University of Magdeburg

School of Computer Science

Master's Thesis

Representing Variability in Product Lines: A Survey of Modeling and Specification Techniques

Author:

Fabian Benduhn

April 23, 2014

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake
Dipl.-Inform. Thomas Thüm
Department of Technical and Business Information Systems

Benduhn, Fabian:
Representing Variability in Product Lines: A Survey of Modeling and Specification Techniques
Master's Thesis, University of Magdeburg, 2014.

Abstract

Software product lines are increasingly used to develop safety-critical and mission-critical systems. To reason about the correctness of product lines, researchers have developed special testing techniques and adapted formal verification techniques such as model checking and theorem proving to the requirements of software product lines. Existing research has focused on the strategies to enable efficient reasoning about properties of a product line. However, to reason about properties, we need a representation of the product line such as a formal model or an implementation, as well as a specification of the properties. The contributions of this thesis are twofold. First, we survey formal modeling and specification techniques for software product lines to give an overview of the state-of-the-art. Second, we propose a general taxonomy of product-line representations and classify modeling and property specification techniques for software product lines proposed in the literature. The common taxonomy helps to understand commonalities and differences between different types of representations including implementation, modeling, and specification. Based on the insights provided by our results, we identify potential directions for future research.

Acknowledgements

I would like to thank everyone who has helped me to begin with my work on this thesis or to finish it. I do not want to miss all that I have learned by working on this thesis.

I thank my advisors Thomas Thüm and Prof. Gunter Saake for giving me the opportunity to write this thesis. I am grateful for their trust in my capabilities to pursue my own vision of this thesis and for their guidance that helped me not to get lost on this path. I could not wish for better advisors.

Finally, I thank my family and friends for their moral support and patience during the preparation of this thesis.

Contents

List of Figures

List of Tables

1. Introduction

Today, software systems must often be developed in a large variety of variants to meet the requirements of different customers. Software Product Line Engineering is a paradigm of software development in which multiple products that share a common set of development artifacts are developed simultaneously [39, 125]. Each product of a product line is considered as a combination of features. A feature is a user-visible characteristic of a software system [81].

Software product lines are increasingly used for the development of safety-critical and mission-critical systems in which software errors cannot be tolerated [24, 159]. For single-system engineering, the incorporation of formal specification and verification techniques into the development process has been shown as a possible approach to achieve the desired correctness [33]. A challenge of ongoing research is to adapt formal verification techniques to cope with the challenges that arise from the variability of software product lines [151].

Each verification technique, by definition, relies on a certain specification technique to specify properties of a system and a certain modeling or implementation technique in which the system itself is represented. As a result, researchers have proposed several modeling and specification techniques with varying degree of built-in variability support. It becomes increasingly difficult for researchers to maintain an overview of the diverse modeling and specification approaches originating from different lines of research related to software product lines. Furthermore, it is not clear whether and how results regarding the handling of variability from a given line of research can be applied in other contexts. A goal of this thesis is to provide an overview of the state-of-the art in formal modeling and specification techniques for product lines.

However, the problem to handle variability, i.e., to reduce development effort by taking advantage of commonalities between products, is not limited to models and specifications. Instead, similar problems arise for all kind of development artifacts related to all phases of the development process. For instance, in the context of product-line implementation techniques variability mechanisms have been extensively studied [151].

The underlying assumption of this thesis is that there are common principles of variability that can be discovered and investigated independently of many details of specific artifact types. Accordingly, we think that research results from a given area such as implementation techniques can be transfered to other kind of artifacts. We aim to identify the common principles of variability underlying modeling, specification, and implementation techniques to support future research. We envision the development of fundamental variability concepts that can be applied to all kinds of development artifacts. On the way towards this goal, we hope that variability concepts can be fruitfully transfered between different lines of product line research.

As a first step towards our goal to unify research regarding variability concepts for product lines, we propose a taxonomy of product-line representations that focuses on the mapping between representation and sets of products. Novel to our view on variability is that we do not primarily focus on the mechanisms to transform representations as typically done in existing research. For instance, research on implementation techniques often focuses on mechanisms used to derive products [151]. While mechanisms of product generation are indeed important for implementation artifacts, this is not necessarily the case in other contexts.

We exemplify the usefulness of our taxonomy by considering research on formal modeling and specification techniques. We give an overview about the state of the art in formal modeling and specification of software product lines, and classify existing techniques from the literature according to our taxonomy. Furthermore, we show that the unifying point of view provided by this classification helps to recognize commonalities and differences that may be fruitful for future research by identifying possible directions for future research.

In this thesis, we make the following contributions to the research community:

- We present a taxonomy of product-line representations as a first step towards a unifying theory of variability.
- We survey the formal modeling and specification techniques of software product lines.
- We exemplify the suitability of our taxonomy to capture techniques from different lines of research in a unifying way by classifying formal modeling and specification approaches proposed in the literature according to our taxonomy.
- We discuss possible directions for future research that we have identified by considering the research area in terms of the classification.

Research Scope The survey presented in this thesis considers formal modeling and specification techniques that have been proposed for software product lines. It does not cover informal or semi-formal specification approaches. Furthermore, the focus lies on techniques that have been proposed as a means to model or specify software systems—it does not cover pure formalizations of concepts without obvious or explicitly mentioned practical application in the software development process. The survey covers only specification approaches that can be used to specify the behavior or properties of software systems—it does not cover approaches for pure variability modeling.

Structure of the Thesis In Chapter 2, we present the underlying concepts and terminology used in this thesis. In particular, we introduce the distinction between modeling and specification that is used to structure our literature survey. We present a general taxonomy of product-line representations that considers variability of development artifacts in a unifying way in Chapter 3. Based on our taxonomy, we survey and classify modeling techniques for software product lines in Chapter 4 and specification techniques for software product lines in Chapter 5. We exemplify the usefulness of the novel view on the research area provided by our classification by discussing possible directions for future research in Chapter 6. We discuss related work in Chapter 7 and conclude in Chapter 8.

2. Background

In this chapter, we introduce concepts and terminology used in the rest of this thesis. In Section 2.1, we give an overview of software product lines. We introduce formal methods in Section 2.2.

2.1 Software Product Lines

Mass-customization is a successful production paradigm in many industrial domains. It can be seen as a trade-off between mass production in which development costs are low but individual customer requirements cannot be satisfied and handcrafting in which all individual requirements can be satisfied but development costs are high. The idea of mass-customization is to assemble a product of reusable parts that can be combined in multiple ways to satisfy individual requirements. The reusable parts can be produced with relative low costs as they are reused between multiple products.

Software product lines apply the idea of mass-customization to software development [91]. Following the principle of mass-customization, the goal of software product lines is to achieve a trade-off between development costs and product diversity. Software product lines are used to develop software that is adapted to specific requirements of individual customers [39, 125]. Instead of developing software from scratch, individual products are developed from a set of reusable parts.

In the last two decades, feature-oriented development of software product lines has gained momentum. In this thesis, we focus on feature-oriented software product lines in which the whole development process is driven by the concept of features [7]. In the following, we give a brief overview of the underlying concepts.

Features The notion of a feature has multiple facets that are hard to capture completely in a single definition. We refer to the literature for a more thorough treatise of the notion [7]. We give a definition sufficient for the rest of this thesis that is adapted from the literature [81]:

Definition 2.1. *A feature is a characteristic of a software system relevant to a stakeholder.*

A typical stakeholder is a customer for which a software system is being developed. Without loss of generality, we assume this to be the case in the rest of this thesis. Customers typically express requirements in terms of what they expect a system to do or how something should be done. Each of these characteristics, following Definition 2.1, can be described in terms of features. The following example illustrates a small selection of possible features of a transaction management system:

Example 1. Features:
`Base`: The system provides basic functionality for transactions.
`Rescheduling`: Aborted transactions are rescheduled and executed again.
`Locking`: Transactions lock the database while performing operations.
`In_Private`: Transactions operate on a local copy of the database.
`In_Place`: Transactions directly operate on the database.

Products A customer might be interested in a database transaction system that includes a set of desired features. Such a selection of desired features is called a configuration [7]. To satisfy the requirements of the customer, a product that incorporates all features that belong to the desired configuration must be developed, or preferably generated automatically.

Technically, we must distinguish between the combination of features selected in the configuration and the actual software product that is to be developed. The configuration is an entity of the problem space, while the actual product is an entity of the solution space. However, it is often sufficient and convenient not to make this distinction if there is no risk of confusion. We follow this convention and give the following definition:

Definition 2.2. *A product is a combination of features.*

The following example illustrates the concept of a product as a combination of features:

Example 2. Products:
$\texttt{A} = (\mathit{Base},\ \mathit{In\_Private}, \mathit{Rescheduling})$
$\texttt{B} = (\mathit{Base},\ \mathit{In\_Place},\ \mathit{Locking})$

We observe that Product A shares the basic functionality, i.e., feature *Base*, with Product B but differs in the selection of additional features. The commonalities of a set of products are captured by the set of features they have in common. The differences are captured by the features in which they differ.

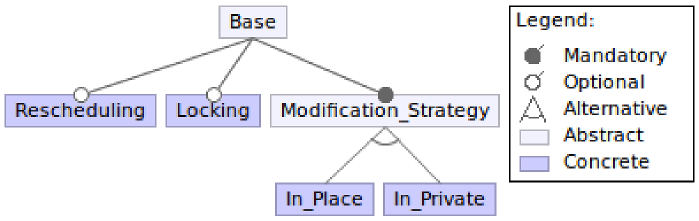

Figure 2.1: Feature Model of the Transaction Product Line

Product Line A software product line is a set of software systems that share common artifacts and are developed simultaneously [39, 125]. We can now define a product line by using the previously defined terminology, simply as a set of product. We always assume an underlying set of features to be given over which products are defined. We give the following definition:

Definition 2.3. *A product line is a set of products.*

We may not wish or cannot develop a product for all possible configurations. In our transaction example, feature *In_Private* and feature *In_Place* cannot be selected together because we cannot choose multiple modification strategies simultaneously. A configuration is called valid, if there is a corresponding product in the product line.

Feature Models Theoretically, a product line can be fully described by listing all its products. However, in practice it is useful to emphasize the variability of a product line by considering it as a set of features with rules to select a valid configuration from them. Typically, this is done by using a feature model [7].

A feature model defines rules about the selection of features to derive a valid configuration. The main rule is that if a feature is selected its parent feature must be selected, too. The root feature must always be selected. If a feature is mandatory, it must be selected if its parent is selected. If features are organized in an alternative-group, exactly one of them must be selected if the parent feature is selected. If features are organized in an or-group at least one of them must be selected if the parent feature is selected.

In Figure 2.1, we show a feature model for the transaction product line. Feature *Transaction* is the root of the model. It is included in all products. The feature *Modification_Strategy* is an abstract feature, i.e., it serves merely as a means to structure the diagram and is not directly mapped to a specification—as mandatory child of the root feature it must always be selected. Feature *In_Place* and feature *In_Private* are alternatives, i.e., exactly one of them must be selected to obtain a valid feature selection. Feature *Rescheduling* is optional.

So far, we have seen that features are a suitable means to capture commonalities and differences between products of a product line. The goal of feature-oriented software development is to enable reuse of development artifacts, such as source code, on the level of features by developing techniques that consider variability. Ideally, products can be generated automatically from the code base. For this purpose, several implementation techniques have been proposed [7]. We give an introduction to annotation-based implementation techniques in Section 2.1.1 and to composition-based implementation techniques in Section 2.1.2.

2.1.1 Annotation-Based Implementation Techniques

In annotation-based implementation techniques, the code of all features is contained in a single code base [7]. The mapping between parts of the code and products is defined by means of annotations that refer to features or combinations of features. Products are generated by removing code that does not belong to features of a given configuration.

A preprocessor manipulates the source code before compilation [7]. The mapping between code and features can be achieved by wrapping the code for a given feature into conditional-compilation directives that depend on the selection of that feature.

The use of preprocessors is a widely used annotation-based implementation technique [7]. Preprocessors are available for several languages [67, 116, 124]. A study that explored 40 open-source projects found that conditional compilation by means of preprocessors have been used in all projects [104]. A reason for the widespread use of preprocessors is that they are easy to use and widely available.

A disadvantage of preprocessors is that the code for a given feature is scattered across the code [7]. Thus, there is no separation of concerns regarding features. To overcome this limitation, concepts to visualize all code that belongs to a feature or a set of features in separate views have been proposed [8]. This approach called virtual separation of concerns helps developers to understand the source code because they can look at features in a modular fashion [8]. Another technique to improve the usability of preprocessors is to highlighted code that belongs to each feature with a certain background color to help developers with program comprehension [58].

2.1.2 Composition-Based Implementation Techniques

In composition-based implementation techniques, the source code is encapsulated into modules that are mapped to features [7]. Individual products are derived by composing modules implementing the desired features. In the following, we present the composition-based implementation techniques feature-oriented programming, aspect-oriented programming, and delta-oriented programming.

Feature-Oriented Programming Feature-oriented programming is a composition-based development technique in which a product line is decomposed into features [128]. The decomposition mechanism used to implement features is based on collaboration-based design that has been proposed as an extension of object-oriented software development [157]. A collaboration is a set of classes or parts thereof that interact to achieve a certain function, i.e., it plays a certain role. For example, a transaction system, could be implemented as a collaboration of a transaction class, and a class representing the scheduler. A collaboration representing locking functionality could be implemented as parts of both classes, e.g., in the form of specific methods. In this case, the individual classes play multiple roles, i.e., different parts of them are part of different collaborations.

In feature-oriented programming, each feature is mapped to a certain collaboration encapsulated into a module [7]. Each product can be derived by composing the desired feature modules, i.e., by merging the corresponding collaborations into a single system.

The main advantage of feature-oriented programming is the modularization of features [7]. In contrast to object-oriented programming, the decomposition by means of feature modules supports to modularize cross-cutting features, i.e., features that effect multiple locations of the system [147]. Furthermore, it is based on uniform composition principles that are easy to understand but limit the granularity of feature refinements to the level of methods [7].

Aspect-Oriented Programming Aspect-oriented programming is an extension of object-oriented programming in which cross-cutting concerns can be modularized [86]. A cross-cutting concern is a unit of functionality that effects different locations in the design of a system. In our database example, feature *Locking* is a cross-cutting concern as it effects parts of multiple classes. In contrast to feature-oriented programming, which allows to modularize cross-cutting concerns by means of collaborations that can be composed, aspect-oriented programming separates a system into a base program, and a set of encapsulated aspects.

The base program contains a set of join points, locations at which code can be possibly inserted [86]. Each aspect contains functionality in the form of advice. An advice is a set of instructions that is woven into the base program at certain join points. The set of join points for a specific advice is specified by an expression that identifies a set of join points in the base program, a pointcut.

A well known characterization by Filman and Friedman, lists quantification and obliviousness as the fundamental principles of aspect-oriented programming [61]. Quantification describes the ability to select a set of locations that can be scattered within the base program in the form of join points. Obliviousness means that the base program is developed conventionally without considering possible extensions in the form of aspects in the design.

Aspect-oriented programming can be used to implement software product lines in different ways [6, 9, 10, 60, 82, 112]. A simple approach is to map each aspect to a feature. More complex mappings are possible and can be useful in practice. For simplicity, we assume a one-to-one mapping between features and aspects in the rest of this thesis.

In contrast to feature-oriented programming, the composition mechanism of aspect-oriented programming is more flexible and allows fine grained extensions [7]. In particular, the support for quantification increases the ability to encapsulate cross-cutting features. This higher expressiveness comes with the disadvantage that developers need to learn more complex language extensions.

Delta-Oriented Programming In delta-oriented programming, a product line is implemented as a base module and a set of delta modules [134]. A delta module encapsulates a transformation of the base module and can add or delete classes or class members or override existing ones. The ability to delete parts of a base program is a fundamental difference compared to feature-oriented programming. Furthermore, delta modules are typically mapped to expressions of features rather than single features. For instance, a given delta module may be applied if a certain combination of features is contained in a product.

A key characteristic that distinguishes delta-oriented programming from feature-oriented programming and aspect-oriented programming is that it allows to remove parts of the base product [7]. This supports a way of development in which the base system contains all features and products are derived by removing functionality that belongs to undesired features.

2.2 Formal Methods

In this thesis, we present a survey of modeling and specification techniques for software product lines. Typically, these techniques are adaptations of techniques known from single system engineering. In the following, we give a brief overview about selected modeling and specification techniques from single system engineering as far as necessary to understand the rest of this thesis. In Section 2.2.1, we give a brief overview of formal methods and define our terminology regarding models and specifications. We introduce selected modeling techniques in Section 2.2.2 and specification techniques in Section 2.2.3 which we assume to be known by the reader in the rest of this thesis.

2.2.1 Overview

Computers play an increasingly important role in modern society. Important infrastructures such as energy, communication, medicine, finances, security, or transportation rely on the correctness of software-intensive systems. In the case of mission-critical and safety-critical systems, failures may have dramatic consequences as shown by known examples [160].

Failures of mission-critical systems may have substantial financial consequences. For instance, a defect in the floating point division unit of the Intel Pentium II has caused a loss of about 475 million US dollars [20]. It is said that the failure of the online ticket reservation system of a large airplane company would lead to its bankruptcy within 24 hours [20].

Failures of safety-critical systems even threaten human lives. For instance, the catastrophic crash of the Ariane-5 missile in 1996 has been caused by a defect in the software [55]. Similarly, a defect in the software of the radiation therapy machine Therac-25 has caused the death of six cancer patients because they have been exposed to an overdose of radiation [100].

Software Engineering aims to allow developers to develop systems with a minimal amount of defects. One approach to achieve this goal is the use of formal methods. Formal methods incorporate mathematically precise formalisms, languages, techniques, and tools to model, specify and verify systems [33]. In 1962, McCarthy formulated the idea to use computer to verify the correctness of programs, i.e., to establish that the system under consideration possesses certain properties [110]. Since then, formal specification and verification techniques have successfully been applied in domains such as traffic control [26], microprocessors [68, 113], electronic cash systems [145, 162], flight control [23], and for other safety-critical systems such as the Maeslant Kering (a movable barrier protecting the port of Rotterdam from flooding) [155].

A widely used technique to ensure correctness of software is testing, in which software is executed with a set of given inputs called test cases. Thus, testing verifies whether the system behaves correctly for a given subset of all possible execution paths. In contrast, model checking is an automatic technique in which all states of a system are exhaustively explored in a brute-force manner [20]. With model checking it can be shown that the system truly fulfills a property because it considers all possible execution paths. Another verification technique is theorem proving, in which the correctness of a system is deductively shown [22, 139].

All verification techniques require a representation of the system, i.e., a model or implementation, and a specification of properties that the system is expected to possess. Experience in industrial projects indicates that the process of modeling and specifying a software system itself already helps to identify defects and inconsistencies [94, 161].

In the following, we introduce the terminology used in this thesis regarding models and specification. In this thesis, we distinguish between models as descriptions of the behavior of a system and specifications as descriptions of properties the system is expected to fulfill.

A model is a description of behavior of a system. The focus lies on *how* the system is supposed to do something. A characteristic of models is that they are constructive, i.e., they can be used as a foundation to derive an implementation. We define the term as follows:

Definition 2.4. *A model is a description of the behavior of a system with a mathematically defined semantics.*

A specification is a description of properties of a system. The focus lies on *what* the system is supposed to do, opposed to how it should do something.

Definition 2.5. *A specification is a description of properties of a system with a mathematically defined semantics.*

To exemplify the difference between models and specifications, we provide an example:

Example 3. A specification states if the database is in a consistent state before execution of a transaction, then it will be still in a consistent state after its execution.

A model provides an algorithm to be performed, e.g., how a transaction performs operations, how it writes to a log file, how it checks for consistency, and how it reverts the changes if necessary.

Note that the level of abstraction could have been chosen arbitrarily in both cases. On the one hand, a specification may also define what consistency means and go into more detail about what kind of assertions must be fulfilled in order for a database state to be considered consistent. On other hand, the model could be operate on a more abstract level, omitting details about how to write to a log file.

2.2.2 Modeling Techniques

This thesis includes a survey on modeling techniques for product lines. In many cases, it is not neccessary to understand all characteristics of these modeling techniques in detail as our focus lies on the representation of variability. However, a siginificant part of the modeling techniques for product lines are based on transition systems. Thus, we give a brief introduction to transition systems.

Transition Systems Transition systems are a widely used formalism to model systems [20]. A transition system can be seen as a directed graph in which nodes represent states of a system and edges represent transitions between states. Edges can be labeled with names of actions to describe the meaning of a transition. We give the following definition adapted from Baier and Katoen [20]:

Definition 2.6. *A transition system TS is a tuple $(S, Act, \rightarrow, I, AP)$ where*

- *S is a set of states,*
- *Act is a set of actions,*
- *$\rightarrow \subseteq S \times Act \times S$ is a transition relation,*
- *$I \subseteq S$ is a set of initial states.*

$$\begin{tikzcd}
 & & & & \text{aborted} & \\
 {} \arrow[r] & \text{inactive} \arrow[r, "\text{activate}"] & \text{running} \arrow[r, "\text{EOT}"] & \text{ready} \arrow[r, "\text{commit}"] \arrow[u, "\text{abort}"] & \text{committed}
\end{tikzcd}$$

Figure 2.2: Transition System of a Product of the Transaction Product Line

A transition starts in some initial state $s_0 \in I$. In each step of execution, the system evolves according to the transition relation, i.e, a transition from the current state is selected nondeterministically and the system evolves to the next state until a state without outgoing transitions is reached.

Example 4. Figure 2.2 shows an example of a transition system of a single product of the transaction product line. A transaction starts in the state *inactive*. After activation it reaches the state *running*, in which it performs operations on the database. When the end of transaction (EOT) is reached, the transaction either gets aborted or committed depending on whether the performed operations preserve the consistency of the database.

This example also exemplifies that transition systems can be used to model systems on an arbitrary level of abstraction. In this case, details about how operations are performed or how consistency is checked are omitted. Instead, the choice between aborting and committing a transaction is modeled as an nondeterministic choice.

2.2.3 Specification Techniques

To specify properties that a system is expected to possess, various specification techniques based on different formalisms and languages exist. In the following, we give a brief introduction to the specification techniques that we expect the reader to be familiar with in the rest of this thesis.

Temporal Logics So far, we have seen how systems can be modeled with transition systems. Such transition systems are often used for model checking [20]. In model checking all states of a transition system are systematically explored to check whether the system fulfills certain properties. For this purpose, it is neccessary to express such properties formally. This is typically done by using special logics with semantics defined over transition systems. In the following, we introduce Linear Temporal Logic (LTL) and Computation Tree Logic (CTL), two temporal logics that are commonly used to express properties of transition systems.

Both LTL and CTL extend propositional logic with a notion of time [20]. The main difference between LTL and CTL is the nature of time. In LTL, time is linear, i.e., each moment in time has a single successor. In CTL, time is branching, i.e., each moment in time can have multiple alternative successors, resulting in a tree-like structure of time.

LTL extends propositional logic with two basic temporal operators, the next operator $\mathtt{X}$ and the until operator $\mathtt{U}$ [20]. Given any LTL formula α, the formula $\mathtt{X}\alpha$ holds at the current moment if and only if the property expressed by α holds in the next step. Given LTL formulas α and β, the formula $\alpha \mathbin{\mathtt{U}} \beta$ expresses that property α holds at the current moment if and only if there is a future moment in which β holds and α holds at all moment until that future moment. Based on these basic operators, further operators such as the operator $\mathtt{F}$ (eventually) and the operator $\mathtt{G}$ (always) can be derived. The formula $\mathtt{F}\alpha$ expresses that property α will hold eventually in the future. Similarly, the formula $\mathtt{G}\alpha$ expresses that property α will hold in all future moments.

As mentioned above, CTL is based on a branching notion of time. It contains the operator $\mathtt{A}$ (inevitably) and the operator $\mathtt{E}$ (possibly) to consider the tree-like structure of time. The formula $\mathtt{A}\alpha$ holds if property α holds on all paths starting from the current moment. Similarly, the formula $\mathtt{E}\alpha$ holds if property α holds on at least one path starting from the current moment. In CTL, the next operator and until operator must be preceded by operator $\mathtt{A}$ or operator $\mathtt{E}$ to obtain a valid CTL formula. A precise definition of the syntax and semantics can be found in the literature [20].

The expressiveness of LTL and CTL differs in the sense that some properties can be expressed with LTL but not with CTL and vice versa. For a comprehensive comparison of LTL and CTL regarding their suitability for specification and verification we refer to the literature [20, 158]. For this thesis, it suffices to know that both logics are commonly used to specify properties in the context of model checking [20].

Design by Contract Design by contract is a technique in which object-oriented software is specified by means of method contracts and invariants [111, 115]. The underlying principle of design by contract is based on assertions. An assertion is a property that is expected to hold at a certain point of a program's execution. If stated in a formal language, assertions can be checked automatically to verify that a given program fulfills the desired properties. In runtime assertion checking, assertions are compiled into a program and violations are reported during execution [34].

Design by contract applies the idea of assertions to object-oriented programming by introducing method contracts and class invariants. A method contract can be seen as an agreement between a method and its caller. The contract consists of a precondition

```
class Transaction {
...
 /*@
 requires operations != null;
 ensures operations.isEmpty();
 @*/
 void run(){ ...}
```

Figure 2.3: Method Contract of the Transaction Product Line in JML

and a postcondition. A method relies on the properties stated by the precondition, the caller relies on the fulfillment of the precondition after method execution. Invariants state properties that hold in every publicly visible states of a class, i.e., for and after calls to public methods.

The Java Modeling Language (JML) is an extension of Java that supports design by contract [31, 97]. In JML, specifications are embedded into special Java comments as shown in Figure 2.3. The precondition, denoted by keyword *requires*, states that the list of operations must be initialized before calling method *run*. The postcondition, denoted by keyword *ensures*, states that the list of remaining operations must be empty after execution. For JML, there exists tool support for run-time assertion checking, static analysis, theorem proving, and for the generation of documentation or test cases [28].

3. Variability in Product-Line Representations

i

Our long term goal is to develop a unifying theory of variability that can be applied to all kind of artifacts in all phases of product line engineering. To achieve this goal, it is necessary to consider both commonalities and differences between different types of development artifacts regarding their representation of variability. By identifying commonalities, it becomes possible to identify possibilities to transfer research results from a given context to other contexts. By considering the differences, it becomes clear in which cases this transfer is not directly possible and what kind of adaptations might be necessary.

In the following, we propose a taxonomy of product-line representations that focuses on the mapping between representation and sets of products. We introduce the taxonomy in Section 3.1 and exemplify the concepts by discussing its relationship to the well known area of implementation techniques in Section 3.2.

3.1 Taxonomy of Product-Line Representations

We propose a taxonomy of variability concepts for different kind of development artifacts that play a role in product-line engineering, such as implementation, models, specifications including test cases, or documentation. The development of the taxonomy has been guided by the observation that all kinds of artifacts are means to represent certain characteristics of a system. For instance, models and implementations represent behavior of a system, and specifications represent properties that a system is expected to fulfill. We introduce the notion of a product-line representation to abstract from details from specific types of artifacts and to focus on their commonalities.

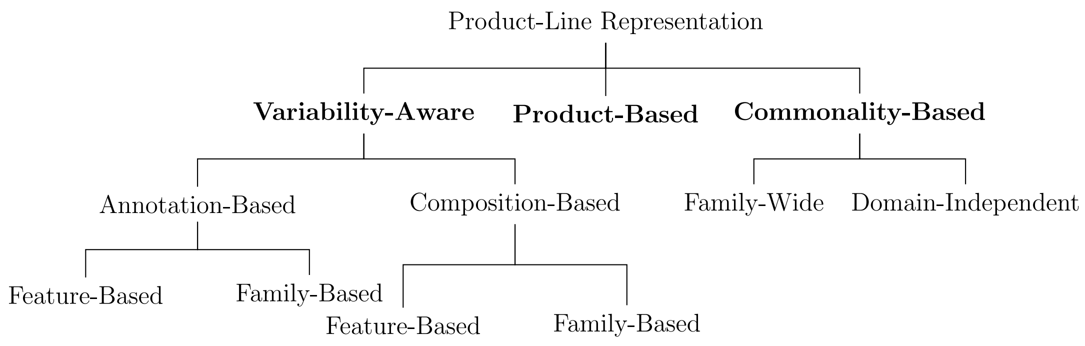

Figure 3.1: Taxonomy of Product-Line Representations

Variability on the level of implementation has been extensively studied in the context of feature-oriented development of product lines [151]. It appears natural to assume that the common distinction between annotation-based and composition-based implementation techniques, as presented in Chapter 2, can also be applied to other artifacts such as models and specifications. In this thesis, we will see that this is possible, but does not suffice to describe the full spectrum of modeling and specification techniques. As our goal is to support collaboration between existing lines of research, we have aimed to incorporate these known concepts of variability into a more general taxonomy, using more general definitions that can be applied independently of a specific research context such as implementation techniques. In the following, we present our taxonomy of product-line representations.

Product-Line Representation A product-line representation represents characteristics such as behavior or properties of a product line. In this thesis, we will exemplify the notion of product-line representations by classifying modeling and specification techniques. We also discuss its relationship to known concepts from implementation techniques. However, we intend the taxonomy to be applicable for further representation types such as informal specifications and models, documentation, and requirements documents. We define a product-line representation as follows:

Definition 3.1. *A **product-line representation** is a set of related development artifacts, also called modules, that as a whole represent certain characteristics of the products of a product line together with a mapping between modules or parts thereof and sets of products.*

Each part of the representation represents characteristics of a particular set of products. We consider this relationship to be defined in terms of a mapping between modules of the representation or parts thereof and sets of products. Our taxonomy distinguishes different classes of representations by considering different characteristics of this mapping function.

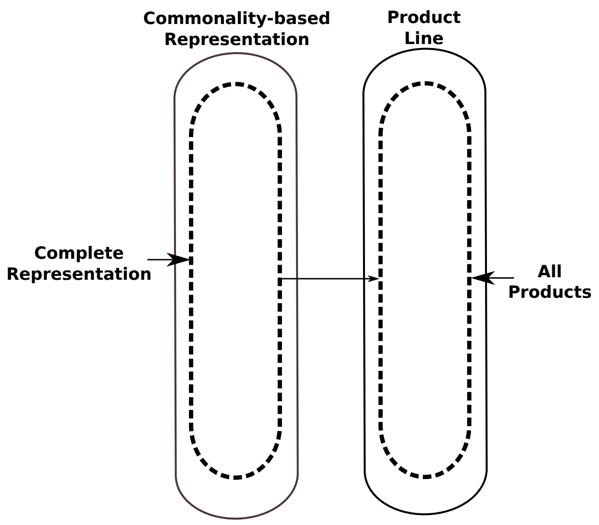

Figure 3.2: Commonality-Based Product-Line Representation

Commonality-Based Representation Generally, it can be said that each part of a representation represents the commonalities of a certain set of products. If the complete representation represents characteristics of all products of one ore more product lines, we say it is commonality-based. We give the following definition:

Definition 3.2. *A* ***commonality-based*** *representation is a product-line representation in which all parts of the representation are mapped to each product of one ore more product lines.*

In Figure 3.2, we illustrate the concept of commonality-based representations. The source of the mapping is the complete representation and the target of the mapping is a specific set of products. We distinguish between two types of commonality-based representations. A family-wide representation represents commonalities of a single product-line, a domain-independent representation represents commonalities of several product-lines. We give the following definition:

Definition 3.3. *If a commonality-based representation is mapped to all products of a single product line, it is* ***family-wide****. Otherwise, i.e., if it is mapped to all products of several product lines, it is* ***domain-independent****.*

Figure 3.3 shows the concept of family-wide product-line representations. While our taxonomy is about concepts related to the development of software product lines, we use the simple analogy of an ice cream product line to illustrate these concepts. The family-wide representation represents the common shape of all ice cream cones in the ice cream product line. In Figure 3.4, we illustrate the concept of domain-independent product-line representations. In this case, the target of the mapping are not all products

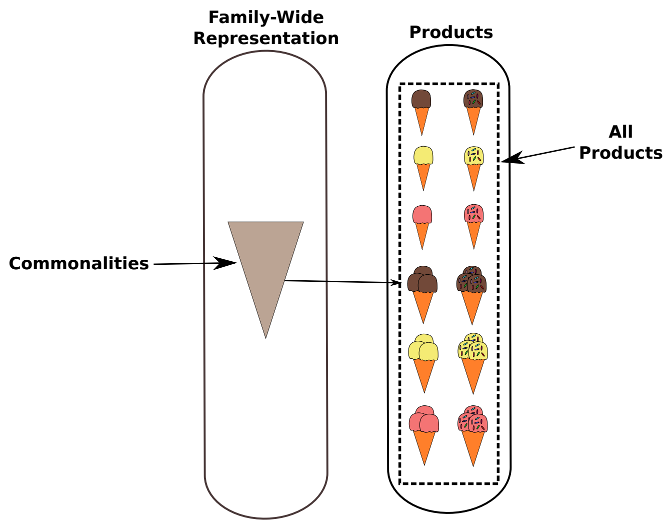

Figure 3.3: Family-Wide Product-Line Representation

of a single product-line but all products of two product lines. The additional product-line is similar to the previous ice cream product-line except that the ice cream is served in cups rather than in cones. The representation represents the common characteristic of both product lines that ice cream is served frozen.

Product-Based Representation A straight-forward way to represent a product-line is to represent characteristics of each product in a separate module. Such a representation is called a product-based product-line representation. The modules of a product-based product-line representation are also called product representations because each module represents a specific product. We give the following definition:

Definition 3.4. *A **product-based** representation is a product-line representation in which each module is mapped to a separate product.*

In Figure 3.5, we illustrate the concept of product-based representation in terms of the ice cream product line. The representation consist of separate modules. Each module represents characteristics of a single product. In our example, the first module represents a small ice cream cone with a small serving of chocolate ice cream without sprinkles. The second product represents a small vanilla ice cream cone with sprinkles. The third product represents a large chocolate ice cream cone with sprinkles.

Variability-Aware Representation Commonality-based representations do not consider variability, and product-based representations only consider variability in a trivial sense by representing each product of a product line separately. However, in the context of product lines variability-aware representations are often desired. We give the following definition:

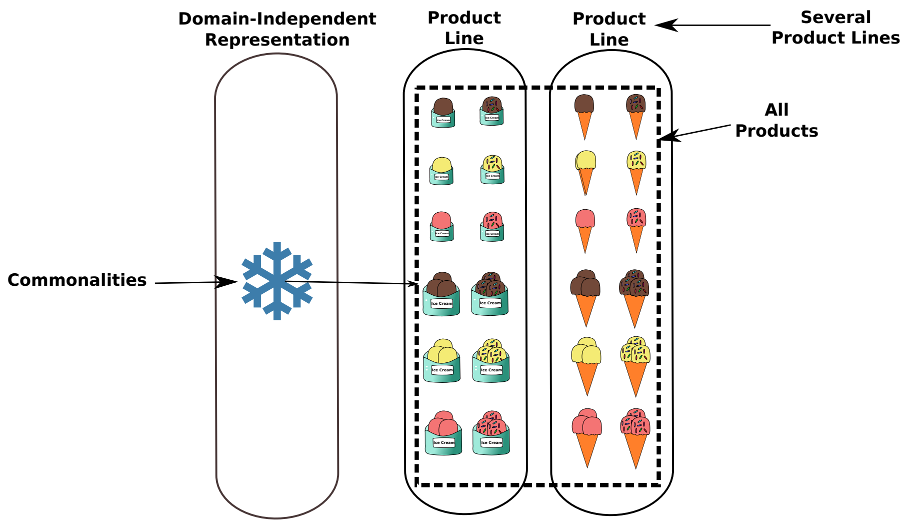

Figure 3.4: Domain-Independent Product-Line Representation

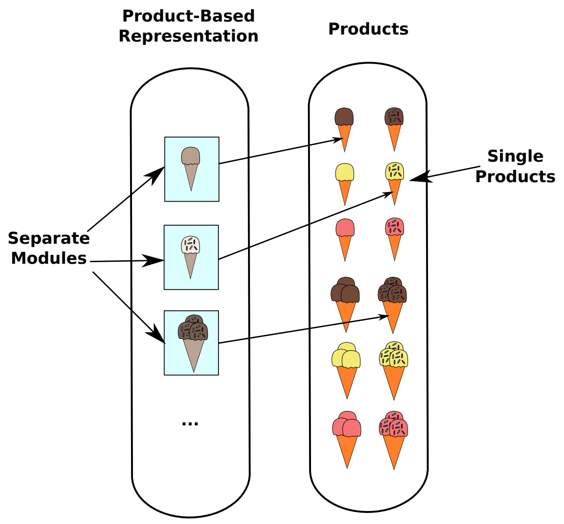

Figure 3.5: Product-Based Product-Line Representation

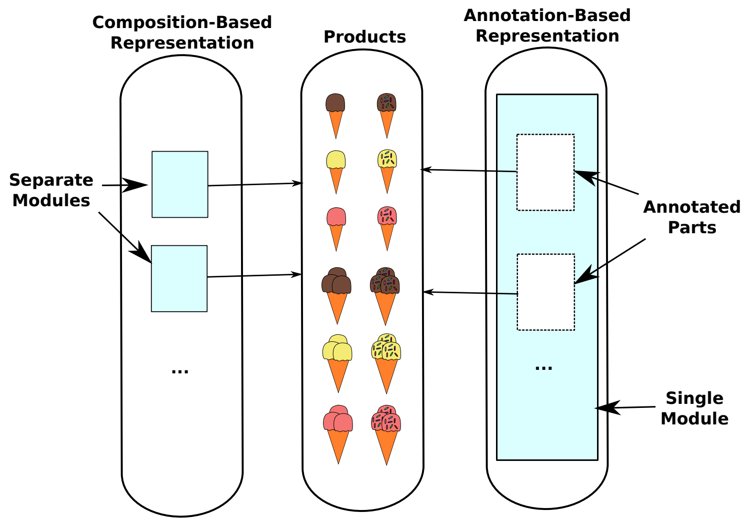
Composition-Based Representation
Products
Annotation-Based Representation
Separate Modules
Annotated Parts
Single Module
...
...

Figure 3.6: Variability-Aware Product-Line Representations

Definition 3.5. *If each part of a product-line representation is mapped to an arbitrary set of products, i.e., if it is not commonality-based or product-based, the representation is* ***variability-aware****.*

We distinguish two types of variability-aware representations: composition-based and annotation-based. Our notion of composition-based and annotation-based representations focus on the representation of variability rather than on mechanisms to derive products as the naming may suggest. However, we have chosen these names in order to emphasize the role they play in known concepts such as composition-based and annotation-based implementation techniques. We further explain this relationship in Section 3.2.

Figure 3.6 shows the concepts of composition-based and annotation-based representations. A composition-based product-line representation consists of multiple separate modules. Each module consists characteristics of a specific set of products. This is in contrast to product-based representations in which each module is mapped to single products rather than sets of products. An annotation-based product-line representation consists of a single module that represents the product-line. This module is divided in several parts, each part representing characteristics of a set of products. Inspired by its application in annotation-based implementation techniques, we consider these parts as annotated parts. However, the notion of annotation should be seen in a very broad sense. Technically, it is only essential that the source of the mapping function are parts of a module rather than separate modules. We give the following definitions:

Definition 3.6. *A variablity-aware representation is* ***composition-based****, if it consists of separate modules that are mapped to sets of products.*

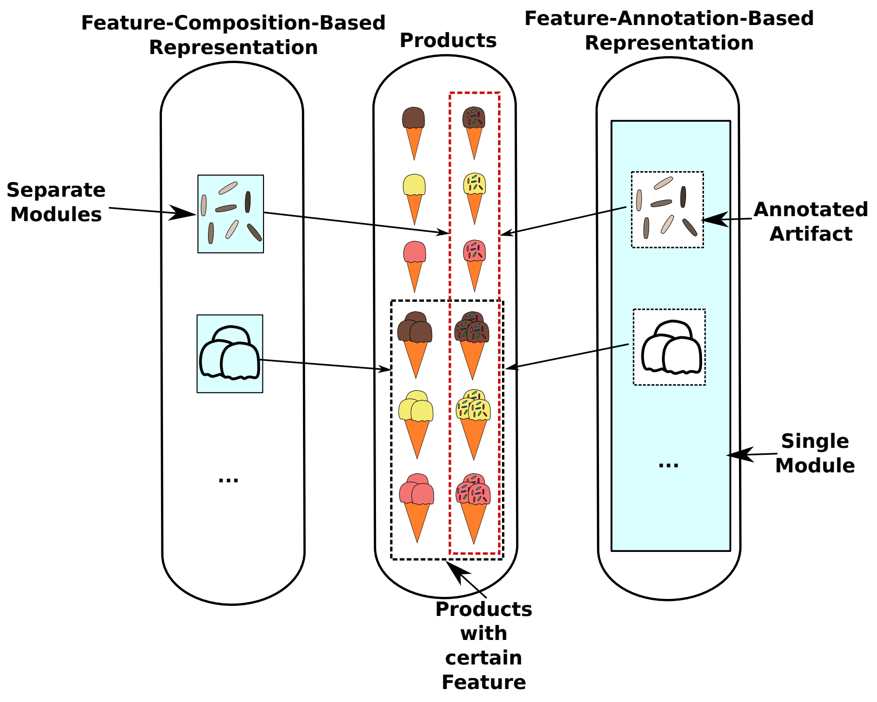
Feature-Composition-Based Representation
Products
Feature-Annotation-Based Representation
Separate Modules
Annotated Artifact
Single Module
...
...
Products with certain Feature

Figure 3.7: Feature-Based Product-Line Representations

Definition 3.7. *A variability-aware representation is* ***annotation-based****, if the representation mapping maps parts of modules to sets of products.*

The distinction between composition-based and annotation-based product-line representations refers to the source of the mapping between representation and products. We further distinguish between feature-based and family-based representations depending on the target of this mapping. We give the following definitions:

Definition 3.8. *A* ***feature-based*** *(product-line) representation, is a variability-aware representation in which each part of the representation is mapped to a set consisting of all products that share a certain feature.*

Definition 3.9. *A* ***family-based*** *(product-line) representation is a variability-aware representation in which each artifact of the representation is mapped to an arbitrary set of products.*

As the distinction between feature-based and family-based representations is orthogonal to the distinction between composition-based and annotation-based representations, it is possible to combine these concepts arbitrarily. We denote these combinations as feature-composition-based, family-composition-based, feature-annotation-based, and family-annotation-based.

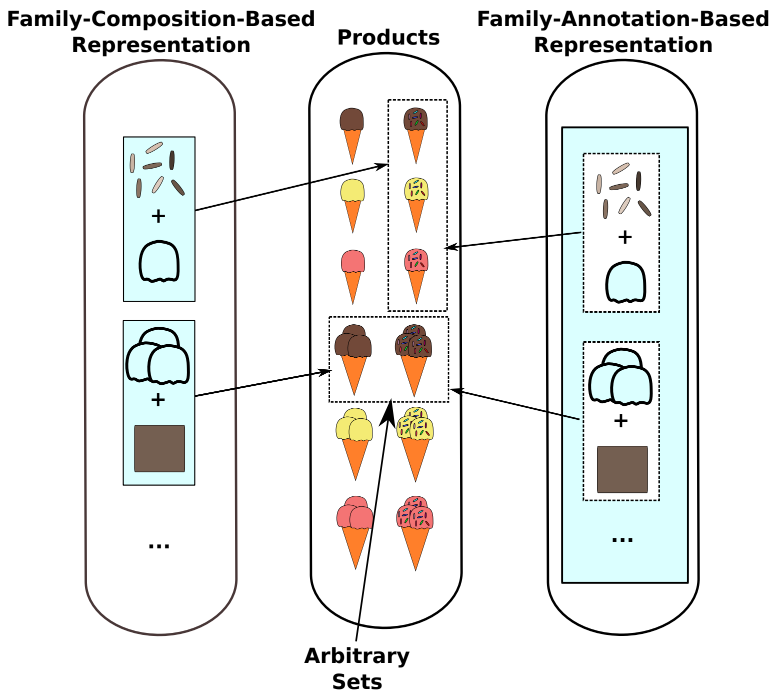
Family-Composition-Based Representation
Products
Family-Annotation-Based Representation
+
+
...
Arbitrary Sets

Figure 3.8: Family-Based Product-Line Representations

We illustrate the concept of feature-based representations in Figure 3.7. In feature-composition-based representations, each module represents characteristics related to a specific feature, i.e. it is mapped to all products containing this feature. The first module in our example refers to the characteristic of having sprinkles and is mapped to all products with sprinkles. The second module represents the characteristic of having large serving size and is mapped to all products with large serving size. Similarly, individual parts of modules are mapped to products in feature-annotation-based representations.

The concept of family-based representation is illustrated in Figure 3.8. In family-composition-based representations,, we have separate modules as the source of the mapping. However, each module now targets arbitrary sets of products. The first module represents characteristics of all products that have sprinkles and small serving size. The second module represents characteristics of all products with large serving size and chocolate flavor. Similarly, in family-annotation-based representations arbitrary products represented, but now by parts of a single module.

In summary, we have introduced product-based, commonality-based, and variability-aware product-line representations. Table 3.1 gives an overview about the taxonomy in terms of the characteristics of the mapping function. Some representation classes specify a certain number of modules. This is the case for composition-based and annotation-based representations in which the representation consists of multiple modules or a single module respectively. Some classes specify a certain source of the mapping.

	#Modules	Mapping Source	Mapping Target
Product-based	**-**	**module**	**single product**
Commonality-based	**-**	**representation**	**-**
Family-wide	-	representation	all products of single product line
Domain-Independent	-	representation	all products of several product lines
Variability-aware	**-**	**-**	**set of products, (not single, not same)**
Composition-based	multiple	module	-
Annotation-based	single	parts of module	-
Feature-based	-	-	all products with certain feature
Family-based	-	-	arbitrary sets of products

Table 3.1: Overview of Product-Line Representation Taxonomy

3.2 Product-Line Representations in Implementation Techniques

We have introduced a taxonomy of product-line representations with a focus on variability handling. In this section, we explain how the taxonomy of product-line representations can be used to classify development techniques and and exemplify this by using the example of implementation techniques. In particular, we show that the resulting classification is a refinement of the existing distinction between annotation-based and composition-based implementation techniques. For this purpose, we define the notion of a product-line development technique. We give the following definition:

Definition 3.10. *A product-line development technique is a technique to develop product-line representations. In particular, it constitutes a specific notion of representation and specifies the mapping to products. A product-line development technique may further define transformations of the initially developed representation to derived representations.*

A product-line development technique is a generalization of techniques to develop product-line representations. Accordingly, the notion includes implementation techniques, modeling techniques, specification techniques and techniques to develop other kinds of representations.

A product-line development technique can be classified by considering the type of its developed product-line representation that is defined by the mapping between representation and products. For instance, we classify a development technique in which a product-based representation is developed as product-based. We exemplify this concept by discussing the classification of common implementation techniques for software product lines. We hope that this helps to understand the concepts of our taxonomy before we use it to classify modeling and specification techniques in particular for readers who are familiar with implementation techniques for software product lines. For an overview of implementation techniques, we refer to Section 2.1. In Figure 3.9, we

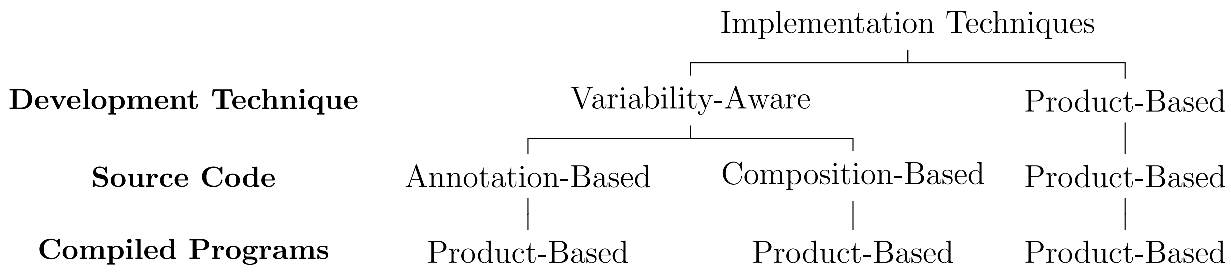

Figure 3.9: Product-Line Representations in Implementation Techniques

illustrate the concept of product-line implementation techniques in terms of the involved product-line representations. In variability-aware implementation techniques, a product-based representation is derived from a composition-based or annotation-based representation, respectively. Product-based implementation techniques, are techniques in which the source code for each product is developed separately. Common to all implementation techniques is that we are ultimately interested in the development of individual products, i.e., in a product-based representation on the level of compiled programs. Thus, commonality-based representations are not used for implementation techniques, because they do not contain enough information to derive individual products. However, such representations are common in specification techniques, e.g., when specifying the common properties of a product line, as we will show. In the following, we discuss variability-aware and product-based implementation techniques.

Variability-Aware Implementation Techniques In the literature, the distinction between composition-based and annotation-based implementation techniques is commonly used [7]. However, definitions typically focus on the derivation of products. In composition-based techniques products are composed. In annotation-based implementation techniques products are derived by removing parts of annotated artifacts. However, the focus of our taxonomy is more general. For instance, we aim to include development techniques that do not even include product derivation. Thus, we focus on the representation of the product line itself by considering an abstract mapping to products that does not necessarily implies that these products are actually derived from the representation. For this purpose, our taxonomy of product-line development techniques in terms of the underlying representations is appropriate as the meaning of the mapping function focuses on representation. In particular, it is a generalization of the traditional distinction between composition and annotation based implementation techniques.

In composition-based implementation techniques, we develop modules that are mapped to features and compose these modules to derive products [7]. In terms of our terminology, we transform a composition-based product-line representation into a product-based product-line representation. Feature-oriented programming and aspect-oriented programming are instances of composition-based implementation techniques, in which each module encapsulates the implementation of a single feature. Thus, feature-oriented programming and aspect-oriented programming are feature-composition-based implementation techniques. In contrast, delta-oriented programming allows developers to express modules with expressions over features. Thus each module can be mapped to arbitrary sets of products. Accordingly, we classify delta-oriented programming as a family-composition-based implementation technique. We also consider the use frameworks with plugin mechanisms as a composition-based development technique. If each plugin encapsulates a single feature, it is feature-based, otherwise family-based.

In annotation-based implementation techniques, we develop an annotation-based product-line representation and, again, transform it into a product-based product-line representation. This is typically, done by annotating parts of the source code that contains the implementation of all products with information about its mapping to products. The use of preprocessors and virtual separation of concerns are instances of annotation-based implementation techniques [7]. Products are derived by removing code that does not belong to a given configuration. Whether an annotation-based implementation technique is feature-based or family-based depends on the characteristics of the annotations. Only if each annotation can be mapped to single features and annotations cannot be nested, it is feature-based. Otherwise, it is family-based. Thus, typically annotation-based implementation techniques are family-based. The use of parameters can also be seen as an annotation-based technique, if the parameters are used to instantiate different products [7]. The use of parameters is family-based, as parts of code may conditionally depend on them in arbitrary ways.

Product-Based Implementation Techniques A further option to develop all products of a product line is to use conventional development techniques without support for automated product derivation. Typically, reuse between products is achieved by applying design patterns, or by adhoc-reuse of components [7]. In this case, we directly develop a product-based product-line representation. If we use techniques from single-systems engineering, e.g., design patterns, to develop a product line, we develop each product separately, i.e., each module encapsulates a separate product. Thus, the use of design patterns to develop product lines is a product-based development technique. Often, product-based implementation techniques are complemented by the use of version control systems to support the development of individual products [7].

Summary The goal of this thesis is to survey the representation of variability in modeling and specification techniques for product lines. We believe that the underlying variability mechanisms are largely independent of the specific artifact type. We have proposed a taxonomy of product-line representations that can be applied to different types of artifacts such as implementation, modeling, specification, or documentation. We have shown that implementation techniques for product lines can be characterized by considering the underlying representation type. We classify modeling techniques (Chapter 4) and specification techniques (Chapter 5) following the same approach.

4. Modeling Software Product Lines

In this chapter, we survey formal modeling techniques for software product lines regarding their strategy to represent variability. We classify modeling techniques according to our taxonomy as explained in Chapter 3. In Section 4.1, we present our classification approach for product-line models. We survey annotation-based modeling techniques in Section 4.2, composition-based modeling techniques in Section 4.3, and summarize in Section 4.4.

4.1 Classification of Modeling Techniques

Building models is a major part of any engineering discipline. In traditional engineering disciplines, models are commonly used to reason about properties of products before they are actually built. For instance, in architecture, small versions of buildings are created to give a vivid impression of the design before it is created. Models of automobiles and airplanes are tested in wind-tunnels to reason about their aerodynamic properties.

In software development, models are build to represent static properties such as the structure of a system and its components or to represent dynamic properties such as the behavior of a system or interactions between components and processes [108]. Accordingly, models are also used in the development of software product lines. The specific characteristic of product-line models is that multiple products that share large parts of their behavior must be developed simultaneously. Thus, modeling techniques that include a notion of variability have been proposed in the literature. In this thesis, we focus on formal models as introduced in Section 2.2.

As explained in Chapter 3, we classify each development technique, i.e., modeling and specification technique, according to its underlying product-line representations. In Section 3.2, we have seen that our taxonomy incorporates the existing notions of implementation techniques. In the following, we elaborate our motivation to classify modeling techniques similarly. For this purpose, it is helpful to consider the commonalities and

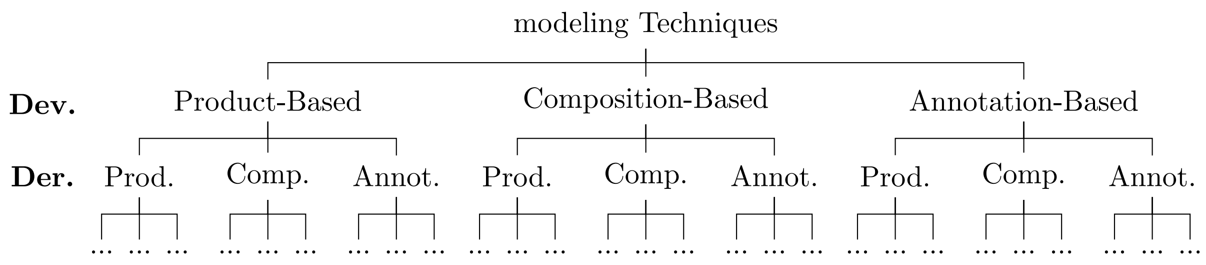

Figure 4.1: Product-Line Representations in Modeling Techniques

differences of modeling techniques and implementation techniques. A model typically abstracts from details that are not important to solve the given problem. In source code, these details must be implemented in order to derive an executable system. The distinction between a model and its implementation sometimes tends to be fuzzy. For instance, in model-based refinement methods an abstract model is successively refined to derive a concrete implementation [1, 2, 25, 80, 130, 144]. In model-driven development, source code is automatically generated from a given model [64, 140]. Some abstract models can be executed and executable source code may also operate on different levels of abstraction, e.g., a Java program could be seen as an abstract model of its corresponding bytecode program. The difference of model and implementation lies largely in the role they play in the development process, and less in technical differences.

For implementation techniques, the goal is to produce product-based representations. However, if we consider modeling techniques, we do not necessarily want to derive a product-based representation in all cases. As we will show in the following, there are cases, in which we want to derive an annotation-based or composition-based model, e.g., for analysis purposes. To take these cases into account, we have included the possibility to derive arbitrary product-line representations from the initially developed representation into the notion of a development technique. Figure 4.1 shows a taxonomy of modeling techniques. In modeling techniques, a product-based, composition-based or annotation-based product-line representation is developed. Depending on the modeling technique, further product-line representations can be automatically derived. Note that, as in implementation techniques, commonality-based representations are not considered. However, in the case of modeling it appears more likely that commonality-based representations might be useful in the context of modeling, e.g., for early phases of development. However, research for software product lines has not considered this possibility. A reason is that conventional modeling techniques from single systems engineering can be used to model the commonalities of all products. Thus, there is no research required to develop special development techniques.

Product-based and commonality-based techniques do not require the application of specific techniques, i.e., traditional development techniques as known from single-system engineering can be applied to develop such models [7]. In the following, we survey variability-aware modeling techniques for software product lines.

4.2 Annotation-Based Modeling

In the following, we survey annotation-based modeling techniques for product lines. We present annotation-based formalisms in Section 4.2.1 and annotation-based modeling languages in Section 4.2.2.

4.2.1 Annotation-Based Formalisms

An important class of annotation-based formalisms are variants of transition systems, which are often used for model checking of software product lines. Common to the formalisms based on transition systems is that variability is encoded by adding specific labels on transitions of a system. For an introduction to transition systems, we refer to Chapter 2. When modeling a product line with transition systems, each product of a product line differs in the set of transitions between states to reflect behavioral differences. Commonalities between products manifest in transitions that are common to all products. For the purpose of model checking, properties must be specified usually in a logic such as LTL or CTL. In this section, we focus on the models and the specification of properties is discussed in Chapter 5.

The most basic approach to use a transition system for the modeling of product-lines, that we call Family-TS, is to include all transitions of all products. This approach has been discussed by Fischbein et al., mainly to motivate the development of further extensions [62]. In this basic approach, no explicit labels are required. The implicit meaning of a transition can informally be seen as: There is at least one product that includes this transition.

A Family-TS can be used to reason about a limited set of properties. Consider the case of reachability, i.e., the question whether a certain target state T is reachable from a source state S. If the answer is negative, i.e., that T is not reachable from S, we can conclude that this property holds for all products. However, if the answer is positive, i.e., state T is reachable from state S, we cannot conclude that this property holds for any product because we do not know whether all involved transitions are contained in any single product, i.e., there is no explicit information about the mapping between transitions and products. The Family-TS in Figure 4.2 contains all transitions from all products of the transaction product line. We can conclude that successfully terminated transitions are not rescheduled and executed again. The reason is that there is no transition from state *done* to state *running*.

To include more information about the mapping between transitions and products, Fischbein et al. proposed Modal Transition Systems (MTS) to model software product lines [62]. A modal transition system is a transition system in which a transition is either mandatory (must) or optional (may). Again, all transitions of all products are included, but the additional labels reveal more information about the mapping between transitions and products. Informally, the meaning of a must-transition is that the transition is contained in all products, and the meaning of a may-transitions is that the transition is contained in at least one (but not all) products.

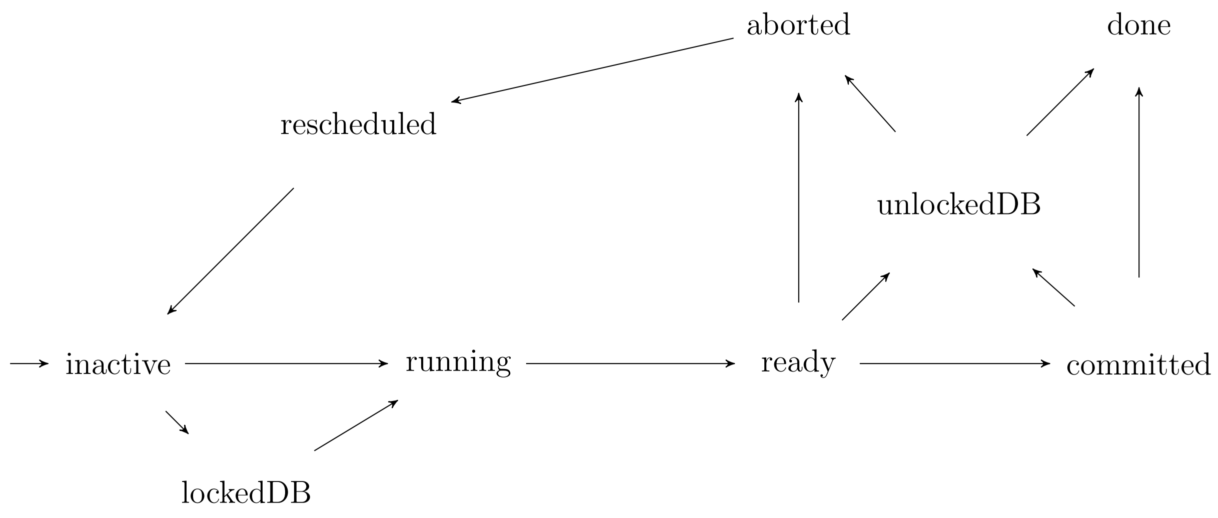

Figure 4.2: Family-TS of Transaction Product Line

In Figure 4.3, we show a MTS for the transaction product line. If we consider the example of reachability, there are now cases, in which we can reason about properties of products. The path from state *running* to state *committed* contains only must-transitions. We can conclude that state *committed* is reachable from state *running* in all products of the product lines. The path between state *lockedDB* and state *committed* includes exactly one may-transition. We can conclude that there exists at least one product in which this is true because a may transition is included in at least one product. However, if there are two or more may-transitions, we cannot conclude anything about products because we do not know whether there exists a product that contains all involved transitions.

A similar formalism based on I/O-automata instead of transition systems, proposed by Larsen et al., has the same properties regarding the mapping between annotations and products [93, 95]. Variability is also achieved by distinguishing between must-transitions and may-transitions. A variant of these variable I/O-automata has been proposed to model individual domain artifacts, i.e., the product line is composed of a set of domain artifacts and each artifact is annotated with variability information [96].

Generalized extended modal transition systems (GEMTS) have been proposed as a generalization of MTS [57]. In GEMTS, the modal annotations are applied to hyper-transitions, i.e., transitions to multiple states. Intuitively, this can be seen as a set of transitions with the same source. The informal meaning of a may-transition is that each product contains a certain number n of the annotated transitions. Analogously, a must transition means that each product contains at most n of the annotated transitions. The value of the number n must be defined additionally for each hyper-transition. While

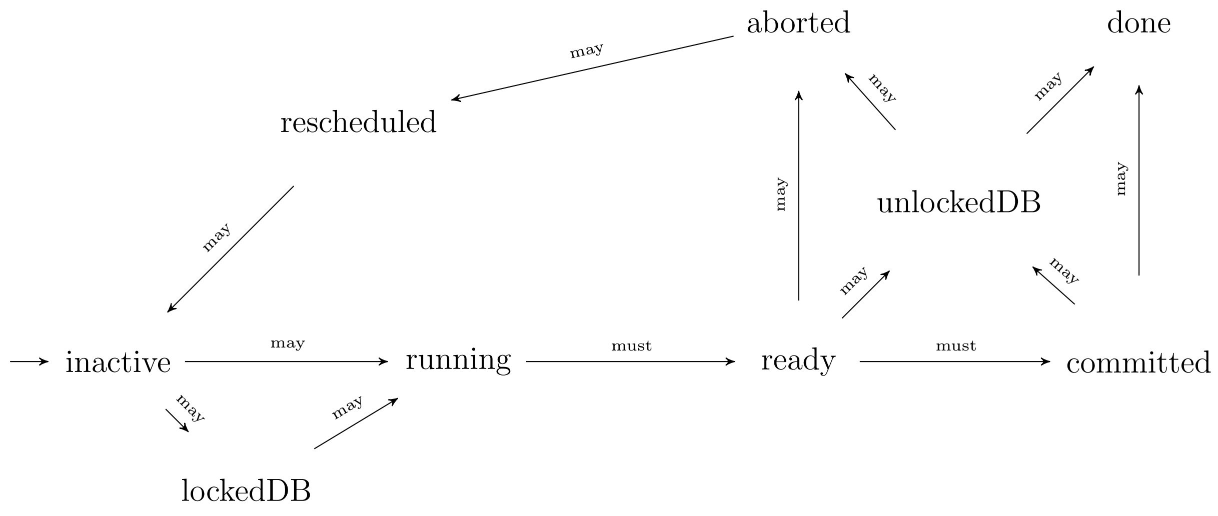

Figure 4.3: MTS of the Transaction Product Line

GEMTS in this sense are more expressive as MTS, they do not overcome the general limitation that relationships between individual annotations cannot be expressed.

So far, the presented approaches, Family-TS, MTS, and GEMTS, are family-annotation-based transition formalisms because each transition can be contained in arbitrary sets of products. However, the information about the mapping is not completely embedded into the model, which is their main limitation. For instance, this manifests in whether model checking algorithms are able to pinpoint specific products that violate a given property [36]. In MTS and GEMTS, specific labels express whether a certain transition is contained in all products or not. However, from such a model we cannot conclude the exact set of products that is target of the mapping. To cope with this limitation for analysis purposes, it is possible to enrich the specification technique for properties with additional variability information, as we will show in Section 2.2.

To solve this problem directly on the modeling level, Featured Transition Systems (FTS) have been proposed by Classen et al. [38]. The initial idea of FTS is to label each transition of an LTS with a feature, directly constituting a mapping between the transition and this feature. The meaning of a transition labeled with feature F can informally be read as: This transition is contained in the products that contain feature F. In Figure 4.4, we show a FTS for the transaction product line. As each transition can be labeled with only one feature, we have to introduce a separate feature *NonLocking* as an alternative to feature *Locking*. Otherwise, products containing feature *Locking*, would still contain transitions that circumvent the Locking mechanism, e.g., *inactive* $\rightarrow$ *running*. The reason is that features cannot remove transitions that are contained in the base feature.

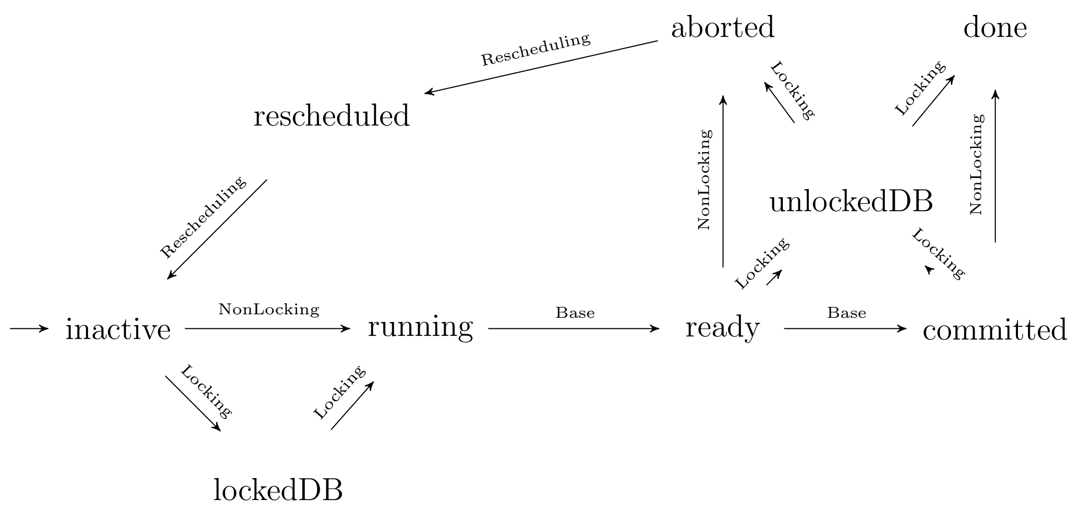

Figure 4.4: FTS of the Transaction Product Line

The main advantage of FTS is that the information of the mapping between transitions and products is completely revealed [38]. Consider the reachability property. If a path leads from the source state S to the target state T, we can conclude that this path is contained in exactly the products that contain all features that are used as transition labels on this path. However, FTS are limited by the set of possible mappings that can be expressed: Each transition is mapped to only one feature. To overcome this limitation, Classen et al. have extended FTS by allowing transitions to be labeled with feature expressions [36, 41, 42, 45]. A feature expression is a logical expression over the set of features, which can be used to denote arbitrary sets of products. In this case, we can replace feature *NonLocking*, by labeling the transitions with the logical negation of feature *Locking*, i.e., with $\neg Locking$. By doing so, we associate this transition with all products that do not contain feature *Locking*.

In Dynamic Software Product Lines (DSPLs), the set of features may change at runtime [69] in dependence of the environment. To model DSPLs, Adaptive Featured Transition Systems (A-FTS), have been proposed [40]. In contrast to FTS, A-FTS consider variability of both the system and of its environment by distuingishing between system features and environment features. System features can be fixed or adaptable. Environment features can change over time, and can be observed by the system to initiate reconfigurations. This is achieved by enabling or disabling system features. The main technical difference between FTS and A-FTS, is that the transition relation is given as a function that defines which transitions exist, to which products they belong, and the current state of the system configuration and environment configuration. However, the principle to use feature expressions to map transitions to products is the same.

Featured Timed Automata (FTA) are an adaption of FTS for real time systems [43]. In a real-time SPL, features cannot only change behavior but also make changes to timing constraints of actions. In Timed Automata (TA), this is reflected in the use of clock constraints, that define timing properties for actions such as the minimum or maximum time required for execution. FTA extend Timed Automata in the same way FTS extend, by using feature expressions, i.e., clock constraints are annotated with feature expressions to define the set of products that must satisfy them.

The idea to annotate transitions of states has also been applied to petri nets [117]. Feature petri nets have been proposed to model the behavior of product lines and for context-aware test models [118, 129]. In Feature Petri nets, the mapping between parts of the model and products is established in the same way as in FTS, i.e., by labeling transitions with feature expressions.

In model-based testing for produt lines, test cases for products are derived from a model of the system [54, 121]. Oster et al. have proposed to derive test cases for individual products from a single test-model [121]. Such a test-model can be seen as an annotation-based model from which a product-based representation in the form of test-cases is derived.

So far, all presented modeling formalisms are family-based with the exception of an early variant of FTS that has been replaced by a family-based variant. Simple hierarchical variability modeling (SHVM) has been proposed in which each variation point of a hierarchical model is mapped to exactly one feature [135]. It is a feature-annotation based technique in which the limitation of the mapping to target single features is supposed to provide benefits for compositional verification.

4.2.2 Annotation-Based Modeling Languages

The previously presented annotation-based formalisms can be used to model all products of a product line in a single model. However, they do not primarily aim to be directly used by engineers to specify systems but rather as a foundation for verification techniques. In particular, they do not provide means to structure the models. Thus, higher-level languages have been proposed for this purpose.

The language fPromela has been proposed as an input language for the SNIP model checker, an FTS-based adaption of the SPIN model checker [35]. Figure 4.5 shows an example of an fPromela model. A transaction is modelled as a process. A special type *features* is used to declare the set of features as variables. Inside the process *transaction*, guarded statements are used to annotate parts of the code with its mapping to features. In contrast to Promela, these guarded statements are denoted by keyword gd rather than if. The statements *lockDB*() and *unlockDB*() are only executed if feature *Locking* is considered to be present. The SNIP model checker interprets the model as an FTS, i.e., for the feature variables all possible values (true and false) are considered to reason about all products.

State Diagram Variability Analysis (SDVA) models have been proposed as an extension of FTS with means to express hierarchical sub-models to structure the model of a

```
typedef features {
bool Locking;
bool Rescheduling;
};

features f;

proctype transaction()
{
 ...

 gd :: f.Locking → lockDB();
 performOperations()
 gd :: f.Locking → unlockDB();

 ...

}
```

Figure 4.5: Example of Feature-Based Annotations in fPromela

product line [53]. This is achieved by the ability to refine states of a model by defining a separate model. In Figure 4.6, we show an example from our transaction product line. The state running is refined by means of a submodel that contains individual states for read and write operations. Note that a SDVA model is composed of a set of submodels. However, this dimension of composition is orthogonal to the mapping of features which is achieved by means of annotation. Thus, SDVA models are classified as family-annotation-based like FTS. The semantics of SDVA models are defined over FTS that are derived by flattening the hierarchical structure of the model. Thus, this modeling technique can be seen as involving a derivation step between to different types of annotation-based representations.

Similar to formalisms based on transition systems, other modeling languages based on different paradigms have been adapted to deal with variability.

Process-algebraic formalims have been proposed to model product lines. The ability to model product lines is achieved by considering special variability operators. Milner's calculus of communicating systems (CCS), PL-CCS, has been enriched with a variant operator as a means to implement variability [73, 133]. In the modeling language FLan, the mapping from model to sets of products is not achieved by means of annotations but given implicitely by treating features as first class entities [148]. FLan is a feature-oriented language, in which variability of processes can be defined by using operators such as alternatives. The main difference compared to PL-CCS is that Flan incorporates capabilities to model relationships between features, as typically done in a separate feature model. We classify both Flan and PL-CCS as family-annotation-based because each part of the specification may belong to arbitrary sets of products.

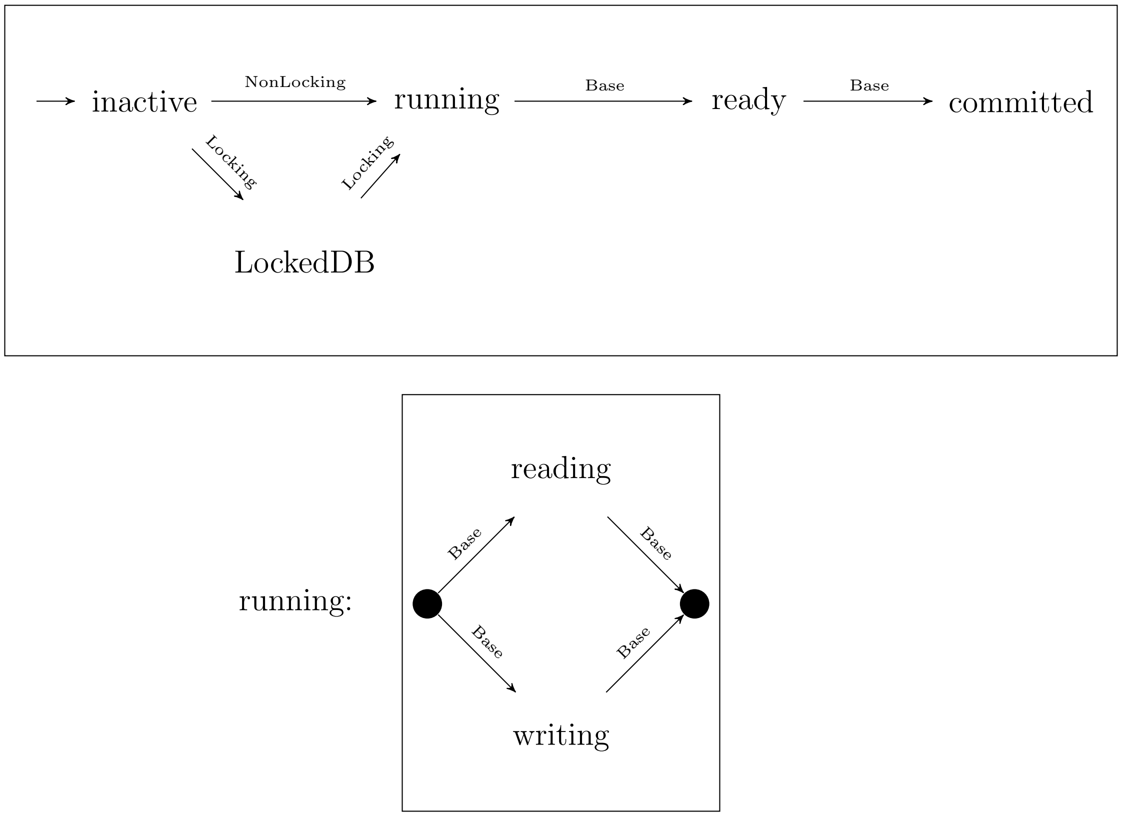

Figure 4.6: Example of Hierarchical Refinement in SDVA

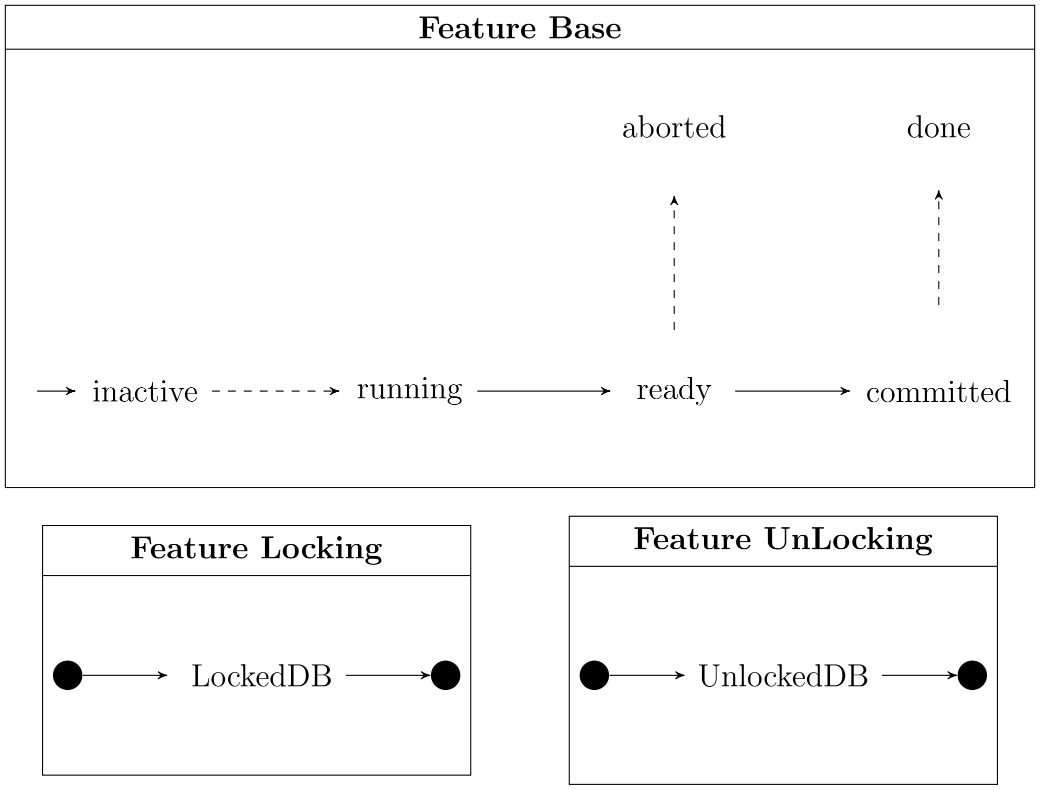

Figure 4.7: Example of Feature-Composition-Based Transition System by Fisler et al.

4.3 Composition-Based Modeling

So far, we have presented annotation-based modeling techniques. In the following, we survey composition-based modeling techniques. We present composition-based formalisms in Section 4.3.1 and composition-based modeling languages in Section 4.3.2.

4.3.1 Composition-Based Formalisms

In the previous section, we have seen that transition-based formalisms are widely used for annotation-based product-line models. For composition-based modeling, similar formalisms have been proposed. All existing composition-based transition systems are feature-based, i.e., each feature is encapsulated in a separate module.

In Figure 4.7, we exemplify a technique for composition-based modeling based on transition systems, initially proposed by Fisler et al. [63, 101–103]. In this technique, a product-line is modelled as a base feature with a number of additional feature modules. Intuitively, the base feature can be seen as a transition system with special transitions that serve as extension points. At these fixed locations, feature modules can be plugged in to extend the behavior of the model. Each feature module is modelled as a partial transition system with special input and output transitions. In the example, feature *Locking* consists of a single state. In our example, this feature can be plugged in between state *inactive* and state *running* of the base model. A limitation of this approach

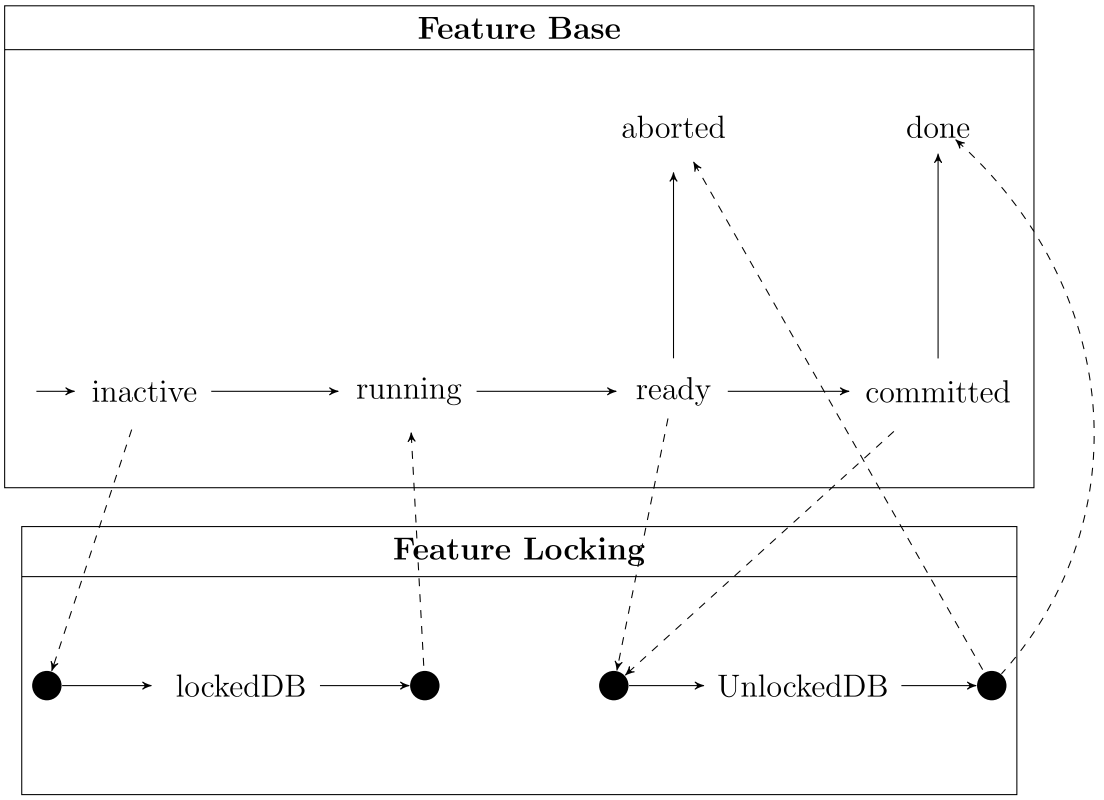

Figure 4.8: Example of Feature-Composition-Based Transition System by Liu et al.

is that feature modules cannot crosscut the base model. Thus, we have to extract the unlocking functionality of feature *Locking* into a separate feature module. Feature *UnLocking* can be plugged in between state *ready* and state *aborted*, or between state *done* and state *committed*.

In Figure 4.8, we show a similar approach in which feature modules are successively added to a base system. The transitions between states of different features are explicitely defined [105]. Thus, each feature can have multiple incoming and outgoing transitions to arbitrary states of the base system. This allows us to incorporate both state *lockedDB* and state *unlockedDB* in a single module. However, this approach does not allow to replace transitions. Thus, the transition between state *start* and state *inactive* of the base feature are included in all products. To avoid this, we would have to extract these transitions into a separate feature *NonLocking* that must be included if feature *Locking* is not selected.

Cordy et al. proposed a similar composition-based technique, in which the base system is modeled as a transition system and feature modules as special transtition systems called TS+ [44]. A TS+ is a transition system with activation conditions that express the states of the base system from which the feature module can be reached, and return conditions that define the set of states into which the feature module may return to the base system.

All of the above presented approaches, are suitable only for a closed-world assumption of product-lines in which all features are previously known [83]. The reason is that they rely on fixed locations at which feature modules can be plugged in. The models of the presented approaches are intended to be used as foundation for the derivation of products and for compositional analysis of individual features. The modularization of features is a prerequisite for such compositional analysis and cannot be performed based on annotation-based models.

In research regarding evolution of product lines, Finite State Machines with Variablity (FSMv) have been proposed [114]. FSMv considers for each feature a model of its requirements (FSMr) and a model of its design (FSMd). The purpose of this composition-based approach is to support an open-world assumption of product-lines in which previously unknown features may be added to a product-line.

A further related line of research in which feature-composition-based models have been proposed is aspect-oriented software development [5, 85, 119, 142]. These approaches also rely on a transition system for the base system and further transition systems to model features. The difference to the previously presented approaches is that the considered variability is limited. That means that the aim is to modularize features, but different combinations of features are not considered. Instead, it is typically assumed that all features are included in the product. Despite the fact that these approaches do not focus on variability, it is still possible that different combinations of features are woven into a system to derive different products.

4.3.2 Composition-Based Modeling Languages

In addition to the previously presented composition-based modeling formalisms, a number of composition-based modeling languages have been proposed to model the behavior of product lines. In the existing composition-based modeling languages, features are encapsulated into separate modules, i.e., they are feature-composition-based. To our knowledge, family-composition-based modeling languages have not been proposed so far. However, it seems likely that such techniques could be developed by applying concepts known from family-composition-based implementation techniques such as delta-oriented programming. In the following, we present composition-based modeling technique based on modeling languages.

Using Conventional Decomposition Mechanisms A simple technique to feature-composition-based modeling that has been proposed in the literature is to use a conventional modeling language known from single systems engineering and use the existing modularization mechanisms to encapsulate features.

This technique has been investigated in the case of abstract state machines (ASM) [25]. The existing decomposition mechanisms of ASM have been explored regarding their capability to encapsulate features [21, 70]. These development methods apply the idea of stepwise refinement to formal models: an abstract model is successively refined to a more concrete model, and finally to executable code. The existing modularization mechanisms have been shown to be sufficient to modularize features for a model of Java 1.0 and its implementation on the JVM together with correctness proofs [21].

Similarly, a feature-oriented extension of Event-B has been proposed in the literature [70, 126, 143]. In this line of research, the conventional Event-B composition mechanisms have been explored regarding their capabilities to implement and compose features. The difference to the work on ASM is that a mapping between features and modules has been explicitely considered. Both approaches can be considered as feature-based modeling techniques, as each feature is encapsulated in a single module. However, these techniques are not sufficient to model arbitrary features. The problem is that they allow only one dimension of decomposition, which does not allow to express features that crosscut the dominant decomposition structure of a system [147].

For the detection of feature interactions, the modeling language Promela has been used to model the behavior of features [29]. However, these features can only be inserted at fixed locations in the base program.

Feature-Oriented Extensions of Modeling Languages In research on implementation techniques the problem to encapsulate crosscutting features has been solved by a number of composition-based implementation techniques. In the case of object-oriented programming, that also lacks the expressiveness to encapsulate crosscutting features, extensions with means to encapsulate crosscutting features such as aspect-oriented programming, feature-oriented programming, or delta-oriented programming have been proposed [86, 128, 134]. Similarly, the idea to extend an existing modeling language with means to modularize features has been applied to modeling techniques. A feature-oriented extension of the formal modeling language Alloy called FeatureAlloy has been proposed by Apel et al. [11]. The main purpose of FeatureAlloy is to bridge the gap between problem space and solution space by means of an abstract model of the product line. A FeatureAlloy model is capabable to encapsulate the behavior of crosscutting features. However, the refinement of models using multiple levels of abstraction as supported by ASM or Event-B has not been considered.

Similarly, a feature-oriented extension of state charts [109], and modeling languages to model aspects for aspect-oriented software development have been proposed [5, 119]. Again, in these techniques, the behavior of single features can be encapsulated into modules. These languages have in common, that features are encapsulated in modules that can be essembled to individual products by means of predefined rules.

The language fSMV is an extension of the input language for the NuSMV model checker and semantically based on FTS. Notable is that fSMV is a composition-based language with semantics based on an annotation-based formalism. Similarly, Modal Sequence Diagrams (MSD) have been proposed, in which each feature is encapsulated into a separate module and translated to annotation-based SMV model for model checking purposes [71] .

	Formalism/Language	Feature-Based	Family-Based
Formalisms			
Schaefer et al. [135]	SHVM	✓	
Classen et al. [38]	FTS	✓	
Muschevici et al. [118], Püschel et al. [129]	Feature Petri Nets		✓
Cordy et al. [41, 42, 45], Classen et al. [36], Devroey et al. [54]	FTS		✓
Cordy et al. [40]	A-FTS		✓
Cordy et al.[43]	FTA		✓
Fantechi and Gnesi [57]	GEMTS		✓
ter Beek et al. [149], Fischbein et al. [62], Asirelli et al. [16, 17, 19]	MTS		✓
Larsen et al. [93]	Modal I/O-automata		✓
Languages			
ter Beek et al. [135]	FLan		✓
Classen et al. [37]	fSMV		✓
Sabouri et al. [132]	Rebeca		✓
Devroey et al. [53]	SDVA		✓
Classen et al. [35]	fPromela		✓
Sabouri et al. [133], Gruler et al. [73]	PL-CCS		✓

Table 4.1: Classification of Annotation-Based Modeling Techniques

4.4 Summary

In various lines of research, several formalisms and languages to model the behavior of product lines have been proposed. In this chapter, we have given an overview of modeling techniques from the literature and classified them by means of the taxonomy of product-line representations introduced in Chapter 3.

Product-based and commonality-based modeling techniques have not been proposed in research. Instead, research has focused on variability-aware modeling techniques.

Annotation-based techniques are used to model the behavior of all products of a product line in a single model. Existing annotation-based modeling techniques differ in the characteristics of the annotations used to represent variability. Some techniques allow to define the mapping between model artifacts and products by explicitly referencing features or feature expressions. Other techniques allow to express notions of variability such as optionality or alternativity to represent the variability of a product line without including information about the mapping in terms of features.

Composition-based techniques are used to encapsulate the behavior of individual features into modules. Table 4.2 summarizes our classification of composition-based modeling techniques. The proposed formalisms are based on transition systems and I/O-automata. Whether the underlying mechanisms of feature-modularization are expressive enough to express a notion of feature refinement as known from implementation techniques is an open question as they suffer from limitations regarding their expressiveness, i.e., what kind of changes a feature can perform on the base system. For instance, in many approaches features can perform changes only to predefined locations in the base system. Similar limitations apply to some of the proposed modeling languages. However, some of these languages apply composition mechanisms that are closer to the mechanisms as used in implementation techniques.

	Formalism/Language	Feature-Based	Family-Based
Transition-Based Formalisms			
Lauenroth et al. [95, 96]	Variable IO-Automata	✓	
Cordy et al. [44]	Transition Systems/TS+	✓	
Fisler and Krishnamurthi [63], Krishnamurthi et al. [90], Li et al. [101–103], Liu et al. [105]	Transition Systems	✓	
Sipma [142]	Modular Transition Systems	✓	
Languages			
Nelson et al. [119]	Alloy	✓	
Batory and Börger [21]	ASM	✓	
Gondal et al. [70], Sorge et al. [143], Poppleton [126]	Event-B	✓	
Mahoney et al. [109]	State Charts	✓	
Calder and Miller [29]	Promela	✓	
Millo et al. [114]	FSMv	✓	
Altisen et al. [5]	Larissa	✓	
Apel et al. [11]	FeatureAlloy	✓	
Greenyer et al. [71]	Modal Sequence Diagrams	✓	

Table 4.2: Classification of Composition-Based Modeling Techniques

Our survey shows that the classification by means of our taxonomy of product-line representations is appropriate to characterize existing modeling techniques. Furthermore, it allows to identify possible directions for future research that we discuss in Chapter 6.

5. Specification of Product Lines

We have given an overview of modeling techniques for product lines that have been proposed in the literature in Chapter 4. In the presented approaches, models are typically developed in order to reason about their properties. Thus, specification techniques to express properties are required. We classify specification techniques by considering the representation type of the developed specification as introduced in Chapter 3.

In this chapter, we survey specification techniques that have been proposed to specify properties of product lines. In Section 5.1, we explain our methodology to classify specification techniques. We present commonality-based specification techniques in Section 5.2, variability-aware specification techniques in Section 5.3, and summarize the results in Section 5.4.

5.1 Classification of Specification Techniques

A specification is a description of properties that a system is expected to fulfill. In the following, we survey specification techniques for product lines that have been proposed in the literature. As we are interested in how variability is represented in existing specification techniques, we use the taxonomy of product-line representations to classify specification techniques. Specification techniques may involve transformation steps to derive further types of specifications. Figure 5.1 illustrates the notion of product-line specification techniques. While many transformations of specifications are possible, the initially developed specification is used to classify a given specification technique. This reflects our focus on the specification itself that is largely independent of possible applications such as model checking or deductive verification.

The characteristics of a specification technique largely depend on the formalism used to model or implement the system. For instance, for model checking, a model is given as transition system and properties are expressed in a logic that can be interpreted in terms of the modeling formalism. If appropriate, we also discuss the characteristics

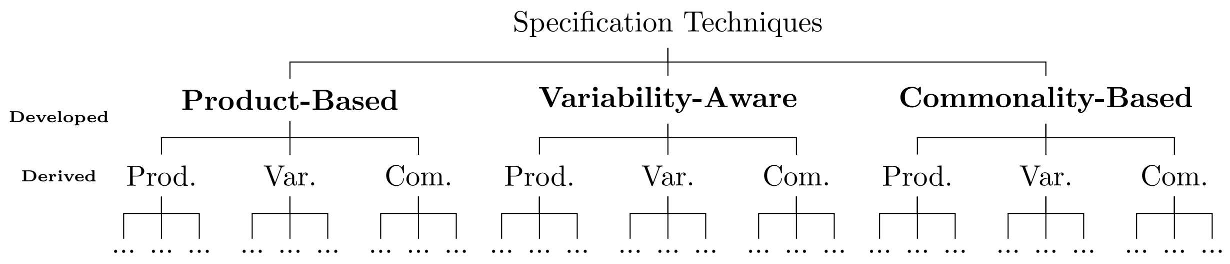

Figure 5.1: Product-Line Representations in Specification Techniques

of the underlying modeling or implementation technique. Our taxonomy allows us to consider the type of the underlying product line representation independent of whether it is a model or an implementation.

Product-based specification techniques are typically not considered in research on product lines. An exception is research on testing, in which the presence of product-specific tests might be assumed [150]. A reason for the focus of research on variability-aware techniques is that the number of products in a product line can be exponential in the number of features. Thus, product-based specification techniques cannot be expected to scale for practical applications. However, product-based specifications are often derived for analysis purposes, e.g., for product-based analysis.

5.2 Commonality-Based Specification

In contrast to implementation and modeling techniques, commonality-based representations play an important role for specification techniques. This reflects an important difference between specifications and models or implementations. A specification typically specifies only a subset of all possible properties of a system. While an implementation needs information about the differences between products to generate products, a specification may only specify the commonalities. In other words, a commonality-based representation intuitively seems to be more useful for specifications than in other contexts such as implementation.

An advantage of commonality-based specifications techniques is that conventional formalisms from single-system engineering can be used. Furthermore, only properties common to all products are specified which reduces the specification effort. This focus on commonalities is also the main disadvantage because differences between products cannot be specified.

In the context of analysis techniques, the presence of a family-wide specification is often enough to exemplify research results, e.g., that a property can be checked for all products of a product line. In the following, we give an overview of commonality-based specification techniques used in the literature.

5.2.1 Family-Wide Specification

A family-wide specification specifies properties common to all products of a product line. One area of research in which family-wide specifications have been proposed is model checking of product lines. As we have shown in Chapter 4, model checking techniques for product-lines require a variability-aware (or product-based) model in order to model the behavior of all products including their differences. When using a family-wide specification, it can be checked whether all products of a product line satisfy the specification.

An illustrative example of a family-wide specification from the literature is the pacemaker product line, in which the property that the pacemaker must send a pulse to the heart when no heartbeat is detected for a certain period [106]. This example indicates a possible motivation for using a family-wide specification. As formal specification and verification is an expensive task, we might be interested in specifying only the most crucial properties of all products. With a family-wide specification, we deliberately do not consider differences between individual products. Instead, we focus on the commonalities between all products. The underlying assumption, when using a family-wide specification, is that checking commonalities of all products provides enough benefits.

Family-wide specification does not require special techniques to handle variability. For instance, for transition-based formalisms used in model checking, conventional formalisms to specify properties such as temporal logics can be used. Accordingly, the use of LTL has been proposed to specify properties for annotation-based modeling languages and formalisms such as Rebeca [132], SHVM [135], TS+ [44], and FTS [38]. Variants of CTL have been proposed for FTA and GEMTS [43, 57]. The real-time logic tCTL has been proposed to specify properties for Featured Timed Automata [43]. The logic tCTL is an extension of CTL that supports the specification of properties related to real-time [20]. Similarly, for the specification of properties for PL-CSS, the use multi-valued modal μ-calculus has been proposed [73].

A family-wide specification can also be checked against a product-based model. In this sense, a family-wide specification can also be seen as a product-based specification that is common to all products. For this purpose, the use of LTL [63, 71, 90, 105] and CTL [44, 142] to specify properties for composition-based transition systems has been proposed.

Family-wide specification has also been used in research on testing of product lines. Variability of test-cases, i.e., the mapping between individual test cases and features is typically not considered explicitly in research on test cases [47]. In these cases, we classify the testing technique as a family-wide specification technique because for all products the same tests are typically assumed to hold [54, 84, 87–89, 123].

5.2.2 Domain-Independent Specification

Similar to family-wide specifications, domain-independent specifications have been used to specify properties for model checking. A typical example from the literature of a domain-independent property is deadlock freedom [52]. Ter Beek et al. use LTL to specify consistency between model representations of features [148].

In the context of aspect-oriented software development, specific analysis tools has been used to check generic properties of a product line. The Java Pathfinder, a model checking tool for Java, has been used to model check properties such as deadlock freedom on the level of source code [156]. Similarly, the verification tool LTSA has been used to check properties of transition systems [119]. In both cases, variability in terms of product lines has not been explicitly considered.

5.3 Variability-Aware Specification

We have shown that commonality-based specifications can be used to specify common properties of all products. Generally, it might also be desirable to consider differences between products. In principle, it is possible to develop a product-based specification for this purpose. However, the assumption in research on product lines is typically that this does not scale as the number of products is potentially exponential in the number of features. Thus, variability-aware specification techniques have been proposed. In the following, we give an overview about variability-aware specification techniques proposed in the literature. We present annotation-based specification techniques in Section 5.3.1, and composition-based specification in Section 5.3.2.

5.3.1 Annotation-Based Specification

In annotation-based specification techniques, a single specification for all products of a product line is developed. Individual parts of this specification represent properties of different sets of products.

A specification technique depends largely on the characteristics of the system to be specified. Typically, the system is either represented in terms of a model or an implementation. Accordingly, most of the modeling techniques presented in Section 4.1 have been proposed together with an accompanying specification technique to specify properties.

5.3.1.1 Feature-Annotation-Based Specification Techniques

Variable CTL has been proposed as a feature-annotation-based specification technique to specify properties of variable I/O automata [96]. Variable CTL is extends CTL with a variability relation that maps properties to features. The assumption is that each property cannot be related to more than one feature.

5.3.1.2 Family-Annotation-Based Specification Techniques

Propositional Deontic Logic (PDL) has been proposed to model permitted behavior of a system [30]. Deontic logics are capable to express notions like violation, obligation, permission, and prohibition [65]. Asirelli et al. have recognized the capability of this logic to model the behavior of product lines and proposed a temporal extension as a means to express properties of product lines modeled as MTS [18]. Their extension is able to express dynamic properties in terms of transitions and to consider the difference between may-transitions and must-transitions. This is achieved by means of the strong permission operator $P(\alpha)$, that holds if an action α is allowed to be performed in every way, and the weak permission operator $P_W(\alpha)$, that holds if an action α is allowed to be performed in some way.

To exemplify the notion of weak and strong permission, we consider an example. A property of the transaction product line might state that transactions are allowed to be rescheduled. Interpreted in the sense of strong permission, this would imply that rescheduling of transactions is also allowed for successfully committed transactions. Interpreted in the sense of weak permission, it is possible to state that rescheduling of a transaction is allowed if this transaction has been canceled.

Obligation $O(\alpha)$ is defined as $P(\alpha) \wedge \neg P_W(\neg\alpha)$, i.e. α is obligated if and only if it is strongly permitted and no other action is weakly permitted.

According to Asirelli et al. deontic logics are a a suitable means to specify both constraints and behavior of product lines [15]. Obligation is a suitable concept to express that a property must hold for all products, permission can be used to express that a property is variable. A MTS can be characterized in terms of a deontic logic formula by associating a logical formula to each state of the MTS, where must-transitions are represented by obligations and may-transitions as permissions. This opens the possibility to state properties of MTS in terms of deontic logic formula. Properties of the following kind can be expressed:

- The product line permits to derive a product that fulfills a condition, e.g., to derive a product in which rescheduling of transactions is possible.
- The product line obliges every product to fulfill a condition, e.g., that in all products transactions can be committed.

Similarly, feature models including constraints between features can be characterized in terms of deontic logic formula. This possibility has been proposed to overcome the limitations of MTS to express such constraints.

The logic DHML has been developed based on the ideas of the previously described deontic logic [15]. It is a deontic extension of Hennessy-Milner logic with Until [51, 92]. It contains the deontic operators described previously, and the CTL path operator $E(\Pi)$ that holds if there exists a path on which Π holds, and the operator $A(\Pi)$ that holds if on each of the possible paths Π holds, action operators $[a]\varphi$ denoting that for all next states reachable with action a, φ holds, the Until operator $\varphi U \varphi'$ denoting that in the current or a future state φ' holds, while φ holds until that state. An example of a DHML formula for a property of the transaction product line is `[abort](P(reschedule))`, that states that after the execution of action abort, action reschedule is permitted.

A further deontic logic, vaCTL (initially called MHML), a variability-based and action-based branching time logic interpreted over MTS has been proposed [16, 17, 19]. It can be seen as an action-based variant of DHML. It contains an action-based variant of the Until operator and directly allows to use deontic variants of operators. Action-based means that it is possible to refer to transitions in the specification of properties. The deontic interpretation of an operator, denoted by the additional symbol $\Box$, is used to express a property that must hold for all products, i.e., it considers paths with only must-transitions. Similarly, the use of an operator without its deontic interpretation is used to express properties that hold for some but not all products, i.e., it considers paths independent of the transition type.

Using the deontic interpretation of the Until operator, the property that a transaction cannot start after it has been aborted can be expressed in vaCTL as

$$\neg\texttt{E[true \{aborted\} U}^{\Box}\texttt{ \{start\} true]}$$

As the Until operator is used in its deontic interpretation, this property must hold for all products. In this case, the database product line does not fulfill the property. If the rescheduling feature is selected, a transaction can be rescheduled and activated again after it has been aborted. However, the property

$$\neg\texttt{E[true \{aborted\} U \{start\} true]}$$

holds as it is not restricted to must-transitions.

The proposed use of deontic logics to specify product lines is twofold. On the one hand, these logics can be used to express constraints between features in terms of transitions such as alternatives and exclusion to overcome the limitation of MTS to express such contraints. On the other hand, DHML and vaCTL can be used to express behavioral properties of the product line. A property can be required to hold for all products of a product line, or only by some. This can be expressed by using the deontic operators. We classify deontic logics to specify properties for product line as family-annotation-based specifications because different parts of a specification are mapped to either all products or to a subset of the products.

For Featured Transition Systems, feature-oriented variants of LTL and CTL, fLTL [35, 36, 42, 45] and fCTL [37], have been proposed. They support the use of feature expressions as guards to specify for which products a property holds. An example for a fCTL formula is

$$\texttt{[}\neg\texttt{Rescheduling] AG }\neg\texttt{ [rescheduled]}$$

that states that if feature Rescheduling is not selected, transactions are never rescheduled.

For the adaptive variant of FTS, A-FTS, a corresponding variant of CTL called Adaptive Configuration Time Logic (AdaCTL) has been proposed [40]. In adaCTL, feature formula are used to express conditions over both the system configuration and the environment configuration of the underlying A-FTS. A further difference is that these feature expressions can be nested within subformulas. This allows properties to consider changes to the configuration that happen over time.

In contrast, to deontic logics as proposed for MTS, feature-oriented logics add a further dimension of variability to the specification of properties. While deontic logics for MTS allow to express properties that consider the different types of transitions of the underlying model, feature-oriented logics for FTS go one step further and support mapping of properties explicitly to sets of products. This mapping is orthogonal to the mapping between the transitions of the model and the products.

5.3.2 Composition-Based Specification of Product Lines

In composition-based models, each module is mapped to a set of products. In Section 4.3, we have presented several composition-based modeling techniques. For instance, the use of service diagrams has been proposed to specify the behavior of product lines. For each feature a separate specification is developed [74]. This specification is largely independent of the representation of the system. In the other cases, a specific type of model or implementation is specified by enriching its modules with a set of properties that are expected to hold for the same set of products. Thus, the characteristics of the specification can be seen as inherited from the underlying modeling technique. Accordingly, the composition-based specification techniques have been proposed to specify properties of composition-based models.

In some of the presented techniques, the specification of the models is not explicitly considered but a set of properties for each feature module is assumed to exist. This is the case for CTL properties in research regarding compositional verification [95, 101–103] for LTL properties in research on feature interactions and aspect-orientation [29, 52] and for test cases in research on product-line testing [32, 79, 163].

Composition-Based specification of properties has been proposed for composition-based modeling techniques. The composition-based modeling language FeatureAlloy also allows to specify properties together with the model in feature modules [11]. As discussed in Chapter 4, FeatureAlloy is a feature-composition-based language, accordingly. As it also allows to specify properties, we classify it as a feature-based specification language accordingly.

Feature-Composition-Based Specification In research regarding feature-oriented programming, the specification of systems by means of method contracts and class invariants have been proposed. Figure 5.2 illustrates the mapping between specification

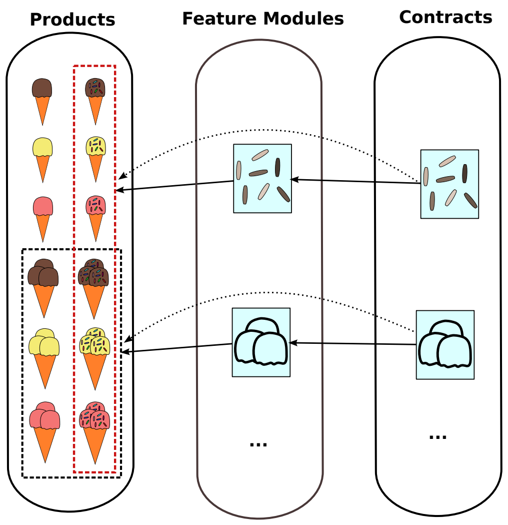

Figure 5.2: Contracts for Feature-Oriented Programming

and sets of products. Each feature module is enriched with a separate specification. As each feature module is mapped to a feature, the specification of a feature is transitively mapped to the same feature.

The Java Modeling Language has been used to specify method contracts for product lines implemented in Java [27, 152–154]. In other cases, similar contract-based languages have been proposed [12, 14, 156]. Thüm et al. propose several approaches of contract composition, that differ in the characteristics of the composition mechanism that is used to refine method contracts [154]. For instance, contracts can be overridden or merged, contract refinement can be permitted or allowed, and contract refinements may be allowed to reference the contract of the original method.

Apel et al. introduces a notion of feature-composition-based specifications that are not allowed to reference parts of the specification from other features [13]. We show that our taxonomy, can be extended to include this notion by giving the following definition:

Definition 5.1. *A feature-modular specification is a feature-composition-based specification in which the specification of a feature does not reference specification parts of other features.*

In this line of research, Apel et al. consider contract-based specifications, and investigate their expressiveness regarding the possibility to detect feature interactions. The research results imply that feature-composition-based may play an important role in the future as they provide benefits in terms of increased modularity [122].

For aspect-oriented programming the specification of a system by means of contracts has also been proposed. In this line of research, we can distinguish between conventional method contracts and advice contracts. An advice contract states preconditions and postconditions for an advice rather than for a method. Some techniques proposed in

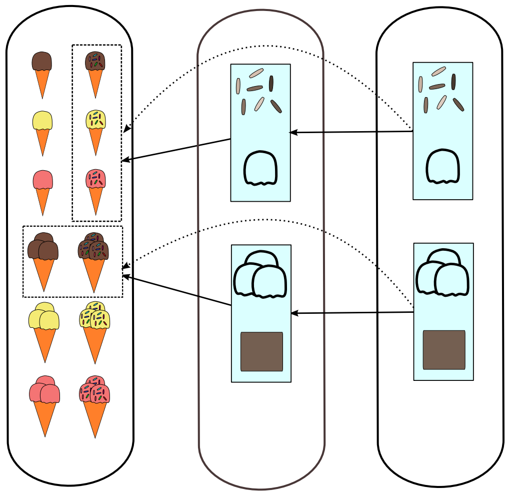

Figure 5.3: Contracts for Delta-Oriented Programming

the literature support the specification of a system in terms of method contracts [3, 107, 141]. Some additionally support advice contracts [3, 107, 164]. In both cases, we classify the specification technique as feature-composition-based because each contract is mapped to a single feature.

Family-Composition-Based Specification Similar to the use of contracts to specify properties for product lines implemented with feature-oriented programming, contracts have also been proposed for delta-oriented programming [48, 137]. Figure 5.3 illustrates the principle of the underlying mapping between specification and products. The specification inherits the mapping to products from the underlying implementation technique. Accordingly, the specification is family-composition based. It should be noted, that this is the only family-composition-based specification technique that has been proposed to our knowledge.

5.4 Summary

To specify properties of a product-line, specification techniques are necessary. We distinguish between techniques by considering the type of the developed product-line representation according to our taxonomy introduced in Chapter 3. Table 5.1 shows an overview of our classification.

Family-wide specifications that specify properties common to all products of a single product line have been proposed in the literature. This approach has been proposed mainly for model checking and testing of product lines. Similarly, domain-independent specifications have also been proposed. In the context of model checking, generic properties such as deadlock freedom can be checked independent of a specific product line. Domain-independent specifications are not used for testing, as a test case is always specific to a certain product-line. Commonality-based specifications can be used to specify

	Formalism/Language	Feature-Based	Family-Based	Composition-Based	Annotation-Based	Domain-Independent	Family-Wide
Lauenroth et al. [96]	Variable CTL	✓			✓		
Classen et al.[37]	fCTL		✓		✓		
Classen et al. [35, 36], Cordy et al. [42]	fLTL		✓		✓		
Cordy et al. [40]	AdaCTL		✓		✓		
Millo et al. [114]	VMC (CTL-Based)		✓		✓		
Asirelli et al. [16, 19]	vaCTL/MHML		✓		✓		
Harhurin and Hartmann [74]	Service Diagrams	✓		✓			
Calder and Miller [29]	LTL	✓		✓			
Apel et al. [11]	FeatureAlloy	✓		✓			
Johansen et al. [79]	Tests	✓		✓			
Zhao and Rinard [164], Lorenz and Skotiniotis [107], Griswold et al. [72], Shinotsuka et al. [141],Agostinho et al. [3]	Contracts (AOP)	✓		✓			
Scholz et al. [137], Thüm et al. [152–154], Apel et al. [12, 14]	Contracts (FOP)	✓		✓			
Damiani et al. [48], Bruns et al. [27]	Contracts (DOP)		✓	✓			
Denaro and Monga [52, 148]	LTL					✓	
Ubayashi and Tamai [156]	JPF (verification tool)					✓	
Nelson et al. [119]	LTSA (verification tool)					✓	
Greenyer et al. [71],Krishnamurthi et al. [90], Fisler and Krishnamurthi [63], Liu et al. [105]	CTL						✓
ter Beek et al. [148], Schaefer et al. [135], Classen et al. [38],Sipma [142], Cordy et al. [44], Sabouri and Khosravi [132]	LTL						✓
Cordy et al. [43]	TCTL						✓
Fantechi and Gnesi [57]	ACTL						✓
Gruler et al. [73]	multi-valued modal μ-calculus						✓
Perrouin et al. [123], Kästner et al. [84], Kim et al. [87–89], Devroey et al. [54]	Tests						✓

Table 5.1: Classification of Specification Techniques.

properties, however the role they should play in product line development processes, i.e., when and how they should be used, is unclear.

Annotation-based specification has been proposed, mainly in the form of deontic logics that are used for MTS, and feature-oriented logics that have been proposed for FTS. Deontic logics allow properties to refer to the different transition types of the underlying formalism. Feature-oriented logics as introduced for FTS, add a further dimension of variability to the specification of properties that is more independent of the underlying formalism.

Composition-based specification has been proposed, mainly in the context of contract-based specifications for composition-based implementation techniques. The mapping between specification is constituted transitively in this case, as the specification is added to the modules of the underlying implementation technique. Thus, the characteristics of these specification approaches depend on the underlying implementation technique. The question whether it is possible to use a feature-composition-based specification for properties of a family-composition-based model or vice versa has not been considered in research.

In research regarding testing of product-lines, variability-aware test-cases have not been considered extensively. Typically, the characteristics of the tests that are to be applied on a specific products are not discussed as the focus of research lies on the selection of products, e.g., by using sampling strategies.

We have shown that the classification of specification techniques according to our taxonomy of product-line representations provides an insightful view on the research area. We discuss possible directions for future research in Chapter 6.

6. Directions for Future Research

Our survey and classification of formal modeling and specification techniques for product lines in the previous chapter allows us to identify promising research directions. In the following, we present a set of possible directions for future research. For each research direction, we formulate a set of research questions.

Gap between Problem and Solution Space Our results suggests that a significant amount of research is related to modeling of product lines using formalisms that are mainly intended to serve as a foundation for verification techniques [16, 17, 19, 36, 38, 40–45, 54, 57, 62, 63, 90, 93, 95, 96, 101–103, 105, 118, 129, 135, 142, 149]. However, researchers have also recognized the need for modeling languages and proposed first concepts [5, 11, 21, 29, 35, 37, 53, 70, 71, 109, 114, 119, 126, 132, 135, 143]. A problem is that most of these approaches force the developer to choose a certain level of abstraction for development. Refinement of abstraction has been identified as an important research challenge [11, 21, 53, 70, 126, 143].

In general, there are two possible dimensions of refinement [2]. Horizontal refinement describes the addition of functionality. The second type of refinement, vertical refinement, adds more details to an abstract model. An open problem is how to cope with refinement in the presence of variability, i.e., how to deal with the relationship between the two possible dimensions of refinement [70]. Figure 6.1 illustrates vertical refinement in the context of product lines. Horizontal refinement in a product-line context means to add features. So far, research have merely identified the challenges of vertical refinement of product-lines but not solved it. Open questions are whether and how feature models should be refined, e.g., by introducing additional features during vertical refinement, and how product-line representations should be refined. For instance, it could be possible to include transformations from one representation type to other types during vertical refinement, e.g., by transforming a commonality-based model into

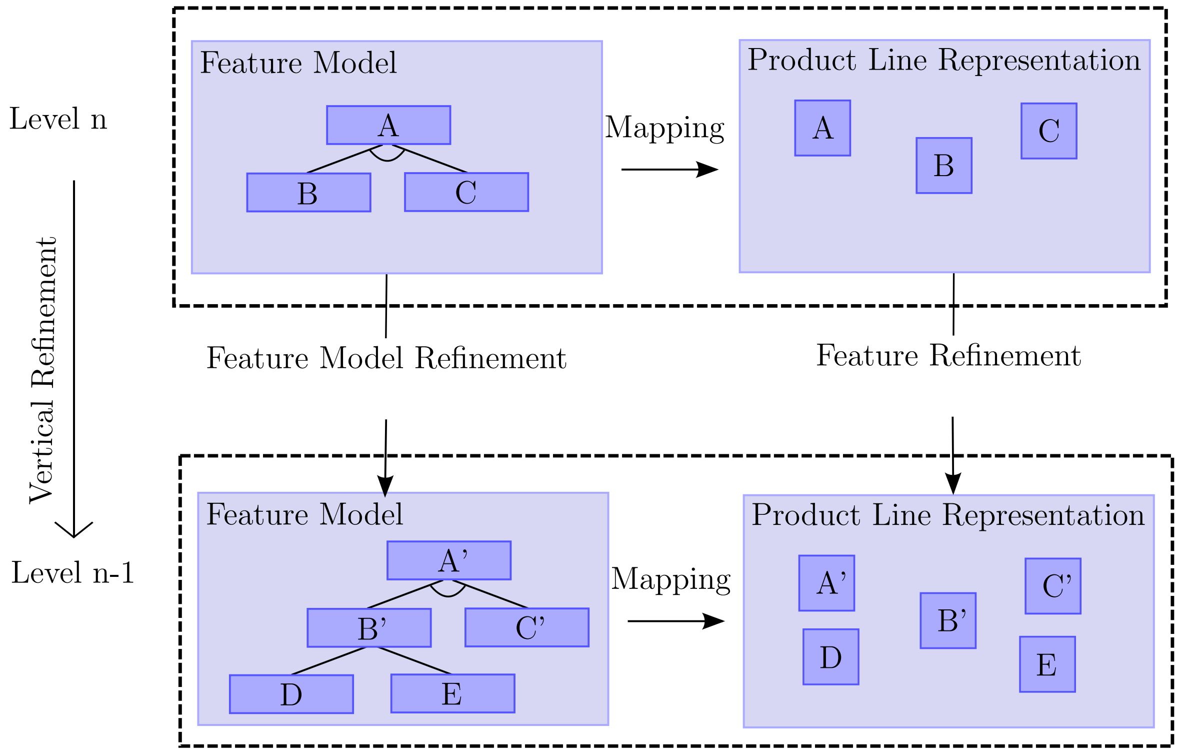

Figure 6.1: Vertical Refinement of Product-Line Representations and Feature Models

a variability-aware model. Another important challenge is to develop concepts to reuse verification effort between different refinement levels.

Furthermore, feature-oriented notions of horizontal refinement need to be developed for modeling languages. The research on Event-B and ASM has explored the capabilities of existing refinement mechanisms to encapsulate features [21, 70]. In single-systems development, typically a model is first refined horizontally until it contains the required functionality on an abstract level. This abstract model is then refined vertically too add further details. Whether this process is appropriate in a product-line setting is at least questionable, and how it could be adapted is not known.

Research Questions:

- RQ 1.1: How can refinement-based development methods be used (and possibly adapted) to develop software product lines?
- RQ 1.2: How can we deal with variability-binding between different levels of abstraction?
- RQ 1.3: How can features be traced through multiple levels of abstraction?
- RQ 1.4: How can we save verification effort with property-preserving concepts of refinement for feature modules?

Empirical Evaluation of Modeling and Specification Techniques Several modeling and specification techniques have been proposed in the literature. Existing evaluations typically focus on analysis techniques rather then modeling or specification techniques. Characteristics such as usability could be taken into account in future research. Special challenges for research are the suitability of modeling and specification techniques for evolving product-lines and for multi product lines [133, 138, 150]. In multi product lines, formal specifications may use as behavioral interfaces between related product lines [138].

Research Questions:

- RQ 2.1: What are the advantages and disadvantages of different product-line representation types regarding usability?
- RQ 2.2: Can results regarding reusability from research on variability-aware implementation techniques be applied to modeling and specification?
- RQ 2.3: What kind of specifications can be expected to be developed by software developers to formally specify product lines?
- RQ 2.4: What kind of behavior do developers expect from a contract-language for feature-oriented software development?
- RQ 2.5: How do modeling and specification techniques cope with evolving product lines?
- RQ 2.6: How do modeling and specification techniques cope with multi product lines?

Further Product-Line Representations The classification of modeling and specification techniques based on the common taxonomy of product-line representations helps to identify research gaps. Our results show that commonality-based representations are used for specification of properties but not for modeling and implementation. The main reason is that commonality-based representations do not contain enough information to derive products automatically. For implementation techniques this is a necessary requirement, but not necessarily the case for modeling techniques. For instance, commonality-based models might be used in early phases of development. The role of product-based and commonality-based models within the product-line development should be investigated in future research.

Research Questions:

- RQ 3.1: At which phases of product-line development and under which circumstances the use of commonality-based models should be recommended?
- RQ 3.2: How should different commonality-based models for the same product-line be combined?

- RQ 3.3: How can commonality-based models be refined and transformed to variability-aware models during development?

Feature-Modular Representations Existing feature-composition-based specification techniques are extensions of feature-composition-based implementation techniques [3, 11, 12, 14, 29, 72, 74, 79, 107, 137, 141, 152–154, 164]. In feature-composition-based implementation techniques, feature modules typically do not refer to parts of other feature modules directly to increase the degree of modularization. Apel et al. emphasize a notion of feature-modular specification in which the specification of each feature module does not refer to artifacts belonging to other modules [13]. Research suggests that feature-modular specifications can be used to express important but not all properties of product lines [13]. The investigation of the possible role that feature-modular specifications should play in the development of product lines, in particular for other kind of representations, is a possible direction for future research. We believe that it might also be possible that feature-modularity can be investigated independently of specific representation types.

Research Questions:

- RQ 4.1: Can the notion of feature-modularization introduced for specifications be extended for other representation types?
- RQ 4.2: When should feature-modular representations be used?
- RQ 4.3: Should languages and tools enforce feature-modular representations?
- RQ 4.4: Can representations be divided into feature-modular and non-modular parts to take advantage of both concepts?

Specification Inference for Product Lines Design by Contract has emerged as a promising specification technique to support product-line development [3, 12, 14, 27, 48, 72, 107, 137, 141, 152–154, 164]. However, research indicates that developers may benefit from automated techniques to infer or improve specifications. For this purpose, static and dynamic techniques have been proposed for single-system engineering [46, 56, 66, 131]. These approaches cannot be expected to be applicable to software product lines, as they operate on single products. Thus, they do not scale for large numbers of products. Furthermore, concepts to map results to features are required. It is not clear, how such approaches can be adapted for software product lines. For instance, dynamic invariant inference is a technique to analyze execution traces to infer likely invariants [56]. A research challenge is to adapt such techniques for the requirements of product lines e.g., by identifying commonalities between features or to use techniques such as variability encoding using an annotation-based representation of the product line to reason about multiple products simultaneously. Furthermore, techniques that are used to infer specifications might be useful for refactoring activities such as the transformation of a product-based representation to a variability-aware representation in which they may help with identification of features.

Research Questions:

- RQ 5.1: Can we adapt existing techniques for specification inference to software product lines?
- RQ 5.2: Is it possible to infer specification candidates for multiple products by analyzing a single product?
- RQ 5.3: Is it possible to analyze an annotation-based representation of the product line to infer specification candidates?
- RQ 5.4: Is it possible to apply specification inference techniques for refactoring of product lines?

Mutation Testing and Verification for Product Lines Mutation Testing is a technique that can be used to evaluate the effectiveness of tests to find possible faults in a given code base [78]. For this purpose, faults are automatically injected into the code to check whether the existing tests are able to identify the faults. Given a verification technique, the same approach can be used to evaluate the quality of a specification. In single system engineering, mutations are inserted into the source code or byte code of the product. A challenge of research is to adapt such techniques for variability-aware implementation techniques. For instance, it might be possible to insert the code into feature modules, derive an annotation-based representation to perform tests or analyses, and trace the results back to the faulty feature modules. This technique, would further allow researchers to evaluate and compare product-line analysis techniques, e.g., to compare different sampling strategies for product-based testing. Practitioners could use such techniques to increase the quality of specifications and tests.

Research Questions:

- RQ 6.1: How can we adapt mutation testing and verification techniques for software product lines?
- RQ 6.2: How can mutation testing and verification techniques be used to evaluate product-line analysis techniques?
- RQ 6.3: How can mutation testing and verification techniques be used to improve the quality of tests and specifications?

Variability Encoding Our taxonomy allows us to see commonalities of different types of representations such as implementation and models. It appears natural that some concepts that have been developed for a certain kind of representation may also be applied to other kinds. For instance, variability encoding or configuration lifting has been proposed as a technique to transform compile-time variability into run-time variability [12, 127, 153]. This is done by transforming variability-aware source code into an annotation-based representation on the level of running code. A challenge of research is to develop techniques to transfer this technique on other kind of representations such as formal models. Ideally, a uniform principle of variability encoding that does not only apply to implementations but to product-line representations in general could to be developed. At least, we consider it desirable to derive a set of requirements that a representation needs to fulfill in order to apply techniques to derive an annotation-based representation, e.g., for analysis purposes.

Research Questions:

- RQ 7.1: How can variability encoding be applied to other kinds of product-line representations such as models and specifications?
- RQ 7.2: Is it possible to identify uniform principles of variability-encoding that are independent of specific product-line representations?
- RQ 7.3: What requirements does a representation need to fulfill in order to support variability encoding?

Transfer of Research Results from Aspect-Oriented Software Development Our survey has included some of the research related to aspect-oriented software development. On the one hand, a large body of research exists for the application of design by contract to aspect-oriented programming [3, 72, 107, 141, 164]. On the other hand, composition-based models together with weaving mechanisms have been proposed [5, 85, 119, 142]. The main limitation of this research, from a product-line perspective, is that the consideration of variability is restricted, i.e., it is typically assumed that all aspects are woven together into the base system resulting in only a single product (and the base product). Aspect-oriented programming has been identified as a possible implementation technique for software product lines [6, 9, 10, 60, 82, 112]. In fact, aspect-oriented programming can be seen as a generalization of feature-oriented programming for which the use of contracts has been proposed in research related to product lines. It should be investigated to what degree techniques and research results from research on aspect-oriented software development can be transfered to a product-line contest. Ideally, unifying concepts that incorporate aspect-oriented programming as well as concepts for software product lines such as feature-oriented software development should be developed.

Research Questions:

- RQ 8.1: Can we transfer techniques and research results from aspect-oriented software development to software product lines?
- RQ 8.2: Can we develop unifying concepts of the underlying principles of both aspect-oriented software development and feature-oriented development of software product lines?

Contracts for Annotation-Based Implementation Techniques Based on our classification of specification techniques, we have identified a dichotomy pointing to a promising research direction. On the one hand, a large body of research in implementation techniques lies on annotation-based techniques. On the other hand, in research on variability-aware specification techniques the use of design by contract has gained considerable attention [3, 12, 14, 27, 48, 72, 107, 137, 141, 152–154, 164]. However, the application of design by contract has been limited to feature-annotation-based implementation techniques (FOP and AOP) and family-annotation-based implementation techniques (DOP). The application of design by contract for annotation-based implementation techniques has not been explored, so far.

A reason for the focus of research on design by contract is its focus on its application by software developers without expertise in formal methods [111]. As a light-weight formal method, it can be easily applied to existing projects. Languages such as JML are tailored to be easily adaptable by developers [98]. Similarly, annotation-based implementation techniques are a concept that many developers are familiar with, in the form of preprocessor directives used for conditional compilation. Considering the practical relevance of both approaches this seems to be a promising direction for future research.

Research Questions:

- RQ 9.1: How can design by contract be applied to annotation-based implementation techniques?
- RQ 9.2: Can research results from applying design by contract to composition-based implementation techniques be adapted for annotation-based techniques?

Variability in Test Cases Research on testing of software product lines focuses mainly on the underlying implementation representations rather than on the representation of the tests. For instance, a line of research aims to develop sampling strategies to select appropriate subsets of products for testing. However, the fact that each product may require a different set of tests is typically not considered, i.e., a family-wide representation for tests is assumed. So far, variability has been considered by means of techniques that derive product-based tests from an annotation-based test model. In a case study, the possibility of using unit-tests to specify the behavior of domain artifacts has been considered. We see the need to consider variability in test cases in future research. Especially, research on product-based or family-based testing should consider how variability in tests can be represented. The role of different variability-aware test cases in the product-line development process should be considered.

Research Questions:

- RQ 10.1: How should variability in test cases be represented?
- RQ 10.2: Which role should variability-aware test cases play in the product-line development process?

Certification and Industrial Standards In many domains, in which the use of formal methods plays an important role such as automobiles, avionics, or railway systems, the development of systems must be performed in accordance to industrial standards such as ISO 262626 for automobiles, that often require intensive quality assurance measures [49, 50]. These standards typically require that each product, even if the products are built following the product-line engineering paradigm, must be certified separately [106]. Approaches, to adapt the product-line development process to specific requirements of industrial standards in order to certify products based on variability-aware representations could provide substantial by reducing development costs. Formal models and specifications play an important role, as these domains often incorporate model-driven engineering, product-line engineering, and are safety-critical. Thus, the application of formal modeling and specification techniques for product lines are of great interest for these domains. However, the certification of systems according to industrial standards poses great challenges.

- RQ 11.1: How can model-driven development, software product line engineering, and formal verification be incorporated into a single development process?
- RQ 11.2: How can such a development process cope with requirements of industrial standards to certify products?

7. Related Work

Formal modeling and specification techniques for software product lines have been proposed in various lines of research. A number of surveys include an overview of such techniques. Similarly, there exists a number of variability taxonomies for product lines. To our knowledge, this is the first survey with a focus on variability representation of formal modeling and specification techniques. In the following, we give an overview about related work and compare it to this thesis.

Most related to this thesis is a recent survey by Thüm et al. in which the authors classify analysis techniques for software product lines regarding their strategy to cope with the variability of software product lines [151]. The classified analysis strategies include formal verification techniques that require a formal specification. Thüm et al. distinguishes between product-based, feature-based, and family-based analysis techniques as well as combinations thereof. The focus of the survey lies on analysis strategies. Thüm et al. also sketch a classification of strategies to specify properties whose terminology we have adapted and incorporated into our taxonomy. Generally, we have aimed to adopt the existing terminology as introduced by Thüm et al. as far as possible. However, we apply the terms to the representation of variability rather than on analysis strategies. The shared terminology mainly serves to see the connection between both concepts. For instance, product-based analyses require a product-based representation of the product line. The results presented in thesis can be seen as an extension of this work by shifting the focus on modeling and specification techniques in contrast to analysis techniques. Our focus is broader in the sense that we also consider techniques that have been proposed without accompanying analysis techniques. Similarly, we do not consider analysis techniques without accompanying specification technique such as type checking. Furthermore, we consider the use of test cases as a possible specification technique while testing has not been considered as an analysis technique by Thüm et al. [151].

Taxonomies of variability mechanisms for product lines have been proposed in literature [7, 75, 77, 146]. The focus typically lies on mechanisms that are used to bind variability (i.e., to derive products). Jacobson et al. present a set of different variability mechanisms: inheritance, extensions, parameterization, configuration and generation [75]. This research does not focus on product lines. Jazayeri et al. discuss variability mechanisms specific to product lines [77]. Svahnberg et al. extend this work by explicitly considering the role of features [146]. However, these taxonomies do not consider formal modeling or specifications. A reason is that variability is discussed on the level of different phases of development without going into detail of how a product line is represented at each phase. Variability of implementation techniques has been explored intensively [7]. However, in the context of implementation techniques, the focus lies on product derivation mechanisms. In contrast, we focus on the representation itself, resulting in a more general taxonomy. A further distinction between variability mechanisms in existing taxonomies is the binding time of variability, i.e., the time at which a configuration decision is realized [7, 146]. The general assumption in research on product lines is that variability can be bound in all phases of development. Accordingly, variability must be represented in all phases of development which is a motivation for our general taxonomy of product-line representations. However, we have not included a notion of binding time in our taxonomy as our goal is to reveal commonalities regarding the representation of variability in different representation types.

In a survey about software diversity, Schaefer et al. also include an concise overview on behavioral modeling of software product lines [136]. They distinguish between compositional, annotative and transformational approaches. We have decided not to make the distinction between compositional and transformational approaches as this distinction is related to mechanisms of product derivation rather than specification issues. However, we have incorporated notions of composition-based and annotation-based representations into our taxonomy to emphasize the commonalities between implementation and other kind of representations. However, our definition is more general in the sense that it only considers the representation of variability rather. Furthermore, our survey focuses on modeling and specification in more detail.

Alférez et al. compares different approaches of product-line specific requirements modeling [4]. They also use the common distinction between composition-based and annotation-based approaches, exemplifying that this notion is useful for different concepts in all phases of the development process. However, they focus on informal approaches that are out-of-scope of this thesis.

A survey on the use of formal methods for software product lines by Janota et al. provides a sophisticated discussion of the relationship between problem space and solution space from a formal methods perspective [76]. However, they focus on pure variability modeling and do not consider techniques to specify behavior or properties of a system.

Apel et al. distinguish between global, variant-based, and feature-based specifications using a slightly different terminology [13]. The notion of global specification is similar to family-wide specification as introduced in this thesis and variant-based specifications are similar to product-based specifications. In contrast, we provide definitions with a focus on the mapping between specification and products and we generalize these terms to be applicable for arbitrary representations. Furthermore, Apel et al. introduce and explore the notion of feature-modular specifications that is more strict than our notion of feature-based representations [13]. In future work, it would be possible to incorporate this notion by refining our taxonomy. We discuss possible research directions regarding feature-modular specifications in Chapter 6.

Several surveys on product line testing exist. We have incorporated test cases as a kind of formal specification of a software product lines [47, 99, 120]. However, our results indicate that product line-testing research typically does not consider fine-grained variability of test cases. A similar conclusion has been drawn in a systematic mapping study on software product line testing [47]. In this study, the question whether testing techniques involve variability testing or commonality-testing. This distinction is related to our classification of commonality-based and variability-aware representations.

Fenske et al. present a taxonomy of product-line reengineering activities [59]. Our taxonomy is related to this work, as reengineering can be seen as a transformation between two types of product-line representations, e.g., by transforming a product-based representation into a variability-aware representation. However, Fenske et al. focus on representations on the level of implementation techniques. Thus, our taxonomy generalizes the taxonomy of reengineering activities. In particular, reeengineering activities are a special case of transformations between product-line representations as introduced in our taxonomy.

8. Conclusion

Software product lines are increasingly used to develop safety-critical and mission-critical systems. To reason about the correctness of product lines, researchers have adapted verification techniques, such as model checking and theorem proving to the requirements of software product lines. Existing research has focused on the strategies to enable efficient reasoning about properties of a product line. However, to reason about properties, we need a representation of the product line such as a model or an implementation, as well as a representation of the desired properties by means of a specification.

We have presented a taxonomy of product-line representations, presented techniques for of modeling and specification of product lines, and classified the modeling and specification techniques for product lines according to our taxonomy. Furthermore, we have discussed its relationship to implementation techniques. We intend this taxonomy to be usable for many other kinds of representations such as documentation, and as a first step towards research on variability representation for software product lines that is independent of particular representation types. We expect the unifying view provided by this taxonomy to be useful for research on software product lines.

The taxonomy can be seen as a generalization from the common distinction between annotation-based and composition-based implementation techniques to other kind of representations. We focus on the representation itself, rather than on mechanisms for product generation. To exemplify the usefulness of our taxonomy, we have presented possible directions for future research that we have identified by considering the research area by means of our classification.

As far as modeling techniques are concerned, product-line research has focused on annotation-based and composition-based techniques. A large part of the proposed techniques are concerned with formalisms based on transition systems for model checking of product lines. As a result, a large part of the proposed specification techniques consist of appropriate logics that can be used to express properties for model checking. Another focus of research is on the application of design by contract to variability-aware implementation techniques.

Our classification based on a common taxonomy of representations has helped to identify research gaps such as the need for concepts to apply design by contract to annotation-based implementation techniques. We distinguish between models that are used to express behavior of a system and specifications that are used to express properties that the system is expected to fulfill. Our impressions is that variability-aware specification techniques have gained less attention as variability-aware modeling techniques. In particular, variability-aware representations of test cases should be considered in future research. Another example is the application of commonality-based representations for modeling of product lines. While commonality-based models do not require special modeling techniques, the potential role of commonality-based models within the product-line development process is unclear.

By comparing research related to specification of product lines, with research in the context of single systems engineering, it is also possible to identify possible directions of research. For instance, specification inference has not been adapted to software product lines but may lead to fruitful results. We have also identified mutation testing as a possible technique to support researchers to evaluate analysis techniques and practitioners to evaluate a given specification or set of test cases.

Our focus on modeling and specification rather than on analysis has helped to identify research related to aspect-oriented software development with results potentially transferable to product line research. Furthermore, the general notion of product-line representation suggests that research results for a specific type of representation might be transfered to other types. For instance, we have identified variability-encoding of modeling languages, i.e., the derivation of an annotation-based representation as possible research direction.

Furthermore, our survey makes clear that practical evaluation has focused on analysis rather than on specification. Evaluation of the usability of modeling and specification approaches from a developers point of view has not been considered explicitly in the literature. A large part of the techniques presented in this survey, the LTS-based modeling formalisms and corresponding specification techniques are not even aimed to be used by engineers directly but rather as a foundation for automatic analysis or for higher-level specification languages. Generally, we see large potential in further research on higher-level modeling languages, especially regarding refinement-based modeling techniques.

We have seen that the common taxonomy helps to understand commonalities and differences between implementation techniques, modeling techniques, and specification techniques resulting in a set of possible directions for future research. We hope our classification to be extended for further types of representations. In the long term, we hope to identify general variability concepts as a solid foundation for all types of representations in the software product line development process.

Bibliography

[1] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996. (cited on Page 30)

[2] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010. (cited on Page 30 and 57)

[3] Sérgio Agostinho, Ana Moreira, and Pedro Guerreiro. 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, 2008. ACM. (cited on Page 53, 54, 60, 62, and 63)

[4] Mauricio Alférez, Ana Moreira, and João Araújo. Evaluating Scenario-Based SPL Requirements Approaches — The Case for Modularity, Stability and Expressiveness. *Requirements Engineering*, pages 1–22, 10 2013. (cited on Page 66)

[5] K. Altisen, F. Maraninchi, and D. Stauch. Aspect-oriented Programming for Reactive Systems: Larissa, a Proposal in the Synchronous Framework. *Science of Computer Programming (SCP)*, 63(3):297–320, December 2006. (cited on Page 40, 41, 43, 57, and 62)

[6] Sven Apel. *The Role of Features and Aspects in Software Development*. Dissertation, University of Magdeburg, Germany, March 2007. (cited on Page 9 and 62)

[7] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, Berlin, Heidelberg, 2013. (cited on Page 5, 6, 7, 8, 9, 10, 26, 27, 30, and 66)

[8] Sven Apel and Christian Kästner. Virtual Separation of Concerns - A Second Chance for Preprocessors. *Journal of Object Technology*, 8(6):59–78, 2009. (cited on Page 8)

[9] Sven Apel, Thomas Leich, and Gunter Saake. Aspectual Mixin Layers: Aspects and Features in Concert. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 122–131, New York, NY, USA, 2006. ACM. (cited on Page 9 and 62)

[10] Sven Apel, Thomas Leich, and Gunter Saake. Aspectual Feature Modules. *IEEE Trans. Software Engineering (TSE)*, 34(2):162–180, 2008. (cited on Page 9 and 62)

[11] Sven Apel, Wolfgang Scholz, Christian Lengauer, and Christian Kästner. Detecting Dependences and Interactions in Feature-Oriented Design. In *Proc. Int'l Symposium Software Reliability Engineering (ISSRE)*, pages 161–170, Washington, DC, USA, 2010. IEEE. (cited on Page 41, 43, 51, 54, 57, and 60)

[12] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. Detection of Feature Interactions using Feature-Aware Verification. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 372–375, Washington, DC, USA, 2011. IEEE. (cited on Page 52, 54, 60, 62, and 63)

[13] Sven Apel, Alexander von Rhein, Thomas Thüm, and Christian Kästner. Feature-Interaction Detection based on Feature-Based Specifications. *Computer Networks*, 57(12):2399–2409, August 2013. (cited on Page 52, 60, 66, and 67)

[14] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. Strategies for Product-Line Verification: Case Studies and Experiments. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 482–491, Piscataway, NJ, USA, May 2013. IEEE. (cited on Page 52, 54, 60, and 63)

[15] Patrizia Asirelli, Maurice H. ter Beek, Alessandro Fantechi, and Stefania Gnesi. A Logical Framework to Deal with Variability. In *Proc. World Conf. Integrated Formal Methods (IFM)*, pages 43–58, Berlin, Heidelberg, 2010. Springer. (cited on Page 49)

[16] Patrizia Asirelli, Maurice H. ter Beek, Alessandro Fantechi, and Stefania Gnesi. A Model-Checking Tool for Families of Services. In *Proc. Int'l Conf. Formal Methods for Open Object-Based Distributed Systems and Int'l Conf. Formal Techniques for Networked and Distributed Systems (FMOODS/FORTE)*, pages 44–58, Berlin, Heidelberg, 2011. Springer. (cited on Page 42, 50, 54, and 57)

[17] Patrizia Asirelli, Maurice H. ter Beek, Alessandro Fantechi, Stefania Gnesi, and Franco Mazzanti. Design and Validation of Variability in Product Lines. In *Proc. Workshop Product Line Approaches in Software Engineering (PLEASE)*, pages 25–30, New York, NY, USA, 2011. ACM. (cited on Page)

[18] Patrizia Asirelli, Maurice H. ter Beek, Stefania Gnesi, and Alessandro Fantechi. Deontic Logics for Modeling Behavioural Variability. In David Benavides, Andreas Metzger, and Ulrich W. Eisenecker, editors, *VaMoS*, volume 29 of *ICB Research Report*, pages 71–76. Universität Duisburg-Essen, 2009. (cited on Page 49)

[19] Patrizia Asirelli, Maurice H. ter Beek, Stefania Gnesi, and Alessandro Fantechi. Formal Description of Variability in Product Families. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 130–139, Washington, DC, USA, August 2011. IEEE. (cited on Page 42, 50, 54, and 57)

[20] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. (cited on Page 10, 11, 12, 13, 14, and 47)

[21] Don Batory and Egon Börger. Modularizing Theorems for Software Product Lines: The Jbook Case Study. *J. Universal Computer Science (J.UCS)*, 14(12):2059–2082, 2008. (cited on Page 40, 41, 43, 57, and 58)

[22] Bernhard Beckert, Reiner Hähnle, and Peter Schmitt. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, Berlin, Heidelberg, New York, London, 2007. (cited on Page 11)

[23] Gerard Berry. Synchronous Design and Verification of Critical Embedded Systems Using SCADE and Esterel. In Stefan Leue and Pedro Merino, editors, *Formal Methods for Industrial Critical Systems*, volume 4916 of *Lecture Notes in Computer Science*, pages 2–2. Springer Berlin / Heidelberg, 2008. (cited on Page 11)

[24] Danilo Beuche. *Composition and Construction of Embedded Software Families*. PhD thesis, University of Magdeburg, Germany, 2003. (cited on Page 1)

[25] Egon Börger and Robert F. Stark. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, Secaucus, NJ, USA, 2003. (cited on Page 30 and 40)

[26] Jonathan P. Bowen and Victoria Stavridou. Safety-critical systems, formal methods and standards. *Software Engineering Journal*, 8:189–209, 1993. (cited on Page 11)

[27] Daniel Bruns, Vladimir Klebanov, and Ina Schaefer. Verification of Software Product Lines with Delta-Oriented Slicing. In *Proc. Int'l Conf. Formal Verification of Object-Oriented Software (FoVeOOS)*, volume 6528 of *LNCS*, pages 61–75, Berlin, Heidelberg, New York, London, 2011. Springer. (cited on Page 52, 54, 60, and 63)

[28] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An Overview of JML Tools and Applications. *Int'l J. Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005. (cited on Page 15)

[29] Muffy Calder and Alice Miller. Feature Interaction Detection by Pairwise Analysis of LTL Properties—A Case Study. *Formal Methods in System Design*, 28(3):213–261, 2006. (cited on Page 41, 43, 51, 54, 57, and 60)

[30] Pablo F. Castro and T. S. E. Maibaum. A Complete and Compact Propositional Deontic Logic. In *Proceedings of the 4th International Conference on Theoretical Aspects of Computing*, ICTAC'07, pages 109–123, Berlin, Heidelberg, 2007. Springer-Verlag. (cited on Page 48)

[31] Patrice Chalin, Joseph Kiniry, Gary T. Leavens, and Erik Poll. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In *Proc. Int'l Symposium Formal Methods for Components and Objects (FMCO)*, volume 4111 of *LNCS*, pages 342–363, Berlin, Heidelberg, New York, London, November 2005. Springer. (cited on Page 15)

[32] Harald Cichos, Sebastian Oster, Malte Lochau, and Andy Schürr. Model-based Coverage-driven Test Suite Generation for Software Product Lines. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems*, MODELS'11, pages 425–439, Berlin, Heidelberg, 2011. Springer-Verlag. (cited on Page 51)

[33] Edmund M. Clarke and Jeannette M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, December 1996. (cited on Page 1 and 11)

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

[35] Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. Model Checking Software Product Lines with SNIP. *Int'l J. Software Tools for Technology Transfer (STTT)*, 14(5):589–612, 2012. (cited on Page 35, 42, 50, 54, and 57)

[36] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-Francois Raskin. 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, August 2013. (cited on Page 33, 34, 42, 50, 54, and 57)

[37] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic Model Checking of Software Product Lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 321–330, New York, NY, USA, 2011. ACM. (cited on Page 42, 50, 54, and 57)

[38] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. 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, 2010. ACM. (cited on Page 33, 34, 42, 47, 54, and 57)

[39] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, USA, 2001. (cited on Page 1, 5, and 7)

[40] Maxime Cordy, Andreas Classen, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. Model Checking Adaptive Software with Featured Transition Systems.

In *Proc. Workshop Assurances for Self-Adaptive Systems (ASAS)*, volume 7740 of *LNCS*, pages 1–29, Berlin, Heidelberg, 2013. Springer. (cited on Page 34, 42, 51, 54, and 57)

[41] Maxime Cordy, Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. ProVeLines: A Product Line of Verifiers for Software Product Lines. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 141–146, New York, NY, USA, 2013. ACM. (cited on Page 34 and 42)

[42] Maxime Cordy, Andreas Classen, Gilles Perrouin, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. Simulation-Based Abstractions for Software Product-Line Model Checking. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 672–682, Piscataway, NJ, USA, 2012. IEEE Press. (cited on Page 34, 42, 50, and 54)

[43] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. Behavioural Modelling and Verification of Real-Time Software Product Lines. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 66–75, New York, NY, USA, 2012. ACM. (cited on Page 35, 42, 47, and 54)

[44] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. 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, 2012. ACM. (cited on Page 39, 43, 47, and 54)

[45] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. 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, May 2013. IEEE. (cited on Page 34, 42, 50, and 57)

[46] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. Precondition Inference from Intermittent Assertions and Application to Contracts on Collections. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'11, pages 150–168, Berlin, Heidelberg, 2011. Springer-Verlag. (cited on Page 60)

[47] Paulo Anselmo Da Mota Silveira Neto, Ivan Do Carmo Machado, John D. Mcgregor, Eduardo Santana De Almeida, and Silvio Romero De Lemos Meira. A Systematic Mapping Study of Software Product Lines Testing. *J. Information and Software Technology (IST)*, 53(5):407–423, May 2011. (cited on Page 47 and 67)

[48] Ferruccio Damiani, Olaf Owe, Johan Dovland, Ina Schaefer, Einar Broch Johnsen, and Ingrid Chieh Yu. A Transformational Proof System for Delta-Oriented Programming. In *Proc. Int'l Workshop Formal Methods and Analysis in Software*

Product Line Engineering (FMSPLE), pages 53–60, New York, NY, USA, 2012. ACM. (cited on Page 53, 54, 60, and 63)

[49] Raghad Dardar, Barbara Gallina, Andreas Johnsen, Kristina Lundqvist, and Mattias Nyberg. Industrial experiences of building a safety case in compliance with iso 26262. In *Proceedings of the 2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops*, ISSREW '12, pages 349–354, Washington, DC, USA, 2012. IEEE Computer Society. (cited on Page 64)

[50] Jose Luis de la Vara, Sunil Nair, Eric Verhulst, Janusz Studzizba, Piotr Pepek, Jerome Lambourg, and Mehrdad Sabetzadeh. Towards a Model-based Evolutionary Chain of Evidence for Compliance with Safety Standards. In *Proceedings of the 2012 International Conference on Computer Safety, Reliability, and Security*, SAFECOMP'12, pages 64–78, Berlin, Heidelberg, 2012. Springer-Verlag. (cited on Page 64)

[51] Rocco De Nicola and Frits Vaandrager. Three Logics for Branching Bisimulation. *J. ACM*, 42(2):458–487, March 1995. (cited on Page 49)

[52] Giovanni Denaro and Mattia Monga. An Experience on Verification of Aspect Properties. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, IWPSE '01, pages 186–189, New York, NY, USA, 2001. ACM. (cited on Page 48, 51, and 54)

[53] Xavier Devroey, Maxime Cordy, Gilles Perrouin, Eun-Young Kang, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Benoit Baudry. A Vision for Behavioural Model-Driven Validation of Software Product Lines. In *Proc. Int'l Symposium Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 208–222, Berlin, Heidelberg, 2012. Springer. (cited on Page 36, 42, and 57)

[54] Xavier Devroey, Gilles Perrouin, Maxime Cordy, Pierre-Yves Schobbens, Axel Legay, and Patrick Heymans. Towards Statistical Prioritization for Software Product Lines Testing. In *Proc. Int'l Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, pages 10:1–10:7, New York, NY, USA, January 2014. ACM. (cited on Page 35, 42, 47, 54, and 57)

[55] Mark Dowson. The Ariane 5 Software Failure. *SIGSOFT Softw. Eng. Notes*, 22(2):84–, March 1997. (cited on Page 11)

[56] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Sci. Comput. Program.*, 69(1-3):35–45, December 2007. (cited on Page 60)

[57] Alessandro Fantechi and Stefania Gnesi. Formal Modeling for Product Families Engineering. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 193–202, Washington, DC, USA, 2008. IEEE. (cited on Page 32, 42, 47, 54, and 57)

[58] Janet Feigenspan, Christian Kästner, Sven Apel, Jörg Liebig, Michael Schulze, Raimund Dachselt, Maria Papendieck, Thomas Leich, and Gunter Saake. Do Background Colors Improve Program Comprehension in the #Ifdef Hell? *Empirical Software Engineering*, 18(4):699–745, August 2013. (cited on Page 8)

[59] Wolfram Fenske, Thomas Thüm, and Gunter Saake. A Taxonomy of Software Product Line Reengineering. In *Proc. Int'l Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, pages 4:1–4:8, New York, NY, USA, January 2014. ACM. (cited on Page 67)

[60] Eduardo Figueiredo, Nelio Cacho, Claudio Sant'Anna, Mario Monteiro, Uira Kulesza, Alessandro Garcia, Sérgio Soares, Fabiano Ferrari, Safoora Khan, Fernando Castor Filho, and Francisco Dantas. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 261–270, New York, NY, USA, 2008. ACM. (cited on Page 9 and 62)

[61] Robert E. Filman and Daniel P. Friedman. Aspect-Oriented Programming Is Quantification and Obliviousness. pages 21–35. Addison-Wesley, Boston, 2005. (cited on Page 9)

[62] Dario Fischbein, Sebastian Uchitel, and Victor Braberman. 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, 2006. ACM. (cited on Page 31, 42, and 57)

[63] Kathi Fisler and Shriram Krishnamurthi. 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, 2001. ACM. (cited on Page 38, 43, 47, 54, and 57)

[64] Robert France and Bernhard Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *2007 Future of Software Engineering*, FOSE '07, pages 37–54, Washington, DC, USA, 2007. IEEE Computer Society. (cited on Page 30)

[65] Dov M. Gabbay and Franz Guenthner. *Handbook of Philosophical Logic: Volume 15*. Springer Publishing Company, Incorporated, 2nd edition, 2010. (cited on Page 48)

[66] Mark Gabel and Zhendong Su. Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 339–349, New York, NY, USA, 2008. ACM. (cited on Page 60)

[67] GCC Development Team. The C Preprocessor. Website, 2011. Available online at http://gcc.gnu.org/onlinedocs/cpp/index.html; visited on January 11th, 2011. (cited on Page 8)

[68] Jeremy Gibbons. Formal methods: Why should i care? - the development of the t800 transputer floating-point unit. In *In Proc. 13th New Zealand Computer Society Conference*, pages 207–217, 1993. (cited on Page 11)

[69] Hassan Gomaa and Mohamed Hussein. Dynamic Software Reconfiguration in Software Product Families. In Frank van der Linden, editor, *PFE*, volume 3014 of *Lecture Notes in Computer Science*, pages 435–444. Springer, 2003. (cited on Page 34)

[70] Ali Gondal, Michael Poppleton, and Michael Butler. Composing Event-B Specifications: Case-Study Experience. In *Proc. Int'l Symposium Software Composition (SC)*, pages 100–115, Berlin, Heidelberg, 2011. Springer. (cited on Page 40, 41, 43, 57, and 58)

[71] Joel Greenyer, Amir Molzam Sharifloo, Maxime Cordy, and Patrick Heymans. Efficient Consistency Checking of Scenario-Based Product-Line Specifications. In *Proc. Int'l Conf. Requirements Engineering (RE)*, pages 161–170, Piscataway, NJ, USA, September 2012. IEEE. (cited on Page 41, 43, 47, 54, and 57)

[72] William G. Griswold, Kevin Sullivan, Yuanyuan Song, Macneil Shonle, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan. Modular Software Design with Crosscutting Interfaces. *IEEE Software*, 23(1):51–60, January 2006. (cited on Page 54, 60, 62, and 63)

[73] Alexander Gruler, Martin Leucker, and Kathrin Scheidemann. 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, New York, London, 2008. Springer. (cited on Page 36, 42, 47, and 54)

[74] Alexander Harhurin and Judith Hartmann. Towards Consistent Specifications of Product Families. In *Proc. Int'l Symposium Formal Methods (FM)*, pages 390–405, Berlin, Heidelberg, 2008. Springer. (cited on Page 51, 54, and 60)

[75] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1997. (cited on Page 66)

[76] Mikoláš Janota, Joseph Kiniry, and Goetz Botterweck. Formal Methods in Software Product Lines: Concepts, Survey, and Guidelines. Technical Report Lero-TR-SPL-2008-02, Lero, University of Limerick, May 2008. (cited on Page 66)

[77] Mehdi Jazayeri, Alexander Ran, and Frank van der Linden. *Software Architecture for Product Families: Principles and Practice.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. (cited on Page 66)

[78] Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Softw. Eng.*, 37(5):649–678, September 2011. (cited on Page 61)

[79] Martin Fagereng Johansen, Øystein Haugen, Franck Fleurey, Erik Carlson, Jan Endresen, and Tormod Wien. A Technique for Agile and Automatic Interaction Testing for Product Lines. In *Proc. Int'l Conf. Testing Software and Systems (ICTSS)*, volume 7641 of *LNCS*, pages 39–54, Berlin, Heidelberg, 2012. Springer. (cited on Page 51, 54, and 60)

[80] Cliff B. Jones. *Systematic Software Development using VDM.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2nd edition, 1990. (cited on Page 30)

[81] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990. (cited on Page 1 and 6)

[82] Christian Kästner, Sven Apel, and Don Batory. A Case Study Implementing Features Using AspectJ. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 223–232, Washington, DC, USA, 2007. IEEE. (cited on Page 9 and 62)

[83] Christian Kästner, Sven Apel, and Klaus Ostermann. The Road to Feature Modularity? In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD)*, pages 5:1–5:8, New York, NY, USA, 2011. ACM. (cited on Page 40)

[84] Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. Toward Variability-Aware Testing. In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD)*, pages 1–8, New York, NY, USA, September 2012. ACM. (cited on Page 47 and 54)

[85] Shmuel Katz. Aspect Categories and Classes of Temporal Properties. *Trans. Aspect-Oriented Software Development*, 1:106–134, 2006. (cited on Page 40 and 62)

[86] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 220–242, Berlin, Heidelberg, New York, London, 1997. Springer. (cited on Page 9 and 41)

[87] Chang Hwan Peter Kim, Don Batory, and Sarfraz Khurshid. Reducing Combinatorics in Testing Product Lines. In *Proc. Int'l Conf. Aspect-Oriented Software*

Development (AOSD), pages 57—68, New York, NY, USA, 2011. ACM. (cited on Page 47 and 54)

[88] Chang Hwan Peter Kim, Sarfraz Khurshid, and Don Batory. Shared Execution for Efficiently Testing Product Lines. In *Proc. Int'l Symposium Software Reliability Engineering (ISSRE)*, pages 221–230, Washington, DC, USA, November 2012. IEEE. (cited on Page)

[89] Chang Hwan Peter Kim, Darko Marinov, Sarfraz Khurshid, and Don Batory. SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, New York, NY, USA, August 2013. ACM. To appear. (cited on Page 47 and 54)

[90] Shriram Krishnamurthi, Kathi Fisler, and Michael Greenberg. Verifying Aspect Advice Modularly. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, pages 137–146, New York, NY, USA, 2004. ACM. (cited on Page 43, 47, 54, and 57)

[91] Charles W. Krueger. Easing the Transition to Software Mass Customization. In *Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, PFE '01, pages 282–293, London, UK, UK, 2002. Springer-Verlag. (cited on Page 5)

[92] Kim G. Larsen. Proof systems for satisfiability in Hennessy-Milner Logic with recursion. *Theoretical Computer Science*, 72(2–3):265 – 288, 1990. (cited on Page 49)

[93] Kim G. Larsen, Ulrik Nyman, and Andrzej Wasowski. Modal I/O Automata for Interface and Product Line Theories. In *Proc. Europ. Conf. Programming (ESOP)*, pages 64–79, Berlin, Heidelberg, 2007. Springer. (cited on Page 32, 42, and 57)

[94] Peter Gorm Larsen, John Fitzgerald, and Tom Brookes. Applying Formal Specification in Industry. *IEEE Softw.*, 13(3):48–56, May 1996. (cited on Page 11)

[95] Kim Lauenroth, Andreas Metzger, and Klaus Pohl. Quality Assurance in the Presence of Variability. In *Intentional Perspectives on Information Systems Engineering*, pages 319–333, Berlin, Heidelberg, 2010. Springer. (cited on Page 32, 43, 51, and 57)

[96] Kim Lauenroth, Klaus Pohl, and Simon Toehning. Model Checking of Domain Artifacts in Product Line Engineering. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 269–280, Washington, DC, USA, 2009. IEEE. (cited on Page 32, 43, 48, 54, and 57)

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

[98] Gary T. Leavens and Yoonsik Cheon. Design by Contract with JML, September 2006. (cited on Page 63)

[99] Jihyun Lee, Sungwon Kang, and Danhyung Lee. A Survey on Software Product Line Testing. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 31–40, New York, NY, USA, 2012. ACM. (cited on Page 67)

[100] N. G. Leveson and C. S. Turner. An Investigation of the Therac-25 Accidents. *Computer*, 26(7):18–41, July 1993. (cited on Page 11)

[101] Harry Li, Shriram Krishnamurthi, and Kathi Fisler. Interfaces for Modular Feature Verification. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 195–204, Washington, DC, USA, 2002. IEEE. (cited on Page 38, 43, 51, and 57)

[102] Harry Li, Shriram Krishnamurthi, and Kathi Fisler. Verifying Cross-Cutting Features as Open Systems. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, pages 89–98, New York, NY, USA, November 2002. ACM. (cited on Page)

[103] Harry Li, Shriram Krishnamurthi, and Kathi Fisler. Modular Verification of Open Features Using Three-Valued Model Checking. *Automated Software Engineering*, 12(3):349–382, July 2005. (cited on Page 38, 43, 51, and 57)

[104] Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, AOSD '11, pages 191–202, New York, NY, USA, 2011. ACM. (cited on Page 8)

[105] Jing Liu, Samik Basu, and Robyn Lutz. Compositional Model Checking of Software Product Lines using Variation Point Obligations. *Automated Software Engineering*, 18(1):39–76, 2011. (cited on Page 39, 43, 47, 54, and 57)

[106] Jing Liu, Josh Dehlinger, and Robyn Lutz. Safety Analysis of Software Product Lines using State-based Modeling. *J. Systems and Software (JSS)*, 80(11):1879–1892, November 2007. (cited on Page 47 and 64)

[107] David H. Lorenz and Therapon Skotiniotis. Extending Design by Contract for Aspect-Oriented Programming. *Computing Research Repository (CoRR)*, abs/cs/0501070, 2005. (cited on Page 53, 54, 60, 62, and 63)

[108] Jochen Ludewig. Models in software engineering – an introduction. *Software and Systems Modeling*, 2(1):5–14, 2003. (cited on Page 29)

[109] Mark Mahoney, A. Bader, Tzilla Elrad, and O. Aldawud. Using Aspects to Abstract and Modularize Statecharts. 2005. (cited on Page 41, 43, and 57)

[110] John McCarthy. Towards a Mathematical Science of Computation. In *IFIP Congress*, pages 21–28, 1962. (cited on Page 11)

[111] Bertrand Meyer. Applying Design by Contract. *IEEE Computer*, 25(10):40–51, 1992. (cited on Page 14 and 63)

[112] Mira Mezini and Klaus Ostermann. Variability Management with Feature-Oriented Programming and Aspects. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, pages 127–136, New York, NY, USA, 2004. ACM. (cited on Page 9 and 62)

[113] Steven P. Miller. The Industrial Use of Formal Methods: Was Darwin Right? In *WIFT'98*, pages 74–74, 1998. (cited on Page 11)

[114] Jean-Vivien Millo, S. Ramesh, Shankara Narayanan Krishna, and Ganesh Khandu Narwane. Compositional Verification of Evolving Software Product Lines. *Computing Research Repository (CoRR)*, abs/cs/1212, 2012. (cited on Page 40, 43, 54, and 57)

[115] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997. (cited on Page 14)

[116] Munge Development Team. Munge: A Purposely-Simple Java Preprocessor. Website, 2011. Available online at http://github.com/sonatype/munge-maven-plugin; visited on January 11th, 2011. (cited on Page 8)

[117] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989. (cited on Page 35)

[118] Radu Muschevici, Dave Clarke, Jose Proenca, Goetz Botterweck, Stan Jarzabek, Tomoji Kishi, Jaejoon Lee, and Steve Livengood. Feature Petri Nets. In *Proc. Int'l Workshop Formal Methods and Analysis in Software Product Line Engineering (FMSPLE)*, pages 99–106, Lancaster, UK, September 2012. Lancaster University. (cited on Page 35, 42, and 57)

[119] Torsten Nelson, Donald D. Cowan, and Paulo S. C. Alencar. Supporting Formal Verification of Crosscutting Concerns. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, REFLECTION '01, pages 153–169, London, UK, 2001. Springer. (cited on Page 40, 41, 43, 48, 54, 57, and 62)

[120] Sebastian Oster, Andreas Wübbeke, Gregor Engels, and Andy Schürr. 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, September 2011. (cited on Page 67)

[121] Sebastian Oster, Ivan Zorcic, Florian Markert, and Malte Lochau. MoSo-PoLiTe - Tool Support for Pairwise and Model-Based Software Product Line Testing. In *Proc. Int'l Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, pages 79–82, New York, NY, USA, 2011. ACM. (cited on Page 35)

[122] David L. Parnas. On the Criteria to be used in Decomposing Systems into Modules. *Comm. ACM*, 15(12):1053–1058, December 1972. (cited on Page 52)

[123] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves le Traon. 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, April 2010. IEEE. (cited on Page 47 and 54)

[124] Jörg Pleumann, Omry Yadan, and Erik Wetterberg. Antenna: An Ant-to-End Solution For Wireless Java. Website, 2011. Available online at http://antenna.sourceforge.net/; visited on November 22nd, 2011. (cited on Page 8)

[125] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering : Foundations, Principles and Techniques.* Springer, Berlin, Heidelberg, New York, London, September 2005. (cited on Page 1, 5, and 7)

[126] Michael Poppleton. Towards Feature-Oriented Specification and Development with Event-B. In *Proc. Int'l Working Conf. Requirements Engineering: Foundation for Software Quality (REFSQ)*, volume 4542 of *LNCS*, pages 367–381, Berlin, Heidelberg, New York, London, 2007. Springer. (cited on Page 41, 43, and 57)

[127] Hendrik Post and Carsten Sinz. Configuration Lifting: Software Verification meets Software Configuration. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 347–350, Washington, DC, USA, 2008. IEEE. (cited on Page 62)

[128] Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 419–443, Berlin, Heidelberg, New York, London, 1997. Springer. (cited on Page 8 and 41)

[129] Georg Püschel, Ronny Seiger, and Thomas Schlegel. Test Modeling for Context-aware Ubiquitous Applications with Feature Petri Nets. In *Proc. Workshop Model-based Interactive Ubiquitous Systems (MODIQUITOUS)*, 2012. (cited on Page 35, 42, and 57)

[130] Raise Language Group. *Raise Method Manual.* Prentice-Hall Inc., Upper Saddle River, NJ, USA, 1993. (cited on Page 30)

[131] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Static Specification Inference Using Predicate Mining. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 123–134, New York, NY, USA, 2007. ACM. (cited on Page 60)

[132] Hamideh Sabouri and Ramtin Khosravi. Reducing the Model Checking Cost of Product Lines Using Static Analysis Techniques. In *Proc. Int'l Symposium Formal Aspects of Component Software (FACS)*, volume 7253 of *LNCS*, pages 296–312, Berlin, Heidelberg, September 2011. Springer. (cited on Page 42, 47, 54, and 57)

[133] Hamideh Sabouri and Ramtin Khosravi. Efficient Verification of Evolving Software Product Lines. In *Proc. Int'l Conf. Fundamentals of Software Engineering (FSEN)*, volume 7141 of *LNCS*, pages 351–358, Berlin, Heidelberg, 2012. Springer. (cited on Page 36, 42, and 59)

[134] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-Oriented Programming of Software Product Lines. In *Proc. Int'l Software Product Line Conference (SPLC)*, volume 6287 of *LNCS*, pages 77–91, Berlin, Heidelberg, New York, London, 2010. Springer. (cited on Page 10 and 41)

[135] Ina Schaefer, Dilian Gurov, and Siavash Soleimanifard. Compositional Algorithmic Verification of Software Product Lines. In *Proc. Int'l Symposium Formal Methods for Components and Objects (FMCO)*, volume 6957 of *LNCS*, pages 184–203, Berlin, Heidelberg, New York, London, November 2010. Springer. (cited on Page 35, 42, 47, 54, and 57)

[136] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. Software Diversity: State of the Art and Perspectives. *Int'l J. Software Tools for Technology Transfer (STTT)*, 14:477–495, 2012. (cited on Page 66)

[137] Wolfgang Scholz, Thomas Thüm, Sven Apel, and Christian Lengauer. Automatic Detection of Feature Interactions using the Java Modeling Language: An Experience Report. In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD)*, pages 7:1–7:8, New York, NY, USA, August 2011. ACM. (cited on Page 53, 54, 60, and 63)

[138] Reimar Schröter, Norbert Siegmund, and Thomas Thüm. Towards Modular Analysis of Multi Product Lines. In *Proc. Int'l Workshop Multi Product Line Engineering (MultiPLE)*, pages 96–99, New York, NY, USA, August 2013. ACM. (cited on Page 59)

[139] Johann Schumann. *Automated Theorem Proving in Software Engineering.* Springer, Berlin, Heidelberg, New York, London, 2001. (cited on Page 11)

[140] Bran Selic. The Pragmatics of Model-Driven Development. *IEEE Softw.*, 20(5):19–25, September 2003. (cited on Page 30)

[141] Suguru Shinotsuka, Naoyasu Ubayashi, Hideaki Shinomi, and Tetsuo Tamai. An Extensible Contract Verifier for AspectJ. In *Proc. Asian Workshop Aspect-Oriented Software Development (AOAsia)*, pages 1:1–1:6, Washington, DC, USA, 2006. IEEE. (cited on Page 53, 54, 60, 62, and 63)

[142] Henny Sipma. A formal model for cross-cutting modular transition systems. In *In Proceedings of the workshop on Foundations of Aspect-Oriented Languages, Northeastern University*, 2003. (cited on Page 40, 43, 47, 54, 57, and 62)

[143] Jennifer Sorge, Michael Poppleton, and Michael Butler. 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, 2010. Springer. (cited on Page 41, 43, and 57)

[144] J. M. Spivey. *Understanding Z: A Specification Language and Its Formal Semantics*. Cambridge University Press, New York, NY, USA, 1988. (cited on Page 30)

[145] Susan Stepney, David Cooper, and Jim Woodcock. An Electronic Purse: Specification, Refinement, and Proof. Technical monograph PRG-126, Oxford University Computing Laboratory Programming Research Group, July 2000. (cited on Page 11)

[146] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A Taxonomy of Variability Realization Techniques: Research Articles. *Software: Practice and Experience*, 35(8):705–754, July 2005. (cited on Page 66)

[147] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 107–119, New York, NY, USA, 1999. ACM. (cited on Page 9 and 41)

[148] Maurice H. ter Beek, Alberto Lluch Lafuente, and Marinella Petrocchi. 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, 2013. ACM. (cited on Page 36, 48, and 54)

[149] Maurice H. ter Beek, Henry Muccini, and Patrizio Pelliccione. Guaranteeing Correct Evolution of Software Product Lines: Setting up the Problem. In *Proc. Int'l Conf. Software Engineering for Resilient Systems (SERENE)*, pages 100–105, Berlin, Heidelberg, New York, London, 2011. Springer. (cited on Page 42 and 57)

[150] Maurice H. ter Beek, Henry Muccini, and Patrizio Pelliccione. Assume-Guarantee Testing of Evolving Software Product Line Architectures. In *Proc. Int'l Conf. Software Engineering for Resilient Systems (SERENE)*, pages 91–105, Berlin, Heidelberg, 2012. Springer. (cited on Page 46 and 59)

[151] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys*, 2014. To appear. (cited on Page 1, 2, 18, and 65)

[152] Thomas Thüm, Sven Apel, Andreas Zelend, Reimar Schröter, and Bernhard Möller. Subclack: Feature-Oriented Programming with Behavioral Feature Interfaces. In *Proc. Workshop MechAnisms for SPEcialization, Generalization and inHerItance (MASPEGHI)*, pages 1–8, New York, NY, USA, July 2013. ACM. (cited on Page 52, 54, 60, and 63)

[153] Thomas Thüm, Ina Schaefer, Sven Apel, and Martin Hentschel. 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, September 2012. ACM. (cited on Page 62)

[154] Thomas Thüm, Ina Schaefer, Martin Kuhlemann, Sven Apel, and Gunter Saake. Applying Design by Contract to Feature-Oriented Programming. In *Proc. Int'l Conf. Fundamental Approaches to Software Engineering (FASE)*, volume 7212 of *LNCS*, pages 255–269, Berlin, Heidelberg, New York, London, March 2012. Springer. (cited on Page 52, 54, 60, and 63)

[155] Jan Tretmans, Klaas Wijbrans, and Michel Chaudron. Software Engineering with Formal Methods: The Development of a Storm Surge Barrier Control System - Revisiting Seven Myths of Formal Methods, 2001. (cited on Page 11)

[156] Naoyasu Ubayashi and Tetsuo Tamai. Aspect-oriented programming with model checking. In *Proceedings of the 1st International Conference on Aspect-oriented Software Development*, AOSD '02, pages 148–154, New York, NY, USA, 2002. ACM. (cited on Page 48, 52, and 54)

[157] Michael VanHilst and David Notkin. 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, 1996. ACM. (cited on Page 8)

[158] Moshe Y. Vardi. Branching vs. Linear Time: Final Showdown. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2001, pages 1–22, London, UK, UK, 2001. Springer-Verlag. (cited on Page 14)

[159] David M. Weiss. The Product Line Hall of Fame. In *Proc. Int'l Software Product Line Conference (SPLC)*, page 395, Washington, DC, USA, 2008. IEEE. (cited on Page 1)

[160] W. Eric Wong, Vidroha Debroy, Adithya Surampudi, HyeonJeong Kim, and Michael F. Siok. Recent Catastrophic Accidents: Investigating How Software Was Responsible. In *Proceedings of the 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement*, SSIRI '10, pages 14–22, Washington, DC, USA, 2010. IEEE Computer Society. (cited on Page 10)

[161] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4):19:1–19:36, October 2009. (cited on Page 11)

[162] Jim Woodcock, Susan Stepney, David Cooper, John Clark, and Jeremy Jacob. The certification of the Mondex electronic purse to ITSEC Level E6. *Form. Asp. Comput.*, 20(1):5–19, December 2007. (cited on Page 11)

[163] Cemal Yilmaz. Test Case-Aware Combinatorial Interaction Testing. *IEEE Trans. Software Engineering (TSE)*, 39(5):684–706, May 2013. (cited on Page 51)

[164] Jianjun Zhao and Martin C. Rinard. Pipa: A Behavioral Interface Specification Language for AspectJ. In *Proc. Int'l Conf. Fundamental Approaches to Software Engineering (FASE)*, volume 2621 of *LNCS*, pages 150–165, Berlin, Heidelberg, New York, London, 2003. Springer. (cited on Page 53, 54, 60, 62, and 63)

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

Magdeburg, den