature-oriented contracts should establish behavioral feature interfaces to lift blame assignment from classes to feature modules.

6. Conclusion and Future Work

In the following, we conclude our thesis and briefly discuss potential future work on product-line specification and product-line verification.

Conclusion

The success of software-product-line engineering does not only depend on sophisticated techniques for code generation, but also on efficient and effective analysis strategies. As product lines are increasingly used for mission-critical and safety-critical systems, quality assurance becomes indispensable. Whereas existing work on product-line analysis stems from various communities, we propose a common terminology and use it to classify approaches for product-line verification according to their strategy to deal with variability in implementation, specification, and analysis. An interesting insight with our classification and survey is that many authors claim to propose compositional approaches, whereas we distinguish between compositional implementation, compositional specification, and three strategies for compositional analysis, namely feature-based, feature-product-based, and feature-family-based analysis. These strategies indicate how approaches handle feature interactions, which are inherently non-compositional.

In our survey, we have recognized that existing specification techniques for product lines are just used as proof-of-concept for verification techniques and have not been justified theoretically and empirically. We fill this gap, by a chapter dedicated to the specification of product lines, in which we systematically discuss and empirically evaluate how to specify software product lines by means of feature-oriented contracts. While our discussion and evaluation is based on contracts and feature-oriented programming, we are confident that the results can be transferred to other specification and implementation techniques. Our main insight is that behavioral subtyping applies to most feature-oriented method refinements, but not all. The consequence is that existing product-line verification approaches based on behavioral subtyping are not applicable to the majority of product lines. With feature-oriented contracts, we provide the flexibility to decide for each method refinement whether it is supposed to establish behavioral subtyping.

Based on feature-oriented contracts, we explored approaches for product-line verification with theorem proving. We propose proof composition and variability encoding for proof reuse. Proof composition only reduces effort for proof writing, while variability encoding reduces effort for proof checking additionally. In both cases, our evaluation revealed that verification effort is drastically reduced by means of proof reuse. Furthermore, we combined proof reuse with software model checking and measured synergistic effects, such as improved efficiency and effectiveness. Our evaluation also revealed that theorem proving and model checking for product lines perform better for earlier development stages (i.e., if the product line contains more defects).

Future Work

One outcome of our classification and survey of product-line analyses is a research agenda (cf. Section 3.6), which may guide future work for the analysis of product lines. For instance, we propose to create and evaluate approaches for combinations of strategies and verification techniques that have not been considered before. As most proposals for product-line analysis neglect evolution, we plan to investigate approaches pursuing a feature-family-based strategy, because it could be superior to other strategies for evolving product lines.

We discussed and evaluated feature-oriented contracts for feature-oriented programming only, such that it is natural to evaluate how our results transfer to other product-line implementation techniques, namely preprocessors, plug-ins, and aspect-oriented programming. Beyond product lines, different contract composition mechanisms could even be beneficial for object-oriented programming and mixins, especially as behavioral subtyping is desirable, but does often not apply to legacy software. Our considerations of feature-oriented contracts were focused on basic constructs of behavioral interfaces specification languages, whereas an extension to framing conditions, visibility levels, and exceptional statements is required for a practical use.

With variability encoding, we verified product lines according to feature-oriented contracts by transforming compile-time variability into runtime variability. Besides some informal argumentation and practical evaluations, there is no proof that the transformation is correct. Such a proof should be done in future work and requires a formalization of feature model, feature modules, and feature-oriented contracts, as well as a formalization of product generation and variability encoding. Besides the correctness proof, it seems beneficial to combine feature-based analyses with variability encoding resulting in a feature-family-based strategy. Furthermore, new verification approaches should be investigated to exploit that some feature-oriented contracts establish behavioral subtyping, even if not all of them.

Besides these research challenges, we hope that our engineering effort to provide tool support for feature-oriented contracts in FEATUREIDE supports technology transfer. FEATUREIDE is already used in lectures at several universities, and thus feature-oriented contracts could be added to lectures easily. Furthermore, we envision the evaluation of feature-oriented contracts in practice.

A. Appendix

The product lines that we used to evaluate feature-oriented contracts for product-line specification and verification all have different characteristics. We give an overview on statistics with respect to the feature model and implementation in Table A.1. Furthermore, we present several statistics on feature-oriented contracts in Table A.2 and on their composition in Table A.3. As discussed in Section 4.5, some of the product lines have been development from scratch, while others are the result of decomposing an already specified program or by specifying an existing product line. We give a short description for each product line describing the basic functionality and how it has been created. Then, we give some details on the statistics. All product lines are publicly available as examples within FEATUREIDE and at the website SPL2GO.[1]

Product Lines and Their Specifications Developed from Scratch

1. *BankAccount*: A feature-oriented system for bank-account management that models basic concepts such as accounts and users, as well as features for limiting withdrawals and for calculating interests and credit worthiness. In 2010, I have created the product line with five features for the purpose of evaluating proof composition and I verified it using CoQ [Thüm et al., 2011b]. In 2012, I extended the product line by a further feature and verified it using KeY with the help of Martin Hentschel and Richard Bubel [Thüm et al., 2012]. Finally, Jens Meinicke extended the product line by two further features and verified it using KeY and JPF [Meinicke, 2013; Thüm et al., 2014]. The feature model of the last version is shown in Figure 5.2 on Page 108. Excerpts of the feature modules are shown in Listing 5.1 on Page 99, Listing 5.6 on Page 110, Figure 5.7 on Page 124, and Listing 5.9 on Page 127.

2. *GPL-scratch*: The graph product line has been proposed as a standard problem for the evaluation of product-line techniques, and provides variability in data

---

	Products	Features	Core features	Alternative features	Derivatives	Classes	Fields	Methods	Roles	Alternative method introductions	Method refinements	Pure methods	Alternative pure-method introductions	Pure-method refinements
BankAccount	72	8	1	0	0	3	8	13	11	0	5	2	0	0
GPL-scratch	128	10	1	2	0	4	13	42	19	6	7	12	2	6
IntegerList	16	5	1	0	0	2	2	9	8	0	4	0	0	0
UnionFind	6	8	2	5	0	4	8	19	10	0	10	2	0	4
StringMatcher	6	8	2	6	0	2	1	3	9	0	6	0	0	0
DiGraph	8	4	1	0	0	8	12	66	11	0	0	26	0	0
ExamDB	8	8	1	0	4	4	10	50	20	0	29	9	0	0
IntegerSet	2	3	1	2	0	1	7	15	3	4	6	6	0	2
Numbers	2	2	1	0	0	1	0	10	2	0	7	0	0	0
Paycard	6	4	1	0	0	7	42	25	10	0	3	6	0	1
Poker	21	10	3	3	0	8	34	56	16	3	5	15	0	0
Elevator	20	6	1	0	0	11	38	92	19	0	19	8	0	0
Email	40	9	1	0	0	3	23	55	14	0	16	9	0	1
GPL	156	27	2	24	10	21	46	104	87	141	53	17	35	1

Table A.1: Statistics on feature model and implementation of all product lines.

structures and algorithms [Lopez-Herrejon and Batory, 2001]. In this spirit, André Weigelt developed a product line of graph libraries for his bachelor's thesis [Weigelt, 2013], in which edges are directed or undirected and weighted or unweighted. Furthermore, the shortest or weight-optimal path between two given nodes can be calculated. Parts of the feature-oriented contracts and feature modules are shown in Listing 4.2 on Page 58, Listing 4.3 on Page 60, Listing 4.4 on Page 63, and Listing 4.5 on Page 67.

3. *IntegerList*: A feature-oriented implementation of a list structure for storing integer values and corresponding basic operations. Based on the feature selection, the list behaves like a stack or a sorted list. Wolfgang Scholz developed this product line to evaluate the feasibility of feature-oriented contracts for feature-interaction detection [Scholz et al., 2011]. In Listing 4.1 on Page 56, we used an excerpt of this product line to illustrate plain contracting.

4. *UnionFind*: A feature-oriented system representing variations of the union-find algorithm [Sedgewick, 1983], which can calculate the union of sets, check whether two values are in the same component, and calculate the number of components. Fabian Benduhn developed this product line in the course of his bachelor's thesis [Benduhn, 2012].

	Contracts	Invariants	Core contracts	Core invariants	Alternative contract introductions	Alternative invariant introductions	Contract refinements	Invariant refinements	Explicit contract refinement	Contract overriding	Conjunctive contract refinement	Cumulative contract refinement	Consecutive contract refinement
BankAccount	18	3	4	2	0	0	5	0	5	2	5	5	5
GPL-scratch	42	0	16	0	6	0	7	0	5	0	4	0	2
IntegerList	7	3	1	2	0	0	1	0	1	0	1	1	1
UnionFind	7	0	4	0	0	0	2	0	2	0	2	2	2
StringMatcher	7	0	1	0	0	0	6	0	6	0	4	0	0
DiGraph	45	9	26	9	0	0	0	0	0	0	0	0	0
ExamDB	40	5	17	4	0	0	15	0	14	14	0	0	1
IntegerSet	12	7	3	0	2	0	0	0	0	0	0	0	0
Numbers	15	0	7	0	0	0	6	0	5	1	5	5	5
Paycard	10	6	4	2	0	0	1	0	0	0	0	0	1
Poker	48	6	20	2	0	0	2	0	2	0	2	1	0
Elevator	13	0	0	0	0	0	6	0	6	0	6	6	6
Email	7	0	0	0	0	0	3	0	3	0	3	1	0
GPL	110	0	2	0	69	0	6	0	6	5	5	5	5

Table A.2: Statistics on contracts and invariants of all product lines.

5. *StringMatcher*: A product line of string comparison algorithms, whereas each variant considers other pairs of strings as equivalent (e.g., if they have the same length or if one is a substring of the other). Fabian Benduhn developed this product line for his bachelor's thesis [Benduhn, 2012] and verified it later using KeY and JPF.

Decomposition of Existing Programs with Specifications

1. *DiGraph*: A library for representing and manipulating directed graph structures. Beside basic graphs, it supports various operations such as removal, traversal, and transposition. The library has been developed and specified by Albert Baker, Katie Becker, and Gary T. Leavens, and published as a JML example.[2] In 2011, I identified and modularized three optional features, resulting in a product line with eight products.

2. *ExamDB*: An exam database management system that manages exams to be passed by students, including features for subscription and backouting, bonus points, and statistics. Timo Eifler implemented and specified *ExamDB*, verified it with KeY, and published it in the VERIFYTHIS repository.[3] In 2011, I decomposed *ExamDB* into a product line with four features and eight products. As the

---

[2]http://www.eecs.ucf.edu/~leavens/JML-release/org/jmlspecs/samples/digraph/
[3]http://www.verifythis.org/post?pid=database-system-for-managing-exams

	Original-caller-preserving	Original-callee-preserving	Refinement-caller-preserving	Refinement-callee-preserving	No primitive preservation	Original-preserving	Refinement-preserving	Caller-preserving	Callee-preserving	No combound preservation	Precondition and postcondition	Only precondition	Only postcondition	Part of a condition	Identical contract
BankAccount	5	0	5	2	0	0	2	5	0	0	0	0	5	0	0
GPL-scratch	2	2	2	2	3	0	0	2	2	3	6	1	0	2	0
IntegerList	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0
UnionFind	2	0	2	0	0	0	0	2	0	0	0	0	2	0	0
StringMatcher	4	0	4	0	2	0	0	4	0	2	0	0	6	2	0
DiGraph	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ExamDB	1	0	1	0	14	0	0	1	0	14	0	1	14	15	0
IntegerSet	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Numbers	5	0	5	0	1	0	0	5	0	1	0	0	6	1	0
Paycard	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0
Poker	1	1	0	1	0	0	0	0	1	1	0	1	1	0	0
Elevator	6	0	6	0	0	0	0	6	0	0	0	0	6	0	0
Email	1	2	0	1	0	0	0	0	1	2	0	2	1	0	0
GPL	5	3	6	3	0	2	3	5	3	0	0	0	4	0	2

Table A.3: Statistics on contract refinements of all product lines.

chosen features required fine-granular refinements of source code and contracts and to avoid code and specification clones, I slightly modified the design by introducing pure methods and pure-method refinements (cf. Listing 4.6 on Page 71). In 2014, I created a different version of *ExamDB* without pure-method refinement, which required to introduce derivatives (cf. Figure 4.7 on Page 82 and Listing 4.7 on Page 85). Both version are available as FEATUREIDE examples and if not specified explicitly, we refer to the second version in this thesis.

3. *IntegerSet*: A representation of sets to insert and remove values, and to check element containment. A special characteristic is the use of alternative implementations based on trees or hashing. The systems has been implemented and specified by Katie Becker, Arthur Thomas, and Gary T. Leavens.[4] Fabian Benduhn decomposed the system into a base implementation and two features for trees and hashing [Benduhn, 2012].

4. *Numbers*: As an example for JML, Gary T. Leavens specified operations, such as multiplication and division, on complex numbers inspired by the book of Abelson

---

[4] http://www.eecs.ucf.edu/~leavens/JML-release/org/jmlspecs/samples/sets/

and Sussman [1996].[5] Fabian Benduhn decomposed implementation and specification into imaginary and real parts in his bachelor's thesis [Benduhn, 2012].

5. *Paycard*: An implementation of a paycard, which can be charged by the customer with a certain amount of money for cashless payment afterward. The implementation is partitioned into a core implementation and a graphical user interface to simulate transactions. The paycard has a limit of 1,000, 100, or a user-defined limit to which charging is allowed. *Paycard* is an example being used in the KeY tutorial.[6] In 2011, I decomposed the source code and contracts into four feature modules: a base implementation, logging of transactions, lock out after a number of unsuccessful implementations, and access of the largest transaction.

6. *Poker*: An implementation of a poker game specified in JML,[7] which has been decomposed into several feature modules by Fabian Benduhn [Benduhn, 2012]. The features include changing the maximum number of cards a player can have, whether players can bet money while playing, introducing a limit for betting money, or changing the blinds. Figure 4.6 on Page 81 gives an overview on the methods defined in core features.

Specification of Existing Product Lines

1. *Elevator*: Plath and Ryan [2001] developed a model and informal specification of an elevator that has optional features, such as a priority mode for a special floor or ignoring buttons inside the elevator when it is empty. Alexander von Rhein implemented the product line in Java and encoded specifications by means of AspectJ.[8] The product line has been verified with JPF, except for some detected feature interactions [Apel et al., 2013c,d]. In 2013, I encoded the specifications given in AspectJ into feature-oriented contracts.

2. *Email*: Hall modeled an email system providing optional features such as encryption, forwarding, and signatures [Hall, 2005]. He provided informal and formal specifications for feature-interaction detection. Alexander von Rhein, Stefan Boxleitner, and Hendrik Speidel implemented the email system in Java with specifications in AspectJ.[9] Similarly to product line *Elevator*, it has been verified with JPF [Apel et al., 2013c,d] and I encoded AspectJ specifications as feature-oriented contracts in 2013 (cf. Listing 4.8 on Page 89).

3. *GPL*: As written above, the graph product line has been proposed as a benchmark for product-line techniques [Lopez-Herrejon and Batory, 2001], which we used a

---

[5] http://www.eecs.ucf.edu/~leavens/JML-release/org/jmlspecs/samples/dbc/
[6] http://www.key-project.org/download/quicktour/
[7] https://github.com/topless/PokerTop
[8] http://spl2go.cs.ovgu.de/projects/16
[9] http://spl2go.cs.ovgu.de/projects/17

further time for evaluation of feature-oriented contracts. In contrast to *GPL-scratch* and *DiGraph*, an existing implementation[10] of the graph product line was specified with feature-oriented contracts. In particular, Fabian Benduhn specified *GPL* by guessing contracts for the existing design in the course of his bachelor's thesis [Benduhn, 2012].

Comments on the Statistics

Although most of the statistics presented in the tables above should be self-explaining in the context of this thesis, we want to emphasize some peculiarities. In Table A.1, we only count concrete features, as abstract features do have not influence on the decomposition of contracts [Thüm et al., 2011a]. In particular, we present the number of actually different products and not the number of possible configurations [Thüm et al., 2011a]. For simplicity, we do not explicitly count constructors, but count them as methods instead. The column *Methods* only refers to unique methods (i.e., if a method is defined in two alternative features, it is only counted once). Whether a certain method defined in a feature module is a method refinement or a method introduction, can vary from product to product. The reason is that all previous method introductions can be optional. We decided to count a method as method refinement, if it refines a method in at least one product, and else as method introduction.

In Table A.2, we give the number of contracts for each product line, whereas solely the keyword pure does not count as contract (i.e., only if at least a precondition or postcondition is defined). As discussed in Section 4.2, a special case occurs if a method without a contract is refined by a method enriched with a contract *c*. In such a case, we do not consider the contract *c* as a contract refinement, but rather as a method introduction. Otherwise, contract-refinement mechanisms pursuing behavioral subtyping, such as conjunctive contract refinement, could not be used in most of these cases. In particular, refining of the default precondition requires true would not be possible (i.e., restricting the possible method input). Finally, in Table A.3, refinement of a "part of a condition" refers to all refinements beyond adding something in disjunction or conjunction (e.g., refining a part of a condition while the rest remains identical).

---

[10]http://www.infosun.fim.uni-passau.de/spl/apel/fh/#download

Bibliography

Abal, I., Brabrand, C., and Wasowski, A. (2014). 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 421–432, New York, NY, USA. ACM.

Abelson, H. and Sussman, G. J. (1996). *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition.

Abrial, J.-R. (2010). *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition.

Adelsberger, S., Sobernig, S., and Neumann, G. (2014). Towards Assessing the Complexity of Object Migration in Dynamic, Feature-Oriented Software Product Lines. In *Proc. Int'l Workshop Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 17:1–17:8, New York, NY, USA. ACM.

Agostinho, S., Moreira, A., and Guerreiro, P. (2008). Contracts for Aspect-Oriented Design. In *Proc. Workshop Software Engineering Properties of Languages and Aspect Technologies (SPLAT)*, pages 1:1–1:6, New York, NY, USA. ACM.

Ahrendt, W., Beckert, B., Bruns, D., Bubel, R., Gladisch, C., Grebing, S., Hähnle, R., Hentschel, M., Klebanov, V., Mostowski, W., Scheben, C., Schmitt, P. H., and Ulbrich, M. (2014). The KeY Platform for Verification and Analysis of Java Programs. In Giannakopoulou, D. and Kroening, D., editors, *Proc. IFIP Working Conf. Verified Software: Theories, Tools, Experiments (VSTTE)*, pages 55–71, Berlin, Heidelberg. Springer.

Alférez, M., Lopez-Herrejon, R. E., Moreira, A., Amaral, V., and Egyed, A. (2011). Supporting Consistency Checking Between Features and Software Product Line Use Scenarios. In *Proc. Int'l Conf. Software Reuse (ICSR)*, pages 20–35, Berlin, Heidelberg. Springer.

America, P. (1991). Designing an Object-Oriented Programming Language with Behavioural Subtyping. In *Proc. Int'l Workshop Foundations of Object-Oriented Languages (FOOL)*, pages 60–90, London, UK. Springer.

Antkiewicz, M., Ji, W., Berger, T., Czarnecki, K., Schmorleiz, T., Lämmel, R., Stănciulescu, t., Wąsowski, A., and Schaefer, I. (2014). Flexible Product Line Engineering

with a Virtual Platform. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 532–535, New York, NY, USA. ACM.

Apel, S., Batory, D., Kästner, C., and Saake, G. (2013a). *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer, Berlin, Heidelberg.

Apel, S. and Hutchins, D. (2010). A Calculus for Uniform Feature Composition. *ACM Trans. Programming Languages and Systems (TOPLAS)*, 32(5):19:1–19:33.

Apel, S., Kästner, C., Größlinger, A., and Lengauer, C. (2010a). Type Safety for Feature-Oriented Product Lines. *Automated Software Engineering*, 17(3):251–300.

Apel, S., Kästner, C., and Lengauer, C. (2008a). Feature Featherweight Java: A Calculus for Feature-Oriented Programming and Stepwise Refinement. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 101–112, New York, NY, USA. ACM.

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

Apel, S., Leich, T., Rosenmüller, M., and Saake, G. (2005). FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 125–140, Berlin, Heidelberg. Springer.

Apel, S., Leich, T., and Saake, G. (2008b). Aspectual Feature Modules. *IEEE Trans. Software Engineering (TSE)*, 34(2):162–180.

Apel, S., Lengauer, C., Möller, B., and Kästner, C. (2010b). An Algebraic Foundation for Automatic Feature-Based Program Synthesis. *Science of Computer Programming (SCP)*, 75(11):1022–1047.

Apel, S., Scholz, W., Lengauer, C., and Kästner, C. (2010c). Detecting Dependences and Interactions in Feature-Oriented Design. In *Proc. Int'l Symposium Software Reliability Engineering (ISSRE)*, pages 161–170, Washington, DC, USA. IEEE.

Apel, S., Scholz, W., Lengauer, C., and Kästner, C. (2010d). Language-Independent Reference Checking in Software Product Lines. In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD)*, pages 65–71, New York, NY, USA. ACM.

Apel, S., Speidel, H., Wendler, P., von Rhein, A., and Beyer, D. (2011). Detection of Feature Interactions Using Feature-Aware Verification. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 372–375, Washington, DC, USA. IEEE.

Apel, S., von Rhein, A., Thüm, T., and Kästner, C. (2013c). Feature-Interaction Detection Based on Feature-Based Specifications. *Computer Networks*, 57(12):2399–2409.

Apel, S., von Rhein, A., Wendler, P., Größlinger, A., and Beyer, D. (2013d). Strategies for Product-Line Verification: Case Studies and Experiments. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 482–491, Piscataway, NJ, USA. IEEE.

Asirelli, P., ter Beek, M. H., Fantechi, A., and Gnesi, S. (2012). A Compositional Framework to Derive Product Line Behavioural Descriptions. In *Proc. Int'l Symposium Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 146–161, Berlin, Heidelberg. Springer.

Atkinson, C. and Kühne, T. (2003). Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41.

Aversano, L., Penta, M. D., and Baxter, I. D. (2002). Handling Preprocessor-Conditioned Declarations. In *Proc. Int'l Working Conference Source Code Analysis and Manipulation (SCAM)*, pages 83–92, Washington, DC, USA. IEEE.

Balzer, S., Eugster, P. T., and Meyer, B. (2006). Can Aspects Implement Contracts? In *Proc. Int'l Conf. Rapid Integration of Software Engineering Techniques (RISE)*, pages 145–157, Berlin, Heidelberg. Springer.

Barnett, M., Fähndrich, M., Leino, K. R. M., Müller, P., Schulte, W., and Venter, H. (2011). Specification and Verification: The Spec# Experience. *Comm. ACM*, 54:81–91.

Bashardoust-Tajali, S. and Corriveau, J.-P. (2008). On Extracting Tests from a Testable Model in the Context of Domain Engineering. In *Proc. Int'l Conf. Engineering of Complex Computer Systems (ICECCS)*, pages 98–107, Washington, DC, USA. IEEE.

Bass, L., Clements, P., and Kazman, R. (1998). *Software Architecture in Practice*. Addison-Wesley, Boston, MA, USA.

Batory, D. (2005). Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 7–20, Berlin, Heidelberg. Springer.

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

Batory, D. and Börger, E. (2008). Modularizing Theorems for Software Product Lines: The Jbook Case Study. *J. Universal Computer Science (J.UCS)*, 14(12):2059–2082.

Batory, D., Cardone, R., and Smaragdakis, Y. (2000). Object-Oriented Frameworks and Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 227–247, Norwell, MA, USA. Kluwer Academic Publishers.

Batory, D., Sarvela, J. N., and Rauschmayer, A. (2004). Scaling Step-Wise Refinement. *IEEE Trans. Software Engineering (TSE)*, 30(6):355–371.

Beckert, B. and Hähnle, R. (2014). Reasoning and Verification: State of the Art and Current Trends. *IEEE Intelligent Systems*, 29(1):20–29.

Beckert, B., Hähnle, R., and Schmitt, P. (2007). *Verification of Object-Oriented Software: The KeY Approach*. Springer, Berlin, Heidelberg.

Benavides, D., Segura, S., and Ruiz-Cortés, A. (2010). Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–708.

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

Benduhn, F. (2014). Representing Variability in Product Lines: A Survey of Modeling and Specification Techniques. Master's thesis, University of Magdeburg, Germany.

Bertot, Y. and Castéran, P. (2004). *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer, Berlin, Heidelberg.

Bessling, S. and Huhn, M. (2014). Towards Formal Safety Analysis in Feature-Oriented Product Line Development. In *Proc. Int'l Symposium Foundations of Health Information Engineering and Systems (FHIES)*, pages 217–235, Berlin, Heidelberg. Springer.

Bettini, L., Damiani, F., and Schaefer, I. (2013). Compositional Type Checking of Delta-Oriented Software Product Lines. *Acta Informatica*, 50(2):77–122.

Bettini, L., Damiani, F., and Schaefer, I. (2015). Implementing Type-Safe Software Product Lines Using Parametric Traits. *Science of Computer Programming (SCP)*, 97(3):282–308.

Beyer, D., Henzinger, T. A., Jhala, R., and Majumdar, R. (2007). The Software Model Checker Blast: Applications to Software Engineering. *Int'l J. Software Tools for Technology Transfer (STTT)*, 9(5):505–525.

Beyer, D. and Keremoglu, M. E. (2011). CPAchecker: A Tool for Configurable Software Verification. In *Proc. Int'l Conf. Computer Aided Verification (CAV)*, pages 184–190, Berlin, Heidelberg. Springer.

Blundell, C., Fisler, K., Krishnamurthi, S., and Hentenryck, P. V. (2004). Parameterized Interfaces for Open System Verification of Product Lines. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 258–267, Washington, DC, USA. IEEE.

Bodden, E., Tolêdo, T., Ribeiro, M., Brabrand, C., Borba, P., and Mezini, M. (2013). SPLLIFT: Statically Analyzing Software Product Lines in Minutes Instead of Years. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, pages 355–364, New York, NY, USA. ACM.

Börger, E. and Stark, R. F. (2003). *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, Secaucus, NJ, USA.

Bowen, T. F., Dworack, F. S., Chow, C.-H., Griffeth, N., Herman, G. E., and Lin, Y.-J. (1989). The Feature Interaction Problem in Telecommunications Systems. In *Proc. Int'l Conf. Software Engineering for Telecommunication Switching Systems (SETSS)*, pages 59–62, Washington, DC, USA. IEEE.

Brabrand, C., Ribeiro, M., Tolêdo, T., Winther, J., and Borba, P. (2013). Intraprocedural Dataflow Analysis for Software Product Lines. *Trans. Aspect-Oriented Software Development*, 10:73–108.

Bracha, G. and Cook, W. (1990). Mixin-Based Inheritance. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 303–311, New York, NY, USA. ACM.

Bruns, D., Klebanov, V., and Schaefer, I. (2011). Verification of Software Product Lines with Delta-Oriented Slicing. In *Proc. Int'l Conf. Formal Verification of Object-Oriented Software (FoVeOOS)*, pages 61–75, Berlin, Heidelberg. Springer.

Bubel, R., Din, C., and Hähnle, R. (2010). Verification of Variable Software: An Experience Report. In *Proc. Int'l Conf. Formal Verification of Object-Oriented Software (FoVeOOS)*, Karlsruhe, Germany. Technical Report 2010-13, Department of Informatics, Karlsruhe Institute of Technology.

Bubel, R., Hähnle, R., and Pelevina, M. (2014). Fully Abstract Operation Contracts. In Margaria, T. and Steffen, B., editors, *Proc. Int'l Symposium Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 120–134, Berlin, Heidelberg. Springer.

Buchmann, T. and Schwägerl, F. (2012). Ensuring Well-Formedness of Configured Domain Models in Model-Driven Product Lines Based on Negative Variability. In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD)*, pages 37–44, New York, NY, USA. ACM.

Burdy, L., Cheon, Y., Cok, D. R., Ernst, M. D., Kiniry, J., Leavens, G. T., Leino, K. R. M., and Poll, E. (2005). An Overview of JML Tools and Applications. *Int'l J. Software Tools for Technology Transfer (STTT)*, 7(3):212–232.

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

Carmo Machado, I. D., McGregor, J. D., Cavalcanti, Y. a. C., and De Almeida, E. S. (2014). On Strategies for Testing Software Product Lines: A Systematic Literature Review. *J. Information and Software Technology (IST)*, 56(10):1183–1199.

Chalin, P., Kiniry, J., Leavens, G. T., and Poll, E. (2005). Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In *Proc. Int'l Symposium Formal Methods for Components and Objects (FMCO)*, pages 342–363, Berlin, Heidelberg. Springer.

Chen, S. and Erwig, M. (2014). Type-Based Parametric Analysis of Program Families. In *Proc. Int'l Conf. Functional Programming (ICFP)*, pages 39–51, New York, NY, USA. ACM.

Chen, S., Erwig, M., and Walkingshaw, E. (2014). Extending Type Inference to Variational Programs. *ACM Trans. Programming Languages and Systems (TOPLAS)*, 36(1):1:1–1:54.

Cimatti, A., Clarke, E. M., Giunchiglia, F., and Roveri, M. (1999). NuSMV: A New Symbolic Model Verifier. In *Proc. Int'l Conf. Computer Aided Verification (CAV)*, pages 495–499, London, UK. Springer.

Clarke, E. M., Grumberg, O., and Peled, D. A. (1999). *Model Checking*. MIT Press, Cambridge, Massachussetts.

Classen, A., Cordy, M., Heymans, P., Legay, A., and Schobbens, P.-Y. (2014). Formal Semantics, Modular Specification, and Symbolic Verification of Product-Line Behaviour. *Science of Computer Programming (SCP)*, 80, Part B(0):416–439.

Classen, A., Cordy, M., Schobbens, P.-Y., Heymans, P., Legay, A., and Raskin, J.-F. (2013). Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Trans. Software Engineering (TSE)*, 39(8):1069–1089.

Classen, A., Heymans, P., Schobbens, P.-Y., and Legay, A. (2011). Symbolic Model Checking of Software Product Lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 321–330, New York, NY, USA. ACM.

Classen, A., Heymans, P., Schobbens, P.-Y., Legay, A., and Raskin, J.-F. (2010). Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 335–344, New York, NY, USA. ACM.

Clements, P. and Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, USA.

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

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

Cok, D. R. (2011). OpenJML: JML for Java 7 by Extending OpenJDK. In *Proc. Int'l Conf. NASA Formal Methods (NFM)*, pages 472–479, Berlin, Heidelberg. Springer.

Cok, D. R. and Kiniry, J. (2005). ESC/Java2: Uniting ESC/Java and JML. In *Proc. Int'l Conf. Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*, pages 108–128, Berlin, Heidelberg. Springer.

Coq Development Team (2010). *The Coq Proof Assistant Reference Manual.* LogiCal Project. Version 8.3.

Cordy, M., Classen, A., Heymans, P., Legay, A., and Schobbens, P.-Y. (2013a). Model Checking Adaptive Software with Featured Transition Systems. In *Proc. Workshop Assurances for Self-Adaptive Systems (ASAS)*, pages 1–29, Berlin, Heidelberg. Springer.

Cordy, M., Classen, A., Perrouin, G., Schobbens, P.-Y., Heymans, P., and Legay, A. (2012a). Simulation-Based Abstractions for Software Product-Line Model Checking. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 672–682, Piscataway, NJ, USA. IEEE.

Cordy, M., Classen, A., Schobbens, P.-Y., Heymans, P., and Legay, A. (2012b). Managing Evolution in Software Product Lines: A Model-Checking Perspective. In *Proc. Int'l Workshop Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 183–191, New York, NY, USA. ACM.

Cordy, M., Schobbens, P.-Y., Heymans, P., and Legay, A. (2012c). Behavioural Modelling and Verification of Real-Time Software Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 66–75, New York, NY, USA. ACM.

Cordy, M., Schobbens, P.-Y., Heymans, P., and Legay, A. (2012d). Towards an Incremental Automata-Based Approach for Software Product-Line Model Checking. In *Proc. Int'l Workshop Formal Methods and Analysis in Software Product Line Engineering (FMSPLE)*, pages 74–81, New York, NY, USA. ACM.

Cordy, M., Schobbens, P.-Y., Heymans, P., and Legay, A. (2013b). Beyond Boolean Product-Line Model Checking: Dealing with Feature Attributes and Multi-Features. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 472–481, Piscataway, NJ, USA. IEEE.

Cordy, M., Willemart, M., Dawagne, B., Heymans, P., and Schobbens, P.-Y. (2014). An Extensible Platform for Product-Line Behavioural Analysis. In *Proc. Workshop Software Product Line Analysis Tools (SPLat)*, pages 102–109, New York, NY, USA. ACM.

Cousot, P. and Cousot, R. (1977). Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. Symposium Principles of Programming Languages (POPL)*, pages 238–252, New York, NY, USA. ACM.

Czarnecki, K. and Eisenecker, U. (2000). *Generative Programming: Methods, Tools, and Applications.* ACM/Addison-Wesley, New York, NY, USA.

Czarnecki, K. and Pietroszek, K. (2006). Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 211–220, New York, NY, USA. ACM.

Da Mota Silveira Neto, P. A., Carmo Machado, I. D., McGregor, J. D., De Almeida, E. S., and De Lemos Meira, S. R. (2011). A Systematic Mapping Study of Software Product Lines Testing. *J. Information and Software Technology (IST)*, 53(5):407–423.

Damiani, F., Owe, O., Dovland, J., Schaefer, I., Johnsen, E. B., and Yu, I. C. (2012). A Transformational Proof System for Delta-Oriented Programming. In *Proc. Int'l Workshop Formal Methods and Analysis in Software Product Line Engineering (FM-SPLE)*, pages 53–60, New York, NY, USA. ACM.

Damiani, F. and Schaefer, I. (2012). Family-Based Analysis of Type Safety for Delta-Oriented Software Product Lines. In *Proc. Int'l Symposium Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 193–207, Berlin, Heidelberg. Springer.

Darwin, I. F. (1986). *Checking C Programs with Lint*. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

Delaware, B., Cook, W., and Batory, D. (2009). Fitting the Pieces Together: A Machine-Checked Model of Safe Composition. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 243–252, New York, NY, USA. ACM.

Delaware, B., Cook, W., and Batory, D. (2011). Product Lines of Theorems. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 595–608, New York, NY, USA. ACM.

Delaware, B., d. S. Oliveira, B. C., and Schrijvers, T. (2013). Meta-Theory à la Carte. In *Proc. Symposium Principles of Programming Languages (POPL)*, pages 207–218, New York, NY, USA. ACM.

Detlefs, D., Nelson, G., and Saxe, J. B. (2005). Simplify: A Theorem Prover for Program Checking. *J. ACM*, 52(3):365–473.

Dhara, K. K. and Leavens, G. T. (1996). Forcing Behavioral Subtyping through Specification Inheritance. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 258–267, Washington, DC, USA. IEEE.

Dijkstra, E. W. (1972). Chapter i: Notes on structured programming. In Dahl, O. J., Dijkstra, E. W., and Hoare, C. A. R., editors, *Structured Programming*, pages 1–82. Academic Press Ltd., London, UK.

Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition.

Dubslaff, C., Klüppelholz, S., and Baier, C. (2014). Probabilistic Model Checking for Energy Analysis in Software Product Lines. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 169–180, New York, NY, USA. ACM.

Dybjer, P., Haiyan, Q., and Takeyama, M. (2004). Verifying Haskell Programs by Combining Testing, Model Checking and Interactive Theorem Proving. *J. Information and Software Technology (IST)*, 46(15):1011–1025.

Engström, E. and Runeson, P. (2011). Software Product Line Testing - A Systematic Mapping Study. *J. Information and Software Technology (IST)*, 53:2–13.

Etxeberria, L., Sagardui, G., and Belategi, L. (2008). Quality-Aware Software Product Line Engineering. *J. Brazilian Computer Society (JBCS)*, 14(1):57–69.

Fantechi, A. and Gnesi, S. (2008). Formal Modeling for Product Families Engineering. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 193–202, Washington, DC, USA. IEEE.

Filliâtre, J.-C. and Marché, C. (2007). The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In *Computer Aided Verification*, pages 173–177, Berlin, Heidelberg. Springer.

Findler, R. B. and Felleisen, M. (2002). Contracts for Higher-Order Functions. In *Proc. Int'l Conf. Functional Programming (ICFP)*, pages 48–59, New York, NY, USA. ACM.

Findler, R. B., Latendresse, M., and Felleisen, M. (2001). Behavioral Contracts and Behavioral Subtyping. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 229–236, New York, NY, USA. ACM.

Fischbein, D., Uchitel, S., and Braberman, V. (2006). A Foundation for Behavioural Conformance in Software Product Line Architectures. In *Proc. Int'l Workshop Role of Software Architecture for Testing and Analysis (ROSATEA)*, pages 39–48, New York, NY, USA. ACM.

Fisler, K. and Krishnamurthi, S. (2001). Modular Verification of Collaboration-Based Software Designs. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 152–163, New York, NY, USA. ACM.

Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. (2002). Extended Static Checking for Java. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, pages 234–245, New York, NY, USA. ACM.

Flatt, M., Krishnamurthi, S., and Felleisen, M. (1998). Classes and Mixins. In *Proc. Symposium Principles of Programming Languages (POPL)*, pages 171–183, New York, NY, USA. ACM.

Floyd, R. W. (1967). Assigning Meanings to Programs. *Mathematical Aspects of Computer Science*, 19:19–32.

Gazzillo, P. and Grimm, R. (2012). SuperC: Parsing All of C by Taming the Preprocessor. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, pages 323–334, New York, NY, USA. ACM.

Gondal, A., Poppleton, M., and Butler, M. (2011). Composing Event-B Specifications: Case-Study Experience. In *Proc. Int'l Symposium Software Composition (SC)*, pages 100–115, Berlin, Heidelberg. Springer.

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

Greenyer, J., Sharifloo, A. M., Cordy, M., and Heymans, P. (2013). Features Meet Scenarios: Modeling and Consistency-Checking Scenario-Based Product Line Specifications. *Requirements Engineering*, 18(2):175–198.

Gries, D. (1981). *The Science of Programming*. Springer, Secaucus, NJ, USA, 1st edition.

Griswold, W. G., Sullivan, K., Song, Y., Shonle, M., Tewari, N., Cai, Y., and Rajan, H. (2006). Modular Software Design with Crosscutting Interfaces. *IEEE Software*, 23(1):51–60.

Gruler, A., Leucker, M., and Scheidemann, K. (2008). Modeling and Model Checking Software Product Lines. In *Proc. IFIP Int'l Conf. Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 113–131, Berlin, Heidelberg. Springer.

Hähnle, R. and Schaefer, I. (2012). A Liskov Principle for Delta-Oriented Programming. In *Proc. Int'l Symposium Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 32–46, Berlin, Heidelberg. Springer.

Hähnle, R., Schaefer, I., and Bubel, R. (2013). Reuse in Software Verification by Abstract Method Calls. In *Proc. Int'l Conf. Automated Deduction (CADE)*, pages 300–314, Berlin, Heidelberg. Springer.

Hall, R. J. (2005). Fundamental Nonmodularity in Electronic Mail. *Automated Software Engineering*, 12(1):41–79.

Halpern, J. Y. and Vardi, M. Y. (1991). Model Checking vs. Theorem Proving: A Manifesto. In Lifschitz, V., editor, *Artificial Intelligence and Mathematical Theory of Computation*, pages 151–176. Academic Press Professional, Inc., San Diego, CA, USA.

Harhurin, A. and Hartmann, J. (2008). Towards Consistent Specifications of Product Families. In *Proc. Int'l Symposium Formal Methods (FM)*, pages 390–405, Berlin, Heidelberg. Springer.

Harrison, W. and Ossher, H. (1993). Subject-Oriented Programming: A Critique of Pure Objects. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 411–428, New York, NY, USA. ACM.

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

Havelund, K. and Pressburger, T. (2000). Model Checking Java Programs Using Java PathFinder. *Int'l J. Software Tools for Technology Transfer (STTT)*, 2(4):366–381.

Heidenreich, F. (2009). Towards Systematic Ensuring Well-Formedness of Software Product Lines. In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD)*, pages 69–74, New York, NY, USA. ACM.

Helm, R., Holland, I. M., and Gangopadhyay, D. (1990). Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 169–180, New York, NY, USA. ACM.

Hemel, A. and Koschke, R. (2012). Reverse Engineering Variability in Source Code Using Clone Detection: A Case Study for Linux Variants of Consumer Electronic Devices. In *Proc. Working Conf. Reverse Engineering (WCRE)*, pages 357–366, Washington, DC, USA. IEEE.

Hoare, C. A. R. (1969). An Axiomatic Basis for Computer Programming. *Comm. ACM*, 12(10):576–580.

Hoare, C. A. R. (1972). Proof of Correctness of Data Representations. *Acta Informatica*, 1(4):271–281.

Höfner, P. and Möller, B. (2009). An Extension for Feature Algebra. In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD)*, pages 75–80, New York, NY, USA. ACM.

Höfner, P., Möller, B., and Zelend, A. (2012). Foundations of Coloring Algebra with Consequences for Feature-Oriented Programming. In *Proc. Int'l Conf. Relational and Algebraic Methods in Computer Science (RAMiCS)*, pages 33–49, Berlin, Heidelberg. Springer.

Holzmann, G. J. (1997). The Model Checker SPIN. *IEEE Trans. Software Engineering (TSE)*, 23(5):279–295.

Hovemeyer, D. and Pugh, W. (2004). Finding Bugs is Easy. *SIGPLAN Not.*, 39(12):92–106.

Huang, S. S., Zook, D., and Smaragdakis, Y. (2007). cJ: Enhancing Java with Safe Type Conditions. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 185–198, New York, NY, USA. ACM.

Huang, S. S., Zook, D., and Smaragdakis, Y. (2011). Statically Safe Program Generation with SafeGen. *Science of Computer Programming (SCP)*, 76(5):376–391.

Ismail, M., Hasan, O., Ebi, T., Shafique, M., and Henkel, J. (2013). Formal Verification of Distributed Dynamic Thermal Management. In *Proc. Int'l Conf. Computer-Aided Design (ICCAD)*, pages 248–255, Piscataway, NJ, USA. IEEE.

Istoan, P. (2013). *Methodology for the Derivation of Product Behaviour in a Software Product Line.* PhD thesis, Université Rennes 1, Luxembourg.

Janota, M., Kiniry, J., and Botterweck, G. (2008). Formal Methods in Software Product Lines: Concepts, Survey, and Guidelines. Technical Report Lero-TR-SPL-2008-02, Lero, University of Limerick.

Jayaraman, P., Whittle, J., Elkhodary, A. M., and Gomaa, H. (2007). Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. In *Proc. Int'l Conf. Model Driven Engineering Languages and Systems (MODELS)*, pages 151–165, Berlin, Heidelberg. Springer.

Jia, Y. and Harman, M. (2011). An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Software Engineering (TSE)*, 37(5):649–678.

Johnsen, E. B., Hähnle, R., Schäfer, J., Schlatte, R., and Steffen, M. (2012). ABS: A Core Language for Abstract Behavioral Specification. In *Proc. Int'l Symposium Formal Methods for Components and Objects (FMCO)*, pages 142–164, Berlin, Heidelberg. Springer.

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

Kanning, F. and Schulze, S. (2014). Program Slicing in the Presence of Variability. In *Proc. Int'l Conf. Software Maintenance and Evolution (ICSME)*, pages 501–505, Washington, DC, USA. IEEE.

Kästner, C., Apel, S., and Kuhlemann, M. (2009a). A Model of Refactoring Physically and Virtually Separated Features. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 157–166, New York, NY, USA. ACM.

Kästner, C., Apel, S., and Ostermann, K. (2011a). The Road to Feature Modularity? In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD)*, pages 5:1–5:8, New York, NY, USA. ACM.

Kästner, C., Apel, S., Thüm, T., and Saake, G. (2012a). Type Checking Annotation-Based Product Lines. *Trans. Software Engineering and Methodology (TOSEM)*, 21(3):14:1–14:39.

Kästner, C., Apel, S., ur Rahman, S. S., Rosenmüller, M., Batory, D., and Saake, G. (2009b). On the Impact of the Optional Feature Problem: Analysis and Case Studies. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 181–190, Pittsburgh, PA, USA. Software Engineering Institute.

Kästner, C., Giarrusso, P. G., Rendel, T., Erdweg, S., Ostermann, K., and Berger, T. (2011b). Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 805–824, New York, NY, USA. ACM.

Kästner, C., Ostermann, K., and Erdweg, S. (2012b). A Variability-Aware Module System. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 773–792, New York, NY, USA. ACM.

Kästner, C., Thüm, T., Saake, G., Feigenspan, J., Leich, T., Wielgorz, F., and Apel, S. (2009c). FeatureIDE: A Tool Framework for Feature-Oriented Software Development. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 611–614, Washington, DC, USA. IEEE. Formal demonstration paper.

Kästner, C., von Rhein, A., Erdweg, S., Pusch, J., Apel, S., Rendel, T., and Ostermann, K. (2012c). Toward Variability-Aware Testing. In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD)*, pages 1–8, New York, NY, USA. ACM.

Katz, S. (2006). Aspect Categories and Classes of Temporal Properties. *Trans. Aspect-Oriented Software Development*, 1:106–134.

Kenner, A., Kästner, C., Haase, S., and Leich, T. (2010). TypeChef: Toward Type Checking #Ifdef Variability in C. In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD)*, pages 25–32, New York, NY, USA. ACM.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An Overview of AspectJ. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 327–354, London, UK. Springer.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-Oriented Programming. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 220–242, Berlin, Heidelberg. Springer.

Kim, C. H. P., Batory, D., and Khurshid, S. (2011). Reducing Combinatorics in Testing Product Lines. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 57—68, New York, NY, USA. ACM.

Kim, C. H. P., Bodden, E., Batory, D., and Khurshid, S. (2010). Reducing Configurations to Monitor in a Software Product Line. In *Proc. Int'l Conf. Runtime Verification (RV)*, pages 285–299, Berlin, Heidelberg. Springer.

Kim, C. H. P., Kästner, C., and Batory, D. (2008). On the Modularity of Feature Interactions. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 23–34, New York, NY, USA. ACM.

Kim, C. H. P., Khurshid, S., and Batory, D. (2012). Shared Execution for Efficiently Testing Product Lines. In *Proc. Int'l Symposium Software Reliability Engineering (ISSRE)*, pages 221–230, Washington, DC, USA. IEEE.

Kim, C. H. P., Marinov, D., Khurshid, S., Batory, D., Souto, S., Barros, P., and D'Amorim, M. (2013). SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 257–267, New York, NY, USA. ACM.

Kishi, T. and Noda, N. (2006). Formal Verification and Software Product Lines. *Comm. ACM*, 49:73–77.

Klaeren, H., Pulvermueller, E., Rashid, A., and Speck, A. (2001). Aspect Composition Applying the Design by Contract Principle. In *Proc. Int'l Symposium Generative and Component-Based Software Engineering (GCSE)*, pages 57–69, Berlin, Heidelberg. Springer.

Klose, K. and Ostermann, K. (2010). Modular Logic Metaprogramming. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 484–503, New York, NY, USA. ACM.

Kolesnikov, S., von Rhein, A., Hunsen, C., and Apel, S. (2013). A Comparison of Product-Based, Feature-Based, and Family-Based Type Checking. In *Proc. Int'l Conf. Generative Programming: Concepts & Experiences (GPCE)*, pages 115–124, New York, NY, USA. ACM.

Kuhlemann, M., Batory, D., and Kästner, C. (2009). Safe Composition of Non-Monotonic Features. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 177–186, New York, NY, USA. ACM.

Kuhlemann, M. and Sturm, M. (2010). Patching Product Line Programs. In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD)*, pages 33–40, New York, NY, USA. ACM.

Laguna, M. A. and Crespo, Y. (2013). A Systematic Mapping Study on Software Product Line Evolution: From Legacy System Reengineering to Product Line Refactoring. *Science of Computer Programming (SCP)*, 78(8):1010–1034.

Lattner, C. (2008). LLVM and Clang: Next Generation Compiler Technology. In *Proc. BSD Conference (BSDCan)*.

Lauenroth, K., Metzger, A., and Pohl, K. (2010). Quality Assurance in the Presence of Variability. In *Intentional Perspectives on Information Systems Engineering*, pages 319–333, Berlin, Heidelberg. Springer.

Le, D. M., Lee, H., Kang, K. C., and Keun, L. (2013). Validating Consistency between a Feature Model and Its Implementation. In *Proc. Int'l Conf. Software Reuse (ICSR)*, pages 1–16, Berlin, Heidelberg. Springer.

Leavens, G. T., Baker, A. L., and Ruby, C. (2006). Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *SIGSOFT Software Engineering Notes*, 31(3):1–38.

Leavens, G. T. and Müller, P. (2007). Information Hiding and Visibility in Interface Specifications. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 385–395, Washington, DC, USA. IEEE.

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

Lee, J., Kang, S., and Lee, D. (2012). A Survey on Software Product Line Testing. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 31–40, New York, NY, USA. ACM.

Lee, J., Muthig, D., and Naab, M. (2008). An Approach for Developing Service Oriented Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 275–284, Washington, DC, USA. IEEE.

Leino, K. R. M. (1998). Data Groups: Specifying the Modification of Extended State. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 144–153, New York, NY, USA. ACM.

Li, H., Krishnamurthi, S., and Fisler, K. (2002). Verifying Cross-Cutting Features as Open Systems. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, pages 89–98, New York, NY, USA. ACM.

Li, H., Krishnamurthi, S., and Fisler, K. (2005). Modular Verification of Open Features Using Three-Valued Model Checking. *Automated Software Engineering*, 12(3):349–382.

Liebig, J., Apel, S., Lengauer, C., Kästner, C., and Schulze, M. (2010). An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 105–114, Washington, DC, USA. IEEE.

Liebig, J., von Rhein, A., Kästner, C., Apel, S., Dörre, J., and Lengauer, C. (2013). Scalable Analysis of Variable Software. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 81–91, New York, NY, USA. ACM.

Liskov, B. and Guttag, J. (1986). *Abstraction and Specification in Program Development*. MIT Press, Cambridge, MA, USA.

Liskov, B. H. and Wing, J. M. (1994). A Behavioral Notion of Subtyping. *ACM Trans. Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841.

Liu, J., Basu, S., and Lutz, R. (2011). Compositional Model Checking of Software Product Lines Using Variation Point Obligations. *Automated Software Engineering*, 18(1):39–76.

Liu, J., Batory, D., and Lengauer, C. (2006). Feature Oriented Refactoring of Legacy Applications. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 112–121, New York, NY, USA. ACM.

Liu, J., Dehlinger, J., and Lutz, R. (2007). Safety Analysis of Software Product Lines Using State-Based Modeling. *J. Systems and Software (JSS)*, 80(11):1879–1892.

Lochau, M., Mennicke, S., Baller, H., and Ribbeck, L. (2014). DeltaCCS: A Core Calculus for Behavioral Change. In Margaria, T. and Steffen, B., editors, *Proc. Int'l Symposium Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 320–335, Berlin, Heidelberg. Springer.

Lopez-Herrejon, R. E. and Batory, D. (2001). A Standard Problem for Evaluating Product-Line Methodologies. In *Proc. Int'l Symposium Generative and Component-Based Software Engineering (GCSE)*, pages 10–24, London, UK. Springer.

Lorenz, D. H. and Skotiniotis, T. (2005). Extending Design by Contract for Aspect-Oriented Programming. *Computing Research Repository (CoRR)*, abs/cs/0501070.

Lutz, R. (2007). Survey of Product-Line Verification and Validation Techniques. Technical Report 2014/41221, NASA, Jet Propulsion Laboratory, La Canada Flintridge, CA, USA.

McIlroy, M. D. (1968). Mass Produced Software Components. In *Proc. NATO Conf. Software Engineering*, pages 138–155. Springer.

Medeiros, F., Ribeiro, M., and Gheyi, R. (2013). Investigating Preprocessor-Based Syntax Errors. In *Proc. Int'l Conf. Generative Programming: Concepts & Experiences (GPCE)*, pages 75–84, New York, NY, USA. ACM.

Meinicke, J. (2013). JML-Based Verification for Feature-Oriented Programming. Bachelor's thesis, University of Magdeburg, Germany.

Meinicke, J., Thüm, T., Schöter, R., Benduhn, F., and Saake, G. (2014). An Overview on Analysis Tools for Software Product Lines. In *Proc. Workshop Software Product Line Analysis Tools (SPLat)*, pages 94–101, New York, NY, USA. ACM.

Mendonça, M., Wąsowski, A., and Czarnecki, K. (2009). SAT-Based Analysis of Feature Models is Easy. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 231–240, Pittsburgh, PA, USA. Software Engineering Institute.

Metzger, A. (2007). Quality Issues in Software Product Lines: Feature Interactions and Beyond. In *Proc. Int'l Conf. Feature Interactions in Software and Communication Systems (ICFI)*, pages 1–12, Amsterdam, The Netherlands. IOS Press.

Metzger, A., Pohl, K., Heymans, P., Schobbens, P.-Y., and Saval, G. (2007). Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis. In *Proc. Int'l Conf. Requirements Engineering (RE)*, pages 243–253, Washington, DC, USA. IEEE.

Meyer, B. (1988). *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition.

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

Mezini, M. and Lieberherr, K. (1998). Adaptive Plug-and-Play Components for Evolutionary Software Development. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 97–116, New York, NY, USA. ACM.

Midtgaard, J., Brabrand, C., and Wąsowski, A. (2014). Systematic Derivation of Static Analyses for Software Product Lines. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 181–192, New York, NY, USA. ACM.

Molderez, T. and Janssens, D. (2012). Design by Contract for Aspects, by Aspects. In *Proc. Workshop Foundations of Aspect-Oriented Languages (FOAL)*, pages 9–14, New York, NY, USA. ACM.

Molderez, T. and Janssens, D. (2015). Modular Reasoning in Aspect-Oriented Languages from a Substitution Perspective. *Trans. Aspect-Oriented Software Development*, pages 3–59.

Montagud, S. and Abrahão, S. (2009). Gathering Current Knowledge About Quality Evaluation in Software Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 91–100, Pittsburgh, PA, USA. Software Engineering Institute.

Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Mulet, P., Malenfant, J., and Cointe, P. (1995). Towards a Methodology for Explicit Composition of MetaObjects. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 316–330, New York, NY, USA. ACM.

Necula, G. C. (1997). Proof-Carrying Code. In *Proc. Symposium Principles of Programming Languages (POPL)*, pages 106–119, New York, NY, USA. ACM.

Nelson, T., Cowan, D. D., and Alencar, P. S. C. (2001). Supporting Formal Verification of Crosscutting Concerns. In *Proc. Int'l Conf. Metalevel Architectures and Separation of Crosscutting Concerns*, pages 153–169, London, UK. Springer.

Nguyen, H. V., Kästner, C., and Nguyen, T. N. (2014a). Building Call Graphs for Embedded Client-Side Code in Dynamic Web Applications. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, pages 518–529, New York, NY, USA. ACM.

Nguyen, H. V., Kästner, C., and Nguyen, T. N. (2014b). Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 907–918, New York, NY, USA. ACM.

Nie, C. and Leung, H. (2011). A Survey of Combinatorial Testing. *ACM Computing Surveys*, 43(2):11:1–11:29.

Nielson, F., Nielson, H. R., and Hankin, C. (2010). *Principles of Program Analysis*. Springer, Secaucus, NJ, USA.

Nipkow, T., Wenzel, M., and Paulson, L. C. (2002). *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, Berlin, Heidelberg.

Oster, S., Markert, F., and Ritter, P. (2010). Automated Incremental Pairwise Testing of Software Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 196–210, Berlin, Heidelberg. Springer.

Oster, S., Wübbeke, A., Engels, G., and Schürr, A. (2011). A Survey of Model-Based Software Product Lines Testing. In *Model-Based Testing for Embedded System*, pages 339–381. CRC Press, Boca Raton, FL, USA.

Owre, S., Rajan, S. P., Rushby, J. M., Shankar, N., and Srivas, M. K. (1996). PVS: Combining Specification, Proof Checking, and Model Checking. In *Proc. Int'l Conf. Computer Aided Verification (CAV)*, pages 411–414, Berlin, Heidelberg. Springer.

Owre, S., Rushby, J. M., and Shankar, N. (1992). PVS: A Prototype Verification System. In *Proc. Int'l Conf. Automated Deduction (CADE)*, pages 748–752, London, UK. Springer.

Parnas, D. L. (1976). On the Design and Development of Program Families. *IEEE Trans. Software Engineering (TSE)*, SE-2(1):1–9.

Perrouin, G., Sen, S., Klein, J., Baudry, B., and Le Traon, Y. (2010). Automated and Scalable T-Wise Test Case Generation Strategies for Software Product Lines. In *Proc. Int'l Conf. Software Testing, Verification and Validation (ICST)*, pages 459–468, Washington, DC, USA. IEEE.

Pierce, B. C. (2002). *Types and Programming Languages.* MIT Press, Cambridge, Massachusetts, USA.

Plath, M. and Ryan, M. (2001). Feature Integration Using a Feature Construct. *Science of Computer Programming (SCP)*, 41(1):53–84.

Pohl, K., Böckle, G., and van der Linden, F. J. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer, Berlin, Heidelberg.

Poppleton, M. (2007). Towards Feature-Oriented Specification and Development with Event-B. In *Proc. Int'l Working Conf. Requirements Engineering: Foundation for Software Quality (REFSQ)*, pages 367–381, Berlin, Heidelberg. Springer.

Poppleton, M. (2008). The Composition of Event-B Models. In *Proc. Int'l Conf. Abstract State Machines, Alloy, B and Z (ABZ)*, pages 209–222, Berlin, Heidelberg. Springer.

Post, H. and Sinz, C. (2008). Configuration Lifting: Software Verification Meets Software Configuration. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 347–350, Washington, DC, USA. IEEE.

Prehofer, C. (1997). Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 419–443, Berlin, Heidelberg. Springer.

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

Rebêlo, H., Coelho, R., Lima, R., Leavens, G. T., Huisman, M., Mota, A., and Castor Filho, F. (2011). On the Interplay of Exception Handling and Design by Contract: An Aspect-Oriented Recovery Approach. In *Proc. Workshop Formal Techniues for Java-Like Programs (FTfJP)*, pages 7:1–7:6, New York, NY, USA. ACM.

Rebêlo, H., Leavens, G. T., Bagherzadeh, M., Rajan, H., Lima, R., Zimmerman, D. M., Cornélio, M., and Thüm, T. (2014). AspectJML: Modular Specification and Runtime Checking for Crosscutting Contracts. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 157–168, New York, NY, USA. ACM.

Rebêlo, H., Leavens, G. T., Lima, R., Borba, P., and Ribeiro, M. (2013a). Modular Aspect-Oriented Design Rule Enforcement with XPIDRs. In *Proc. Workshop Foundations of Aspect-Oriented Languages (FOAL)*, pages 13–18, New York, NY, USA. ACM.

Rebêlo, H., Lima, R., Cornélio, M., and Soares, S. (2008). A JML Compiler Based on AspectJ. In *Proc. Int'l Conf. Software Testing, Verification and Validation (ICST)*, pages 541–544, Washington, DC, USA. IEEE.

Rebêlo, H., Lima, R., Kulesza, U., Ribeiro, M., Cai, Y., Coelho, R., Sant'Anna, C., and Mota, A. (2013b). Quantifying the Effects of Aspectual Decompositions on Design by Contract Modularization: A Maintenance Study. *Int'l J. Software Engineering and Knowledge Engineering (IJSEKE)*, 23(7):913–942.

Rebêlo, H., Soares, S., Lima, R., Borba, P., and Cornélio, M. (2008). JML and Aspects: The Benefits of Instrumenting JML Features with AspectJ. In *Proc. Int'l Workshop Specification and Verification of Component-Based Systems (SAVCBS)*, pages 11–18, Orlando, Florida, USA. University of Central Florida.

Rhanoui, M. and Asri, B. E. (2014). A Contractual Specification of Functional and Non-Functional Requirements of Domain-Specific Components. *Int'l J. Computer Science Issues (IJCSI)*, 11(2):172–181.

Ribeiro, M., Pacheco, H., Teixeira, L., and Borba, P. (2010). Emergent Feature Modularization. In *Proc. Int'l Conf. Object-Oriented Programming Systems Languages and Applications Companion (SPLASH)*, pages 11–18, New York, NY, USA. ACM.

Robby, Rodríguez, E., Dwyer, M. B., and Hatcliff, J. (2006). Checking JML Specifications Using an Extensible Software Model Checking Framework. *Int'l J. Software Tools for Technology Transfer (STTT)*, 8(3):280–299.

Roy, C. K., Cordy, J. R., and Koschke, R. (2009). Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Science of Computer Programming (SCP)*, 74(7):470–495.

Rubanov, V. V. and Shatokhin, E. A. (2011). Runtime Verification of Linux Kernel Modules Based on Call Interception. In *Proc. Int'l Conf. Software Testing, Verification and Validation (ICST)*, pages 180–189, Washington, DC, USA. IEEE.

Rubin, J. and Chechik, M. (2013). A Framework for Managing Cloned Product Variants. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 1233–1236, Piscataway, NJ, USA. IEEE.

Sabouri, H. and Khosravi, R. (2012). Efficient Verification of Evolving Software Product Lines. In *Proc. Int'l Conf. Fundamentals of Software Engineering (FSEN)*, pages 351–358, Berlin, Heidelberg. Springer.

Sabouri, H. and Khosravi, R. (2013a). Delta Modeling and Model Checking of Product Families. In *Proc. Int'l Conf. Fundamentals of Software Engineering (FSEN)*, pages 51–65, Berlin, Heidelberg. Springer.

Sabouri, H. and Khosravi, R. (2013b). Modeling and Verification of Reconfigurable Actor Families. *J. Universal Computer Science (J.UCS)*, 19(2):207–232.

Sabouri, H. and Khosravi, R. (2014). Reducing the Verification Cost of Evolving Product Families Using Static Analysis Techniques. *Science of Computer Programming (SCP)*, 83(0):35–55.

Schaefer, I., Bettini, L., Bono, V., Damiani, F., and Tanzarella, N. (2010a). Delta-Oriented Programming of Software Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 77–91, Berlin, Heidelberg. Springer.

Schaefer, I., Bettini, L., and Damiani, F. (2011). Compositional Type-Checking for Delta-Oriented Programming. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 43–56, New York, NY, USA. ACM.

Schaefer, I., Gurov, D., and Soleimanifard, S. (2010b). Compositional Algorithmic Verification of Software Product Lines. In *Proc. Int'l Symposium Formal Methods for Components and Objects (FMCO)*, pages 184–203, Berlin, Heidelberg. Springer.

Scholz, W., Thüm, T., Apel, S., and Lengauer, C. (2011). Automatic Detection of Feature Interactions Using the Java Modeling Language: An Experience Report. In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD)*, pages 7:1–7:8, New York, NY, USA. ACM.

Schröter, R., Siegmund, N., and Thüm, T. (2013). Towards Modular Analysis of Multi Product Lines. In *Proc. Int'l Workshop Multi Product Line Engineering (MultiPLE)*, pages 96–99, New York, NY, USA. ACM.

Schröter, R., Siegmund, N., Thüm, T., and Saake, G. (2014). Feature-Context Interfaces: Tailored Programming Interfaces for Software Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 102–111, New York, NY, USA. ACM.

Schumann, J. (2001). *Automated Theorem Proving in Software Engineering*. Springer, Berlin, Heidelberg.

Sedgewick, R. (1983). *Algorithms*. Addison-Wesley.

Shi, J., Cohen, M. B., and Dwyer, M. B. (2012). Integration Testing of Software Product Lines Using Compositional Symbolic Execution. In *Proc. Int'l Conf. Fundamental Approaches to Software Engineering (FASE)*, pages 270–284, Berlin, Heidelberg. Springer.

Shi, Y., Wei, O., and Zhou, Y. (2014). Model Checking Partial Software Product Line Designs. In *Proc. Workshop Innovative Software Development Methodologies and Practices (InnoSWDev)*, pages 21–29, New York, NY, USA. ACM.

Shinotsuka, S., Ubayashi, N., Shinomi, H., and Tamai, T. (2006). An Extensible Contract Verifier for AspectJ. In *Proc. Asian Workshop Aspect-Oriented Software Development (AOAsia)*, pages 1:1–1:6, Washington, DC, USA. IEEE.

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

Smith, B. C. (1985). The Limits of Correctness. *SIGCAS Comput. Soc.*, 14,15(1,2,3,4):18–26.

Sorge, J., Poppleton, M., and Butler, M. (2010). A Basis for Feature-Oriented Modelling in Event-B. In *Proc. Int'l Conf. Abstract State Machines, Alloy, B and Z (ABZ)*, pages 409–409, Berlin, Heidelberg. Springer.

Strichman, O. and Godlin, B. (2008). Verified Software: Theories, Tools, Experiments. In Meyer, B. and Woodcock, J., editors, *Regression Verification - A Practical Way to Verify Programs*, pages 496–501. Springer, Berlin, Heidelberg.

Strickland, T. S., Dimoulas, C., Takikawa, A., and Felleisen, M. (2013). Contracts for First-Class Classes. *ACM Trans. Programming Languages and Systems (TOPLAS)*, 35(3):11:1–11:58.

Strickland, T. S. and Felleisen, M. (2010). Contracts for First-Class Classes. In *Proc. Symposium on Dynamic Languages (DLS)*, pages 97–112, New York, NY, USA. ACM.

Takikawa, A., Strickland, T. S., Dimoulas, C., Tobin-Hochstadt, S., and Felleisen, M. (2012). Gradual Typing for First-Class Classes. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 793–810, New York, NY, USA. ACM.

Tarr, P., Ossher, H., Harrison, W., and Sutton, Jr., S. M. (1999). N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 107–119, New York, NY, USA. ACM.

Tartler, R., Lohmann, D., Dietrich, C., Egger, C., and Sincero, J. (2012). Configuration Coverage in the Analysis of Large-Scale System Software. *ACM SIGOPS Operating Systems Review*, 45(3):10–14.

Tartler, R., Lohmann, D., Sincero, J., and Schröder-Preikschat, W. (2011). Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *Proc. Europ. Conf. Computer Systems (EuroSys)*, pages 47–60, New York, NY, USA. ACM.

Teixeira, L., Borba, P., and Gheyi, R. (2011). Safe Composition of Configuration Knowledge-Based Software Product Lines. In *Proc. Brazilian Symposium Software Engineering (SBES)*, pages 263–272, Washington, DC, USA. IEEE.

ter Beek, M. H. and de Vink, E. P. (2014). Software Product Line Analysis with mCRL2. In *Proc. Workshop Software Product Line Analysis Tools (SPLat)*, pages 78–85, New York, NY, USA. ACM.

ter Beek, M. H., Lafuente, A. L., and Petrocchi, M. (2013). Combining Declarative and Procedural Views in the Specification and Analysis of Product Families. In *Proc. Int'l Workshop Formal Methods and Analysis in Software Product Line Engineering (FMSPLE)*, pages 10–17, New York, NY, USA. ACM.

Tevanlinna, A., Taina, J., and Kauppinen, R. (2004). Product Family Testing: A Survey. *SIGSOFT Software Engineering Notes*, 29:12–17.

Thaker, S., Batory, D., Kitchin, D., and Cook, W. (2007). Safe Composition of Product Lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 95–104, New York, NY, USA. ACM.

Thüm, T. (2010). A Machine-Checked Proof for a Product-Line–Aware Type System. Master's thesis, University of Magdeburg, Germany.

Thüm, T. (2013). Product-Line Verification with Feature-Oriented Contracts. In *Proc. Int'l Symposium in Software Testing and Analysis (ISSTA)*, pages 374–377, New York, NY, USA. ACM.

Thüm, T., Apel, S., Kästner, C., Schaefer, I., and Saake, G. (2014a). A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys*, 47(1):6:1–6:45.

Thüm, T., Apel, S., Zelend, A., Schröter, R., and Möller, B. (2013). Subclack: Feature-Oriented Programming with Behavioral Feature Interfaces. In *Proc. Workshop Mech-Anisms for SPEcialization, Generalization and inHerItance (MASPEGHI)*, pages 1–8, New York, NY, USA. ACM.

Thüm, T., Batory, D., and Kästner, C. (2009). Reasoning about Edits to Feature Models. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 254–264, Washington, DC, USA. IEEE.

Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., and Leich, T. (2014b). FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming (SCP)*, 79(0):70–85.

Thüm, T., Kästner, C., Erdweg, S., and Siegmund, N. (2011a). Abstract Features in Feature Modeling. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 191–200, Washington, DC, USA. IEEE.

Thüm, T., Meinicke, J., Benduhn, F., Hentschel, M., von Rhein, A., and Saake, G. (2014). Potential Synergies of Theorem Proving and Model Checking for Software Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 177–186, New York, NY, USA. ACM.

Thüm, T., Schaefer, I., Apel, S., and Hentschel, M. (2012). Family-Based Deductive Verification of Software Product Lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 11–20, New York, NY, USA. ACM.

Thüm, T., Schaefer, I., Kuhlemann, M., and Apel, S. (2011b). Proof Composition for Deductive Verification of Software Product Lines. In *Proc. Int'l Workshop Variability-intensive Systems Testing, Validation and Verification (VAST)*, pages 270–277, Washington, DC, USA. IEEE.

Thüm, T., Schaefer, I., Kuhlemann, M., Apel, S., and Saake, G. (2012). Applying Design by Contract to Feature-Oriented Programming. In *Proc. Int'l Conf. Fundamental Approaches to Software Engineering (FASE)*, pages 255–269, Berlin, Heidelberg. Springer.

Turing, A. (1949). Checking a Large Routine. In *Conference on High Speed Automatic Calculating Machines*, pages 67–69.

Ubayashi, N. and Tamai, T. (2002). Aspect-Oriented Programming with Model Checking. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 148–154, New York, NY, USA. ACM.

van der Linden, F. J., Schmid, K., and Rommes, E. (2007). *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, Berlin, Heidelberg.

VanHilst, M. and Notkin, D. (1996). Using Role Components in Implement Collaboration-Based Designs. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 359–369, New York, NY, USA. ACM.

Visser, W., Havelund, K., Brat, G. P., and Park, S. (2000). Model Checking Programs. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 3–12, Berlin, Heidelberg. Springer.

von Rhein, A., Apel, S., Kästner, C., Thüm, T., and Schaefer, I. (2013). The PLA Model: On the Combination of Product-Line Analyses. In *Proc. Int'l Workshop Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 14:1–14:8, New York, NY, USA. ACM.

Wampler, D. (2007). Aspect-Oriented Design Principles: Lessons from Object-Oriented Design. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages I6:1–I6:10, New York, NY, USA. ACM.

Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., and Wischnewski, P. (2009). SPASS Version 3.5. In *Proc. Int'l Conf. Automated Deduction (CADE)*, pages 140–145, Berlin, Heidelberg. Springer.

Weigelt, A. (2013). Methoden-basierte Komposition von Kontrakten in Feature-orientierter Programmierung. Bachelor's thesis, University of Magdeburg, Germany. In German.

Weiser, M. (1981). Program Slicing. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 439–449, Piscataway, NJ, USA. IEEE.

Weiß, B. (2011). *Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic, and Predicate Abstraction*. PhD thesis, Karlsruhe Institute of Technology, Germany.

Weiss, D. M. (2008). The Product Line Hall of Fame. In *Proc. Int'l Software Product Line Conf. (SPLC)*, page 395, Washington, DC, USA. IEEE.

Xue, Y., Xing, Z., and Jarzabek, S. (2012). Feature Location in a Collection of Product Variants. In *Proc. Working Conf. Reverse Engineering (WCRE)*, pages 145–154, Washington, DC, USA. IEEE.

Zhao, J. and Rinard, M. C. (2003). Pipa: A Behavioral Interface Specification Language for AspectJ. In *Proc. Int'l Conf. Fundamental Approaches to Software Engineering (FASE)*, pages 150–165, Berlin, Heidelberg. Springer.

Ehrenerklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; verwendete fremde und eigene Quellen sind als solche kenntlich gemacht. Insbesondere habe ich nicht die Hilfe eines kommerziellen Promotionsberaters in Anspruch genommen. Dritte haben von mir weder unmittelbar noch mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Ich habe insbesondere nicht wissentlich:
- Ergebnisse erfunden oder widersprüchliche Ergebnisse verschwiegen,
- statistische Verfahren absichtlich missbraucht, um Daten in ungerechtfertigter Weise zu interpretieren,
- fremde Ergebnisse oder Veröffentlichungen plagiiert,
- fremde Forschungsergebnisse verzerrt wiedergegeben.

Mir ist bekannt, dass Verstöße gegen das Urheberrecht Unterlassungs- und Schadensersatzansprüche des Urhebers sowie eine strafrechtliche Ahndung durch die Strafverfolgungsbehörden begründen kann. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form als Dissertation eingereicht und ist als Ganzes auch noch nicht veröffentlicht."

Magdeburg, den 23.02.2015

Thomas Thüm