Analysis Strategies for Software Product Lines

THOMAS THÜM, University of Magdeburg, Germany,
SVEN APEL, University of Passau, Germany,
CHRISTIAN KÄSTNER, Philipps University Marburg, Germany,
MARTIN KUHLEMANN, University of Magdeburg, Germany,
INA SCHAEFER, University of Braunschweig, Germany,
and
GUNTER SAAKE, University of Magdeburg, Germany

Software-product-line engineering has gained considerable momentum in recent years, both in industry and in academia. A software product line is a set of software products that share a common set of features. Software product lines challenge traditional analysis techniques, such as type checking, testing, and formal verification, in their quest of ensuring correctness and reliability of software. Simply creating and analyzing all products of a product line is usually not feasible, due to the potentially exponential number of valid feature combinations. Recently, researchers began to develop analysis techniques that take the distinguishing properties of software product lines into account, for example, by checking feature-related code in isolation or by exploiting variability information during analysis. The emerging field of product-line analysis techniques is both broad and diverse such that it is difficult for researchers and practitioners to understand their similarities and differences (e.g., with regard to variability awareness or scalability), which hinders systematic research and application. We classify the corpus of existing and ongoing work in this field, we compare techniques based on our classification, and we infer a research agenda. A short-term benefit of our endeavor is that our classification can guide research in product-line analysis and, to this end, make it more systematic and efficient. A long-term goal is to empower developers to choose the right analysis technique for their needs out of a pool of techniques with different strengths and weaknesses.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.9 [**Software Engineering**]: Management—*Software configuration management*; D.2.13 [**Software Engineering**]: Reusable Software—*Domain engineering*

Additional Key Words and Phrases: Product-line analysis, software product lines, program families, deductive verification, theorem proving, model checking, type checking

1. INTRODUCTION

Software-product-line engineering has gained considerable momentum in recent years, both in industry and in academia. Companies and institutions such as NASA, Hewlett Packard, General Motors, Boeing, Nokia, and Philips apply product-line technology with great success to broaden their software portfolio, to increase return on investment, to shorten time to market, and to improve software quality (see Product-Line Hall of Fame [Weiss 2008]).

Software-product-line engineering aims at providing techniques for efficient development of software product lines [Czarnecki and Eisenecker 2000; Clements and Northrop 2001; Pohl et al. 2005]. A *software product line* (or program family) consists of a set of similar software products that rely on a common code base. The software products of a product line are distinguished in terms of the features they provide. A *feature* is a prominent or distinctive user-visible behavior, aspect, quality, or characteristic of a software system [Kang et al. 1990]. Ideally, products can be generated automatically based on a selection of features [Czarnecki and Eisenecker 2000; Batory et al. 2004; Apel and Kästner 2009].

Product-line engineering is increasingly used in mission-critical and safety-critical systems, including embedded, automotive, and avionic systems [Weiss 2008]. Hence, proper analysis methods that provide correctness and reliability guarantees are imperative for success. The underlying assumption of this survey is that every software analysis known from single-system engineering such as type checking, static analysis, and formal verification can and needs to be applied to a software product line to build reliable software products. A simple strategy for applying such analyses is to generate all software products of a product line and apply the analysis method or tool to each product individually. Unfortunately, this strategy often involves highly redundant computations and may even require repeated user assistance (e.g., for interactive theorem proving), since products of a software product line typically have similarities. Inefficiency is especially a problem for software product lines with a high degree of variability. Already a product line with 33 independent, optional features has more products than people on earth; even if the analysis runs automatically and takes only one second for each product, the sequential analysis of the whole product line would take more than 272 years.

Recently, researchers began to develop analysis techniques that take the distinguishing properties of software product lines into account. In particular, they adapted existing standard methods such as type checking and model checking such that they become aware of the variability and the features of a product line. The emerging field of product-line analysis is both broad and diverse such that it is difficult for researchers and practitioners to understand the similarities and differences of available techniques. For example, some approaches reduce the set of products to analyze, others apply a divide-and-conquer strategy to reduce analysis effort, and still others analyze the product line's code base as a whole. This breadth and diversity hinders systematic research and application.

We classify existing and ongoing work in this field, compare techniques based on our classification, and infer a research agenda in order to guide research in product-line analysis. Using our classification, it is possible to assess the analysis effort based on static characteristics of a software product line such as the number of features, the number of products, or the size of features. Our goals are (a) making research more systematic and efficient, (b) enabling tool developers to create new tools based on the research results and combine them on demand for more powerful analyses, and (c) empowering product-line developers to choose the right analysis technique for their needs out of a pool of techniques with different strengths and weaknesses.

In previous work, we proposed first ideas on a classification of *verification* approaches [Thüm et al. 2011]. Here, we extend the classification, generalize it from verification to all kinds of software analyses, and classify a corpus of existing approaches. We concentrate on analysis approaches that focus on reliability and that pursue a holistic view of a product line, incorporating design artifacts, models, and source code. Analyses that focus exclusively on requirements engineering and domain analysis (e.g., feature-model analysis) are outside the scope of this article –

we refer the reader to a recent survey [Benavides et al. 2010].

The remainder of this survey is structured as follows. In Section 2, we give a short introduction to software product lines using a running example and we present an overview on important software analysis that have been applied to software product lines. In Section 3, we define three basic strategies for the analysis of software product lines and all possible combination thereof. We discuss advantages and disadvantages of each strategy and classify existing work accordingly. In Section 4, we apply and extend our classification scheme to specification approaches for software product lines and classify existing work. In Section 5, we conclude our survey and present a research agenda for analysis strategies in software-product-line engineering.

2. BACKGROUND

We briefly introduce the necessary background for the following discussions. We present basic principles of software product lines and some software analyses that are crucial to build reliable software.

2.1 Software Product Lines

The products of a software product line differ in the features they provide, but some features are typically shared among multiple products. For example, features of a product line of database management systems are multi-user support, transaction management, and recovery; features of a product line of operating systems are multi-threading, interrupt handling, and paging.

There is a broad variety of implementation mechanisms used in product-line engineering. For example, developers of the Linux kernel combine variable build scripts with conditional compilation [Tartler et al. 2011]. In addition, a multitude of sophisticated composition and generation mechanisms have been developed [Czarnecki and Eisenecker 2000; Apel and Kästner 2009]; all establish and maintain a mapping between features and implementation artifacts (such as models, code, test cases, and documentation).

Example. We use the running example of a simple object store consisting of three features. Feature *SingleStore* implements a simple object store that can hold a single object including functions for read and write access. Feature *MultiStore* implements a more sophisticated object store that can hold multiple objects, again including corresponding functions for read and write access. Feature *AccessControl* provides an access-control mechanism that allows a client to seal and unseal the store and thus to control access to stored objects.

In Figure 1, we show the implementation of the three features of the object store using feature-oriented programming. In *feature-oriented programming*, each feature is implemented in a separate module called feature module [Prehofer 1997; Batory et al. 2004]. A *feature module* is a set of classes and class refinements implementing a certain feature. Feature module *Single* introduces a class `Store` that implements the simple object store. Analogously, feature module *Multi* introduces an alternative class `Store` that implements a more sophisticated object store. Feature module *AccessControl* refines class `Store` by a field `sealed`, which represents the accessibility status of a store, and by overriding the methods `read()` and `set()` to

Feature module *SingleStore*

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

Feature module *MultiStore*

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

Feature module *AccessControl*

```
refines class Store {
  private boolean sealed = false;
  Object read() {
    if (!sealed) { return Super.read(); }
    else { throw new RuntimeException("Access␣denied!"); }
  }
  void set(Object nvalue) {
    if (!sealed) { Super.set(nvalue); }
    else { throw new RuntimeException("Access␣denied!"); }
  }
}
```

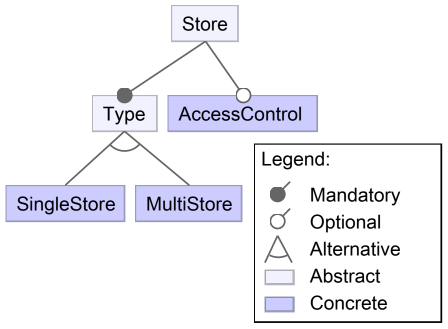
Fig. 1. A feature-oriented implementation of an object store: feature code is separated in multiple composition units.

control access (`Super` is used to refer from the overriding method to the overridden method).

Once a user has selected a list of desired features, a composer generates the final product. In our example, we use the AHEAD tool suite [Batory et al. 2004] for the composition of the feature modules that correspond to the selected features. Essentially, the composer assembles all classes and all class refinements of the features modules being composed. The semantics of a class refinement (denoted with `refines class`) is that a given class is extended by new methods and fields. Similar to a subclass, using class refinements is also possible to override or extend existing methods. While the features *SingleStore* and *MultiStore* introduce only regular Java classes, feature *AccessControl* refines an existing class by adding new members.

As said previously, there are alternative implementation approaches for software product lines (e.g., conditional compilation, frameworks) [Apel and Kästner 2009]. The analysis strategies presented in this article are largely independent of the used implementation approach.

Variability models. Decomposing the object store along its features gives rise to compositional flexibility; features can be composed in any combination. Often



(a) Feature diagram

$$\mathit{SingleStore} \Leftrightarrow \neg \mathit{MultiStore} \wedge$$
$$\mathit{AccessControl} \Rightarrow (\mathit{SingleStore} \vee \mathit{MultiStore})$$

(b) Propositional formula

$$\begin{aligned} P_1 &= \{\mathit{SingleStore}\} \\ P_2 &= \{\mathit{SingleStore}, \mathit{AccessControl}\} \\ P_3 &= \{\mathit{MultiStore}\} \\ P_4 &= \{\mathit{MultiStore}, \mathit{AccessControl}\} \end{aligned}$$

(c) Enumeration of all valid combinations

Fig. 2. The variability model of the object store in three alternative representations.

not all feature combinations are desired (e.g., we must not select *SingleStore* and *MultiStore* in the same product); hence, product-line engineers typically specify constraints on feature combinations in a variability model (i.e., the set of *valid products*). In Figure 2a, we specify the valid combinations of our object store in a feature diagram. A *feature diagram* is a graphical representation of a variability model defining a hierarchy between features, in which child features depend on their parent feature [Kang et al. 1990]. Each object store has a type that is either *SingleStore* or *MultiStore*. Furthermore, the object store may have the optional feature *AccessControl*. Valid feature combinations can alternatively be specified using *propositional formulas* [Batory 2005], as shown in Figure 2b; each variable encodes the absence or presence of a particular feature in the final product, and the overall formula yields true for valid feature combinations. In our example, there are four products that are *valid* according to the variability model, which are enumerated in Figure 2c – another representation of a feature model.

Automatic Product Generation. In this survey, we focus on implementation techniques for software product lines that support the automatic generation of products based on a selection of features. Once a user selects a valid subset of features, a *generator* generates the corresponding product, without any user assistance such as providing glue code. Examples of such implementation techniques are conditional compilation [Kästner 2010; Heidenreich et al. 2008], generative programming [Czarnecki and Eisenecker 2000], feature-oriented programming [Prehofer 1997; Batory et al. 2004], aspect-oriented programming [Kiczales et al. 1997], and delta-oriented programming [Schaefer et al. 2010]. All these approaches give software developers the ability to derive software products automatically based on a selection of desired features. The overall goal is to minimize the effort to implement new features and thus to implement new software products.

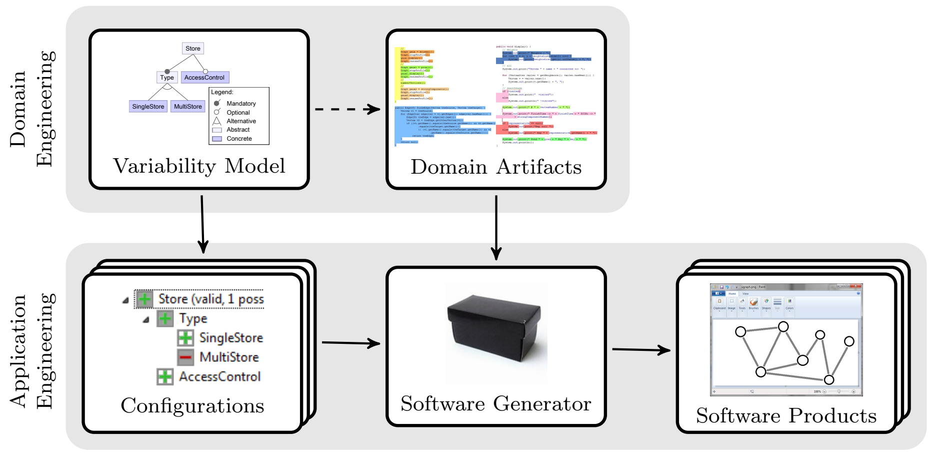


Fig. 3. In domain engineering, variability models and domain artifacts are created, which are used in application engineering to automatically generate software products based on feature selections.

In Figure 3, we illustrate the processes of domain engineering and application engineering (in a simplified form), since they are central to the development of software product lines. In domain engineering, a developer creates a variability model describing the valid combinations of features. Furthermore, a developer creates reusable software artifacts (i.e., domain artifacts) that implement each feature. For example, the feature modules of the object store are considered as domain artifacts. In application engineering, the developer determines a selection of features that is valid according to the variability model. Based on this selection, the software product containing the selected features is generated automatically based on the domain artifacts created during domain engineering. For example, composing the feature modules *SingleStore* and *AccessControl* results in generated software artifacts constituting a software product in our product line of object stores.

Correctness. An interesting issue in our running example (introduced deliberately) is that one of the four valid products misbehaves. The purpose of feature *AccessControl* is to prohibit access to sealed stores. We could specify this intended behavior formally, for example, using temporal logic:

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

The formula states, given that feature *AccessControl* is selected, whenever the object store `s` is accessed, the object store is not sealed. If we select *AccessControl* in combination with *MultiStore* (i.e., generating product P_4 from Figure 2c), the specification of *AccessControl* is violated; a client can access a store using method `readAll()` even though the store is sealed.

To fix the problem, we can alter the implementation of feature *AccessControl*. For example, we can refine method `readAll()` in analogy to `read()` and `set()`. While this change fixes the behavior problem for combining *MultiStore* and *AccessControl*, it introduces a new problem: The changed implementation of *AccessControl* no longer composes with *SingleStore*, because it attempts to override method `readAll()`, which is not present in this feature combination.

The illustrated problem is called the *optional feature problem* [Kästner et al. 2009]: The implementation of a certain feature may rely on the implementation of another feature (e.g., caused by method references) and thus the former feature cannot be selected independently, even if it is desired by the user. There are several solutions (for example, we could modify the variability model to forbid the critical feature combination for P_4, we could change the specification, or we could resolve the problem with alternative implementation patterns) [Kästner et al. 2009], but a discussion is outside the scope of this article. The point of our example is to illustrate how products can misbehave or cause compiler-errors even though they are valid according to the variability model. Even worse, such problems may occur only in specific feature combinations (e.g., only in P_4), out of potentially millions of products that are valid according to the variability model; hence, they are hard to find and may show up only late in the software life cycle. Such situation where the variability model and implementation are inconsistent, have been repeatedly observed in real product lines and are certainly not an exception [Thaker et al. 2007; Kästner et al. 2012; Tartler et al. 2011].

2.2 Software Analyses

We briefly introduce important software analyses that have been applied and adapted to software product lines (from light-weight to heavy-weight). As said previously, we focus analysis that operate statically and can guarantee the absence of errors; thus, we exclude runtime analyses and testing. Each of the discussed analyses has its strengths and weaknesses. We argue that a wide variety of analyses is needed to increase the reliability of software, in general, and software product lines, in particular.

Type Checking. A *type system* is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute [Pierce 2002]. Type systems can be used to classify programs into well-typed and ill-typed programs syntactically based on a set of interference rules. *Type checking* refers to the process of analyzing whether a program is well-typed according to a certain type system defined for the given programming language. A *type checker* is the actual tool analyzing programs written in a certain language, usually part of a compiler or linker [Pierce 2002].

Using type checking, we can detect type errors such as incompatible type casts, dangling method references, and duplicate class names. For instance, a dangling method reference occurs if a method with a certain signature is called that is not declared. For our object store, we discussed that calling method `readAll()` in feature *AccessControl* would result in a dangling method reference in product P_4. Other examples are that a programmer may have misspelled the name of a method, or the number of parameters is not correct. Type errors are frequent in the development of software; the evolution of software often requires to add new parameters to a method declaration or to rename identifiers.

A type system can be seen as a formal specification that is common to all programs written in a certain language. Pierce [2002] argues that, in principle, types can be created to check arbitrary specifications. But in practice, type systems are limited to properties that are efficiently statically decidable and checkable.

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

A model checker is a tool that performs a model-checking task based on the system to verify and its specification. Some model checkers require the use of dedicated input languages for this task (e.g., Promela in SPIN, CMU SMV input language in NuSMV), and some work on programs and specifications written in mainstream programming languages (e.g., C in Blast or CPAchecker, Java in JavaPathfinder). After encoding a model-checking problem into the model checker's input language, the model-checking task is fully automated; each property is either stated valid or a counterexample is provided. The counterexample helps the user to identify the source of invalidity. The most severe reduction for the practical applicability of model checkers is the limit of the size of the state space they can handle [Schumann 2001].

Static Analysis. The term *static analysis* refers to a set of possible program analyses that can be performed without actually executing the program [Nielson et al. 2010]. In this sense, type checking and model checking are special instances of static analysis techniques. Some static analyses approaches operate on source code (e.g., Lint for C), others on byte code (e.g., FindBugs for Java byte code). Some are lightweight such that defects are searched based on simple patterns (e.g., Lint), while others target the whole program behavior such as model checkers. Static analyses can be implemented within compilers such as Clang or in the form of dedicated tools such as FindBugs. Common examples of static analyses are control-flow analysis, data-flow analysis, and alias analysis.

Theorem Proving. Theorem proving is a deductive approach to prove the validity of logical formulas. A theorem prover is a tool processing logical formulas by applying inference rules upon them [Schumann 2001] and it assists the programmer in verifying the correctness of formulas, which can be achieved interactively or automatically. Interactive theorem provers such as Coq, PVS, or Isabelle require a user to write commands to apply inference rules. Instead, automated theorem provers such as Prover9, SPASS, or Simplify evaluate the validity without further assistance by the user.

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

Theorem provers are used in many applications, because of their high expressiveness and generality. In the analysis of software products, theorem provers are used to formally prove that a program fulfills its specification. A formal specification could be that the program terminates and returns a number larger than zero. In program verification, a specification is given in some formal language, and then a verification tool generates theorems based on implementation and specification that is the input for a theorem prover. If a theorem cannot be proved, theorem provers point to the part of a theorem that could not be proved. The main disadvantage of theorem proving is that experts with an education in logical reasoning and considerable experience are needed [Clarke et al. 1999].

3. ANALYSIS STRATEGIES FOR SOFTWARE PRODUCT LINES

Many software systems such as the Linux kernel [Berger et al. 2010; Sincero et al. 2007] are implemented as software product lines. But, contemporary analysis tools are usually inefficient, as they do not take variability into account. The reason is that software product lines require language extensions or preprocessors to express and manage variability. Hence, analysis tools are applicable mostly only to derived software products – not to domain artifacts as developed and maintained by the programmer. But, analyzing each software product of a product line individually does not scale in practice. The mismatch between efficient implementation techniques and inefficient software-analysis techniques is an open research topic. Fisler and Krishnamurthi [2005] argue that the analysis effort should be proportional to the implementation effort. Even if this goal may not be reachable in general, analyses of software product lines need to scale better than exhaustively analyzing each product.

In the last decade, researchers have proposed and developed a number of analysis techniques tailored to software product lines. The key idea is to exploit knowledge about features and the commonalities and variabilities of a product line to systematically reduce analysis effort. Existing product-line analyses are typically based on standard analysis methods, in particular, type checking, static analysis, model checking, and theorem proving. All these methods have been used successfully for analyzing single software products. They have complementary strengths and weaknesses, for instance, with regard to practicality, correctness guarantees, and complexity; so all of them appear useful for product-line analysis.

Unfortunately, in most cases it is hard to compare these analysis techniques regarding scalability or even to find the approach that fits best for a given product-line scenario. The reason is that approaches are presented using varying nomenclatures, especially if multiple software analyses are involved. In this section, we classify existing product-line-analysis approaches based on how they attempt to reduce analysis effort – the *analysis strategy*. We distinguish three basic strategies: product-based, family-based, and feature-based. We explain the basic strategies and discuss existing approaches realizing each strategy. While surveying the literature, we found that some approaches for the analysis of software product lines actually combine the basic strategies, so we discuss possible combinations.

3.1 Product-Based Analyses

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

Definition 3.1 *Product-based analysis.* An analysis of a software product line is called *product-based*, if it operates only on generated products or models thereof. A product-based analysis is called *optimized*, if it operates on a subset of all products or if intermediate analysis results are reused, and is called *unoptimized* otherwise.

Example. In our object store example, we can generate and compile each product to detect type errors. While such unoptimized product-based strategy is applicable to our small example, we need optimizations for larger software product lines. One could save analysis effort when checking whether the specification of feature *AccessControl* is satisfied: First, all products that do not contain *AccessControl* do not need to be checked. Second, if two products differ only in features that do not concern class `Store` (not shown in our example; e.g., features that are concerned with other data structures), only one of these products needs to be checked.

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

An unoptimized product-based analysis is sound and complete with respect to the applied software analysis. First, every error detected using this strategy, is an error of a software product that can be detected by the base software analysis (soundness). Second, every error that can be detected using a the considered software analysis, is also detected using an unoptimized product-based analysis (completeness). Note that while the base software analysis might be unsound or incomplete, the strategy is still sound and complete regarding to this software analysis.

However, there are serious disadvantages of product-based analyses. Already generating all products of a software product line is usually not feasible, because the number of products is up-to exponential in the number of features. Even if deriving all products is possible, the separate analyses of individual products perform inefficient, redundant computations, due to similarities between the products.

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

While an unoptimized product-based strategy is often not feasible, it serves as an benchmark for other strategies in terms of soundness, completeness, and efficiency. An ideal would be optimized strategies that are sound, complete, *and* more efficient. But, we will also discuss strategies that are incomplete (some are even unsound) to increase the efficiency of the overall analysis.

Unoptimized Product-based Analyses. Product-based strategies are widely used in practice, because they are simple and can be applied without creating and using new concepts and tools. For example, when generating and compiling individual software products, type checking is usually done internally by the compiler (e.g., the Java compiler). Type checking of the compiler is redundant when different products share implementation artifacts, and sharing artifacts between products is the common case and goal in software product lines [Czarnecki and Eisenecker 2000; Apel and Kästner 2009]. For example consider the object store, for every product containing feature *Store*, the compiler will check that the type of variable `nvalue` is a valid subtype of the type of variable `value`; but it is sufficient to check this once for all products.

We found no proposal in the literature explicitly suggesting an exhaustive product-based analyses without any optimizations. But, we found some approaches that actually propose product-based analyses and do not discuss how to deal with many products; these approaches apply type checking [Apel et al. 2008] and model checking [Kishi and Noda 2006; Apel et al. 2010; Apel et al. 2011] to software product lines. As said previously, in principle, any standard analysis applicable to the artifacts generated during application engineering can be used for product-based analysis.

Optimized Product-based Analyses. One reason for the success of software product lines is that new combinations of features can be derived automatically. For instance, the effort for the development of new products is much smaller than developing all new products from scratch. But, unoptimized product-based strategies hinder an efficient analysis of software product lines and thus an efficient development. The overall goal of software-product-line engineering is to scale product-line analyses to a similar efficiency as implementation techniques, as the development of software product lines requires both, efficient implementation and analysis strategies. Several optimized product-based strategies have been proposed to improve scalability and reduce redundant computations. Optimizations focus on detecting redundant parts in analyses and on eliminating products that are already covered by other analysis steps according to a certain coverage criteria.

Bruns et al. [2011] present a product-based approach for formal verification of delta-oriented software product lines. Delta modules are similar to feature modules, but can also remove members or classes. Bruns et al. [2011] generate all derivable software products and verify them incrementally using interactive theorem proving. First, a base product needs to be chosen and verified completely. For all other products, they choose the base product as a starting point, copy all proofs to the current product, and mark those as invalid that do not hold due to the differences to the base product. Only invalidated proofs need to be redone and some new proof obligations need to be proven.

Rubanov and Shatokhin [2011] presented runtime analyses for Linux kernel modules based on call interception. Their approach is not aware of the variability in the Linux kernel. It has been applied only to a few common configurations of the kernel and failures were detected in them. Applying the approach to all kernel configurations is infeasible as the Linux kernel has more than 10,000 features [Tartler et al. 2011] and billions of valid combinations thereof. Domain knowledge is necessary to select representative products, but errors in products not selected may go unnoticed.

Other approaches improve the efficiency of product-based strategies by eliminating products from the set of products to analyze, because some products may already be fully covered by the analyses of other products. Such an elimination is the idea behind pair-wise testing [Oster et al. 2010]. The general observation is that most errors are caused by an interaction of two features. Hence, those approaches retrieve a minimal set of products fulfilling a given coverage criterion and only those products are analyzed. The coverage criteria for pair-wise testing is that for every pair of features (F, G) products must exist in the calculated set containing (a) F but not G, (b) G but not F, and (c) both features F and G.[1] First results showed that this can substantially reduce the number of products to analyze [Oster et al. 2010], but clearly, interactions between more than two features are not covered. Thus, pair-wise testing was extended to t-wise testing to cover also interactions between t features [Perrouin et al. 2010]. But, those approaches do not scale well for a high t. If t is equal to the number of features, we need to test all products, as with unoptimized product-based testing.

Tartler et al. [2012] use a more selective strategy to sample products from the overall set of products for analysis. The idea is that the analysis procedure touches each domain artifact and individual piece of code, at least, once. This way, it attains *full code coverage*. Anyway, this and similar strategies are incomplete as well, because not all valid *combinations* of domain artifacts and variable code pieces are analyzed.

3.2 Family-Based Analyses

The main problem with product-based analyses is that the products of a software product line share code [Czarnecki and Eisenecker 2000] resulting in redundant computations. Besides an optimized product-based strategy, another option is to achieve a more efficient analysis by considering domain artifacts instead of generated artifacts (i.e., products).

Family-based analyses operate on domain artifacts and the valid combination thereof specified by a variability model. The variability model is usually converted into a logic formula to allow analysis tools to reason about all valid combinations of features (e.g., a satisfiability solver can be used to check whether a method is defined in all valid feature combinations, in which it is referenced). The overall idea is to analyze domain artifacts and variability model from which we can conclude that some intended properties hold for all products. Often, all implementation artifacts

[1]Note that only combinations of features are considered that are valid according to the variability model.

of all features are merged into a single virtual product, which is not necessarily a valid product due to optional and mutually exclusive features. We give a definition of family-based analyses as follows:

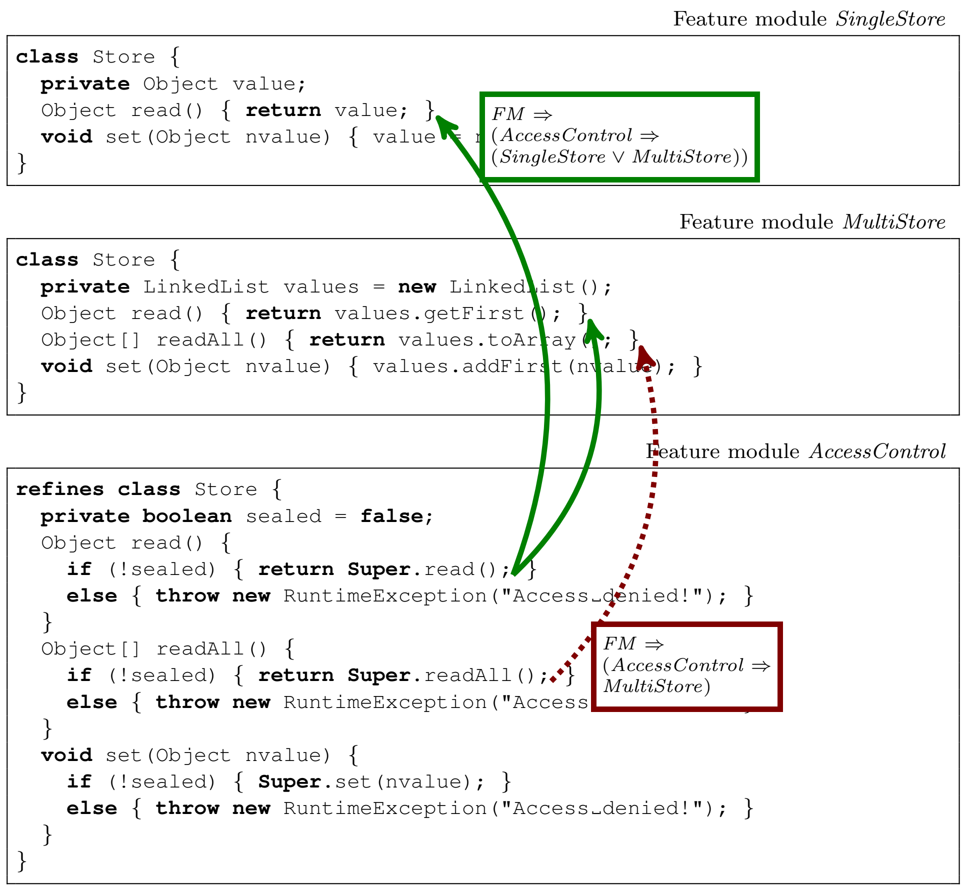
Definition 3.2 *Family-based analysis.* An analysis of a software product line is called *family-based*, if it (a) operates only on domain artifacts and (b) incorporates the knowledge about valid feature combinations.

Example. A product-line type checker, for instance, analyzes the code base of the object store example (i.e., all feature modules) in a single pass although the features are combined differently in the individual products. To this end, it takes variability into account in the sense that individual feature modules may be present or absent in certain products. Regarding method invocations, it checks whether a corresponding target method is declared in *every* valid product in which it is invoked. This may be the case because there is one feature module with a corresponding target method that is present in every valid product in which the method is called, or because there are multiple matching feature modules, of which (at least) one is present in every valid product. In Figure 4, we illustrate how a family-based type system checks whether the references of the modified feature module *AccessControl* to the methods `read()` and `readAll()` are well-typed in *every* valid product. For `read`, the type system infers that the method is introduced by the feature modules *SingleStore* and *MultiStore*, and that one of them is always present (checked using a satisfiability solver; green, solid arrows). For `readAll()`, it infers that the method is introduced only by feature module *MultiStore*, which may be absent when feature module *AccessControl* is present (red, dotted arrow). Hence, the type system reports an error and produces a counter example with a valid feature selection that contains a dangling method invocation: $\{SingleStore, AccessControl\}$. Other type checks can be made variability-aware in a similar way.

Advantages and Disadvantages. Family-based strategies have advantages compared to product-based strategies. First of all, not every single product must be generated and analyzed because family-based analyses operate on domain artifacts, thus avoid redundant computations for similarities across multiple products. Reasoning about variabilities and commonalities avoids these duplicate analyses.

Second, the analysis effort is *not* proportional to the number of valid feature combinations. While the satisfiability problem is NP-hard, in practice, satisfiability solvers perform well when reasoning about variability models [Mendonca et al. 2009; Thüm et al. 2009]. Intuitively, the performance is mainly influenced by the number of satisfiability checks (whose results can be cached to improve performance [Apel et al. 2010]) and the number of features, but largely independent of the number of valid feature combinations. For comparison, the effort for product-based approaches increases with every new product.

Third, as for product-based strategies, family-based strategies can also be applied when there are no restrictions on the valid combinations of features. We can easily apply family-based strategies with the trivial variability model containing all features and allowing all feature combinations. Such a variability model converted into a logical formula would be a tautology. Hence, family-based strategies do not require a variability model.


Feature module *SingleStore*
$FM \Rightarrow$
$(AccessControl \Rightarrow$
$(SingleStore \lor MultiStore))$
Feature module *MultiStore*
Feature module *AccessControl*
$FM \Rightarrow$
$(AccessControl \Rightarrow$
$MultiStore)$

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

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

```
refines class Store {
  private boolean sealed = false;
  Object read() {
    if (!sealed) { return Super.read(); }
    else { throw new RuntimeException("Access␣denied!"); }
  }
  Object[] readAll() {
    if (!sealed) { return Super.readAll(); }
    else { throw new RuntimeException("Access
  }
  void set(Object nvalue) {
    if (!sealed) { Super.set(nvalue); }
    else { throw new RuntimeException("Access␣denied!"); }
  }
}
```

Fig. 4. Checking whether references to `read()` and `readAll()` are well-typed in *all* valid products. *FM* denotes the variability model (as propositional formula) of Figure 2; a SAT solver determines whether the formulas in the boxes are tautologies (the upper formula is, but the lower is not).

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

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

Third, changing the variability model usually requires to analyze the whole product line again. For instance, if we add one new product or a very small set of new products, we may be faster analyzing these new products using a product-based strategy than analyzing the whole product line using a family-based strategy. But again, similar to domain artifact changes this may depend on the analysis approach and available caching strategies. There is no need to re-do any analysis, if the variability model was specialized or refactored (i.e., no new products are added) [Thüm et al. 2009].

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

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

Family-Based Type Checking. Family-based strategies were proposed by several authors for type checking of software product lines [Aversano et al. 2002; Czarnecki and Pietroszek 2006; Thaker et al. 2007; Post and Sinz 2008; Kuhlemann et al. 2009; Heidenreich 2009; Apel et al. 2010; Kästner et al. 2012]. The majority of work on family-based type checking is about creating product-line-aware type systems and proving that, whenever a product line is type safe according to the type system, all derivable products are also type safe. The rules of these type system contain reachability checks (basically implications) making sure that every class or class member is defined in all products where it is referenced. Product-line-aware type systems were presented for feature-oriented programming [Thaker et al. 2007; Kuhlemann et al. 2009; Delaware et al. 2009; Apel et al. 2010] and conditional compilation in models [Czarnecki and Pietroszek 2006; Heidenreich 2009] and source code [Kästner et al. 2012]. For product lines implemented using composition such as feature-oriented programming, type checking ensures *safe composition* [Thaker et al. 2007]. Post and Sinz [2008] applied family-based type checking to parts of the Linux kernel and were able to find one dangling method reference.

There are two approaches of family-based type checking [Apel et al. 2010]. *Local approaches* perform distinct reachability checks for every program element, for example, [Apel et al. 2010; Kästner et al. 2012]. This results in many small satisfiability problems to solve, which, however, can be cached efficiently [Apel et al. 2010]. *Global approaches* generate, based on all inferred dependencies between program elements, a single large propositional formula that is checked for satisfiability at the end of type checking [Thaker et al. 2007; Delaware et al. 2009]. This results in one large satisfiability problem to solve. Apel et al. [2010] discuss strengths and weaknesses of local and global approaches.

Family-Based Model Checking. Several family-based analyses were proposed for model checking [Post and Sinz 2008; Gruler et al. 2008; Lauenroth et al. 2009; Classen et al. 2010; Schaefer et al. 2010; Classen et al. 2011; Apel et al. 2011].

Post and Sinz [2008] propose *configuration lifting* to scale off-the-shelf verification techniques to software product lines. The idea of configuration lifting is to convert compile time variability (e.g., preprocessor directives) into runtime variability (e.g., conditional statements in C). They manually converted the implementation of a Linux device driver and analyzed it using the bounded model checker CBMC. Similarly, Apel et al. [2011] convert feature modules into monolithic code with runtime variability (via variability encoding) to be able to use an off-the-shelf model checker for family-based model checking.

Classen et al. [2011] propose *featured transition systems* to model software product lines and use specifications defined in an extension of computation tree logic. They extended the symbolic model checker NuSMV for a family-based verification of featured transition systems. The result of their empirical evaluation is that family-based model checking is faster than unoptimized product-based model checking for most properties, but sometimes even slower. In preceding work, they used specifications defined in linear time logic and implemented model checking in Haskell from scratch [Classen et al. 2010].

Lauenroth et al. [2009] propose family-based model checking based on I/O automata and CTL properties. They define I/O automata as domain artifacts that contain variable parts and can be used to derive I/O automata as the products. Their approach allows to verify the domain artifacts while making sure that every derivable I/O automata fulfills its CTL properties.

The family-based model checking by Gruler et al. [2008] is similar to the approaches of Classen et al. [2011] and Lauenroth et al. [2009]. The difference is that Gruler's approach is based on the process calculus CSS. The approach extends CCS with a variant operator to model families of processes. This variability information is exploited during model checking to verify all variants of processes simultaneously.

Schaefer et al. [2010] present a family-based approach for checking safety properties of control flow for product lines. They use simple hierarchical variability models representing all products in a single model to decompose the set of all method implementations into the set of methods that are common to all products and a set of variant points with associated variants. The variants consist of sets of methods that are again specified by simple hierarchical variability models giving rise to a hierarchical structure. The developed compositional verification principle allows splitting the verification of a global property of all products into the verification of the common methods and the verification of variation point properties.

3.3 Feature-Based Analyses

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

Definition 3.3 *Feature-based analysis.* An analysis of a software product line is called *feature-based*, if it (a) operates only on domain artifacts and (b) software artifacts belonging to a feature are analyzed in isolation (i.e., knowledge about valid feature combinations is not used).

Example. In the object-store example, we can parse and type check each of the three feature modules to *some degree* in isolation. First, we can parse each feature module in isolation to make sure that it conforms to the syntax and to get an abstract syntax tree of each feature module. For syntax checking, it is sufficient to consider each feature module in isolation as syntactic correctness is independent of other features and, thus, a compositional property. Second, the type checker uses the abstract syntax tree to infer which types and references can be resolved by a feature itself and which have to be provided by other features. As an example, all references to field `sealed` are internal and can be checked within the implementation of feature *AccessControl*, as illustrated in Figure 5. That is, there is no need to check this reference for every product. But, some of the references cut across feature boundaries and cannot be checked in a feature-based fashion. For example, references to the methods `read()` and `readAll()` of feature *AccessControl* cannot be resolved within the feature. Type correctness is usually a non-compositional property.

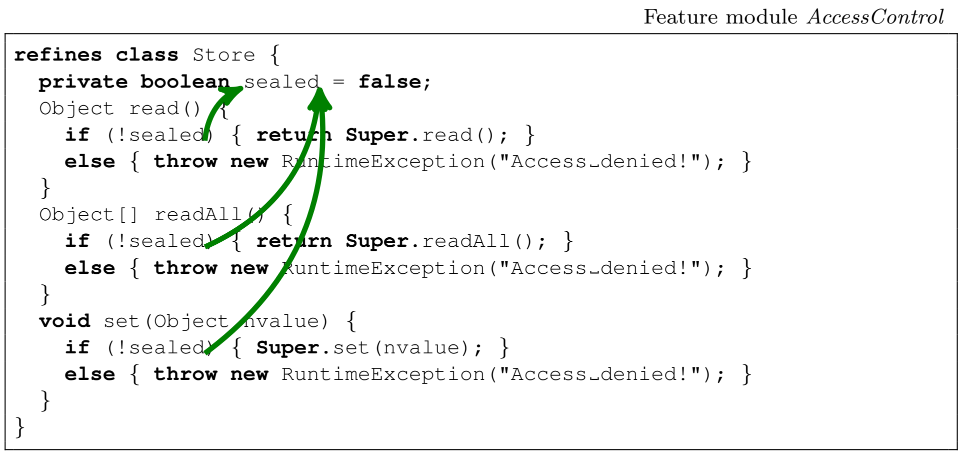
Advantages and Disadvantages. Feature-based strategies have advantages compared to product-based and family-based strategies. First, they analyze domain artifacts (similar to family-based strategies) instead of operating on generated software artifacts and thus there are no redundant computations for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

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

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

Fourth, the analysis of a software product line using a feature-based strategy is divided into smaller analysis tasks. Thus, a feature-based strategy is especially useful for software analysis with extensive resource consumption (e.g., memory) and for large software product lines, for which family-based analysis are not feasible.

Finally, changing the variability model does not affect feature-based analysis at all. Hence, when the variability model evolves, we do not need to perform any



Feature module *AccessControl*

```
refines class Store {
  private boolean sealed = false;
  Object read() {
    if (!sealed) { return Super.read(); }
    else { throw new RuntimeException("Access␣denied!"); }
  }
  Object[] readAll() {
    if (!sealed) { return Super.readAll(); }
    else { throw new RuntimeException("Access␣denied!"); }
  }
  void set(Object nvalue) {
    if (!sealed) { Super.set(nvalue); }
    else { throw new RuntimeException("Access␣denied!"); }
  }
}
```

Fig. 5. Feature-based type checking reasons about features in isolation. For example, references to `sealed` can be checked entirely within feature *AccessControl*. But, references to `read()` and `readAll()` cut across feature boundaries and cannot be checked feature-based.

feature-based analysis again, since features are only analyzed in isolation.

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

However, as indicated previously, there are some strict feature-based strategies. Parsing and syntax checking of software product lines with modular implementations for each feature (such as feature-oriented programs, aspect-oriented programs, delta-oriented programs, and frameworks), is a compositional analysis and can be done feature-based. While parsing is a necessary task for any static analysis, it is only discussed in for non-modular feature implementations such as conditional compilation [Kästner et al. 2011], for which feature-based parsing is impossible. A further example for a simple feature-based analysis is to compute code metrics.

3.4 Combined Analysis Strategies

We have discussed product-based, family-based, and feature-based as different strategies to analyze software product lines. These three strategies form the basis of our classification, but they can also be combined resulting in four further strategies. In the following, we discuss all possible combinations even if some are not yet implemented, but might be in future.

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

Definition 3.4 *Feature-product-based analysis.* An analysis of a software product line is called *feature-product-based*, if (a) it consists of a feature-based analysis followed by a product-based analysis, and (b) the analysis results of the feature-based analysis are used in the product-based analysis.

Example. In our object store, we could start to type-check all features in isolation. As shown in Figure 5, we can check that all intra-feature references are valid and create an interface for every feature. The interface contains all methods, fields, and classes that the feature provides and also those that are required. In the second step, we take these interfaces and iterate over every valid combination of features and check whether the interfaces are compatible (i.e., everything that is required in some interface is provided by another interface). Thus, we can save redundant checks for intra-feature references.

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

Feature-Product-Based Type Checking. Apel and Hutchins [2010] define a calculus including a feature-product-based type system for the composition of feature modules: First, each feature module is type-checked in isolation, producing interfaces, second, a linker checks valid compositions of interfaces following a product-based strategy.

Bettini et al. [2010] propose a feature-product-based type system for Featherweight Record-Trait Java, supporting the implementation of software product lines using traits and records. Units of product functionality are modeled by traits, which only need to be type-checked once for the software product line. Besides this feature-based analysis, it is necessary to check that all traits are compatible pursuing an unoptimized product-based strategy.

Schaefer et al. [2011] propose a compositional type system for delta-oriented product lines. They present the minimal core calculus for delta-oriented programming in Java and define a constraint-based type system for the calculus. The type system generates a set of constraints for each delta, which need to be checked for each product in the second step.

Feature-Product-Based Model Checking. Fisler and Krishnamurthi [2001], Li et al. [2002], Li et al. [2005], and Liu et al. [2011] propose feature-product-based model checking. First, each feature is model-checked in isolation and an interface is generated specifying the provided behavior and the assumed behavior of other features. Then, these interfaces are checked for every product to make sure that features are compatible with each other. In other words, if the composed features satisfy the constrains, the properties of the considered features are maintained.

Poppleton [2007] uses Event-B for the specification of feature-oriented systems by transition systems. Event-B specifications can be verified using model checking, theorem proving, or both. Using a feature-product-based analysis, properties are proven about features in isolation, and for every composed product, it must be verified that proven correctness properties are preserved.

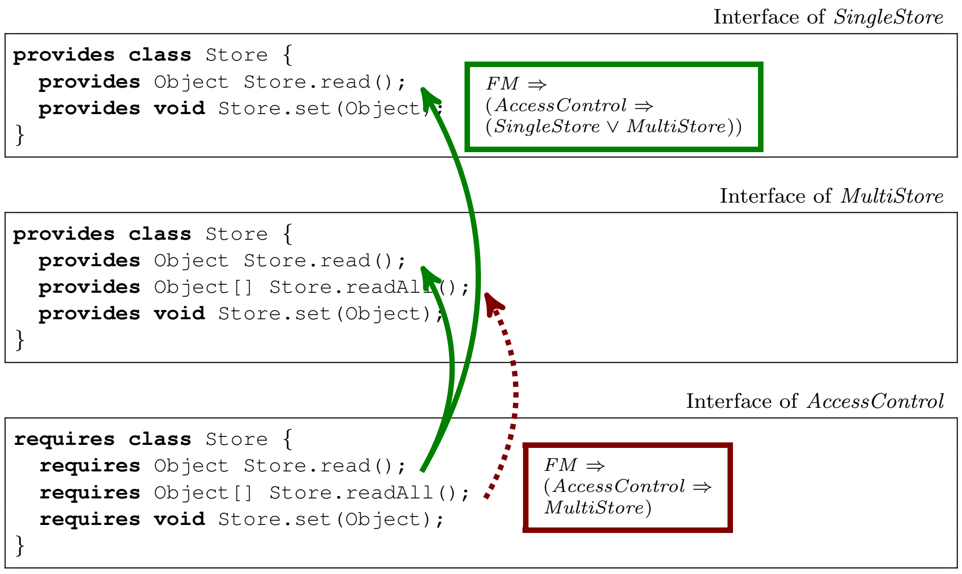
Feature-Product-Based Theorem Proving. Batory and Börger [2008] propose feature-product-based theorem proving to prove that a given Java interpreter is equivalent to the JVM interpreter for Java 1.0. They modularize the Java grammar, theorems about correctness, and natural language proofs into feature modules. Besides the modularization, a human still needs to check that every product has a valid grammar, correctness theorems, and natural language proof.

Similarly, Delaware et al. [2011] propose feature-product-based theorem proving for a product line of type-safety proofs. They propose a product line of languages based on Featherweight Java for which language features, such as generics, interfaces, or casting, can be chosen independently. All eight Featherweight Java variants are proven to be type safe in a feature-product-based manner. First, theorems are created and proved for each feature. Second, these theorems are used to prove progress and preservation for each Featherweight Java variant.

Thüm et al. [2011] propose feature-product-based theorem proving for verification in feature-oriented programming. Features are implemented in feature modules based on Java and specified using the Java Modeling Language (JML). The verification is based on the verification framework Why and the proof assistant Coq. A human has to provide partial proofs in Coq along with every feature. These proofs are then *automatically* checked for each product.

3.4.2 *Feature-Family-Based Analyses.* A strategy that is similar to feature-product-based analysis, is to combine feature-based and family-based analyses. The idea of feature-family-based analysis is to analyze features separately followed by a family-based analysis analyzing everything that could not be analyzed in isolation (based on properties inferred from the feature-based analysis).

Definition 3.5 *Feature-family-based analysis.* An analysis of a software product line is called *feature-family-based*, if (a) it consists of a feature-based analysis followed by a family-based analysis and (b) the analysis effort of the feature-based analysis is used in the family-based analysis.


Interface of *SingleStore*
$FM \Rightarrow (AccessControl \Rightarrow (SingleStore \vee MultiStore))$
Interface of *MultiStore*
Interface of *AccessControl*
$FM \Rightarrow (AccessControl \Rightarrow MultiStore)$

```
provides class Store {
  provides Object Store.read();
  provides void Store.set(Object);
}
```

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

```
requires class Store {
  requires Object Store.read();
  requires Object[] Store.readAll();
  requires void Store.set(Object);
}
```

Fig. 6. Feature-family-based type checking analyses features in isolation and applies family-based type checking on the feature interfaces afterwards. The references to `read()` and `readAll()` cut across feature boundaries and are checked at composition time based on the features' interfaces and the variability model.

Example. In our object store, we can infer interfaces for each feature using feature-based type checking and check these interfaces for compatibility using family-based type checking. The interface of each feature defines the program elements it provides and the program elements it requires (see Figure 6). For example, feature *AccessControl* requires a method `read()` which is provided either by feature *SingleStore* or feature *MultiStore*. But, method `readAll()` required by feature *AccessControl* is not available in all products with feature *AccessControl*. Basically, we can create a propositional formula for each reference which can be checked using a satisfiability solver as described in Section 3.2.

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

Feature-Family-Based Type Checking. Delaware et al. [2009] propose a constraint-based type system for Lightweight Feature Java, an extension of Lightweight Java with support feature-oriented programming. Type checking using their constraint-based type system works in two phases. First, all features are analyzed without considering the variability model and constraints for each feature are retrieved. The constraints describe type references and dependencies that must be fulfilled by other features. Second, a propositional formula is created describing the set of well-typed feature combinations, which is then compared to the variability model to retrieve whether all valid feature combinations according variability model are well-typed. We argue that this strategy is feature-family-based, because constraints can be retrieved for every feature in *isolation*.

Feature-Family-Based Theorem Proving. Hähnle and Schaefer [2011] present a feature-family-based approach for deductively verifying delta-oriented product lines. They restate the Liskov principle known from object-oriented programming to delta-oriented product lines which requires that method contracts introduced by deltas occurring later in the application ordering may only be more specific than the contracts introduced by previous deltas. The presented compositional verification principle allows verifying the specification of each delta in isolation by approximating called methods that are not defined in the delta itself by the specification of the first introduction of this method, either in the core product or in the first delta in the application ordering. In a further step, all deltas are checked for conformance in a family-based fashion.

3.4.3 *Family-Product-Based Analyses.* We have discussed several feature-product-based analyses. This combination is useful, because a solely feature-based analysis is often not sufficient to analyze a software product line as a whole. A family-product-based analysis may not seem useful at the first thought, because everything that can be analyzed product-based could already be analyzed family-based. But, family-product-based analyses can be useful (a) if a product-based analysis is faster for particular parts of the analysis, (b) if there is a part of the analysis (e.g., certain safety properties) that is relevant for one product or a small set of products only, (c) if several software analyses are combined, and (d) if the analysis problem for a family-based analysis is too large to be solved with given resource limitations such as physical memory.

Definition 3.6 *Family-product-based analysis.* An analysis of a software product line is called *family-product-based*, if (a) it consists of a (partial) family-based analysis followed by a product-based analysis and (b) the analysis effort of the family-based analysis is reused in the product-based analysis.

Kim et al. [2011] propose a family-product-based analysis for feature-oriented programming. They apply a family-based static analysis followed by an optimized product-based testing. In the first phase, they calculate a set of products for each test case that is sufficient to cover all possible test results. They extend control-flow and data-flow analyses with variability information to trace the effect of features. In the second phase, they generate products for which tests need to be executed and execute only necessary tests.

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

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

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

3.5 Summary

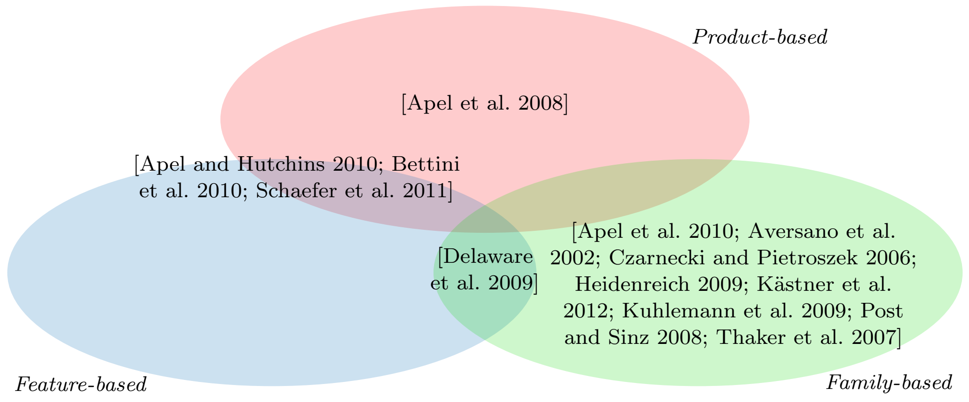
We presented a classification of product-line analyses consisting of three basic strategies and four combined strategies. Furthermore, we classified existing approaches that scale type checking, model checking, and theorem proving from single software products to software product lines. We highlighted advantages and disadvantages of each strategy. In Figure 7, we give an overview on the approaches we have classified – grouped by type checking, model checking, and other analyses. Based on this overview, we can make some observations regarding new and underrepresented research areas.

First, none of the surveyed analysis approaches is solely feature-based. The reason is that analyzing features only in isolation is usually not sufficient for type checking, model checking, and theorem proving, if the properties are not compositional. Usually, compositionality is a very restrictive property. There are several approaches for all software analyses that use a feature-product-based strategy. While this is an intuitive strategy, it still involves redundant computations at the product-based part of the analysis and is still infeasible for software product lines with a huge number of products such as the Linux kernel.

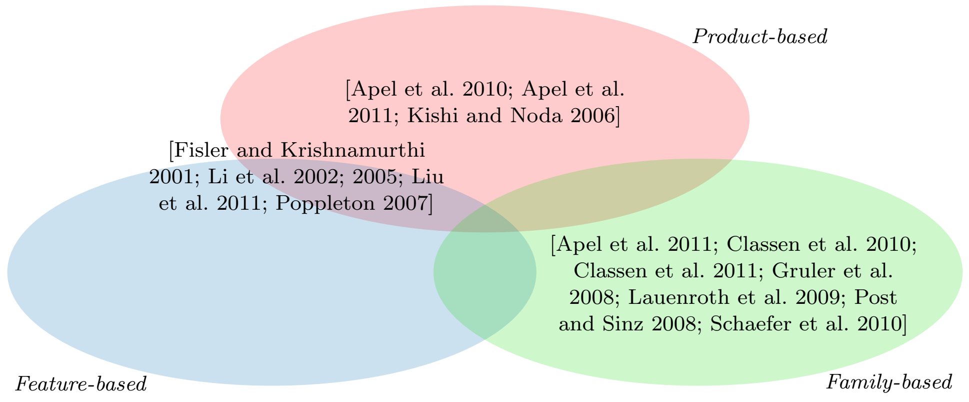
Second, for type checking and model checking there is a large number of family-based strategies. But, we found not a single approach applying a family-based strategy to theorem proving. Hence, we identify the new research area of family-based theorem proving. Future research shall either present such approaches or argue why this it is not possible to fill this gap. But, we are optimistic that such approaches are feasible.

Third, feature-family-based and family-product-based strategies seem to be young research areas that are still underrepresented. A feature-family-based strategy has been proposed in one approach for type checking and one for theorem proving. A family-product-based strategy has been proposed for combining static analysis with testing. Both strategies should be applied to other software analyses to evaluate the feasibility of each strategy. Especially feature-family-based analysis seem to have great potential as they combine open-world with closed-world scenarios while avoiding redundant calculations.

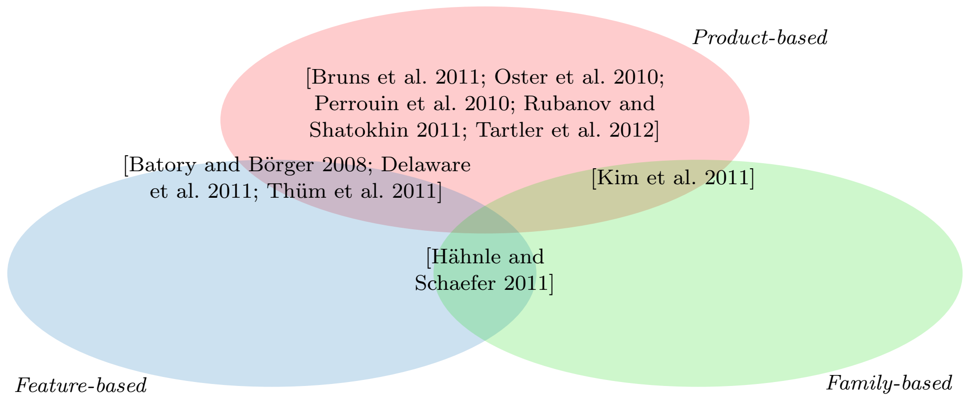
Finally, there are no approaches following a feature-family-product-based strategy. It is not yet clear whether such a strategy is useful at all, but the combination of all three basic strategies can have advantages when combining several software analyses. Again, approaches using type checking, model checking, or theorem prov-


Product-based
[Apel et al. 2008]
[Apel and Hutchins 2010; Bettini et al. 2010; Schaefer et al. 2011]
[Delaware et al. 2009]
[Apel et al. 2010; Aversano et al. 2002; Czarnecki and Pietroszek 2006; Heidenreich 2009; Kästner et al. 2012; Kuhlemann et al. 2009; Post and Sinz 2008; Thaker et al. 2007]
Feature-based
Family-based

(a) Type checking


Product-based
[Apel et al. 2010; Apel et al. 2011; Kishi and Noda 2006]
[Fisler and Krishnamurthi 2001; Li et al. 2002; 2005; Liu et al. 2011; Poppleton 2007]
[Apel et al. 2011; Classen et al. 2010; Classen et al. 2011; Gruler et al. 2008; Lauenroth et al. 2009; Post and Sinz 2008; Schaefer et al. 2010]
Feature-based
Family-based

(b) Model checking


Product-based
[Bruns et al. 2011; Oster et al. 2010; Perrouin et al. 2010; Rubanov and Shatokhin 2011; Tartler et al. 2012]
[Batory and Börger 2008; Delaware et al. 2011; Thüm et al. 2011]
[Kim et al. 2011]
[Hähnle and Schaefer 2011]
Feature-based
Family-based

(c) Other analyses

Fig. 7. Classification of analysis strategies for software product lines.

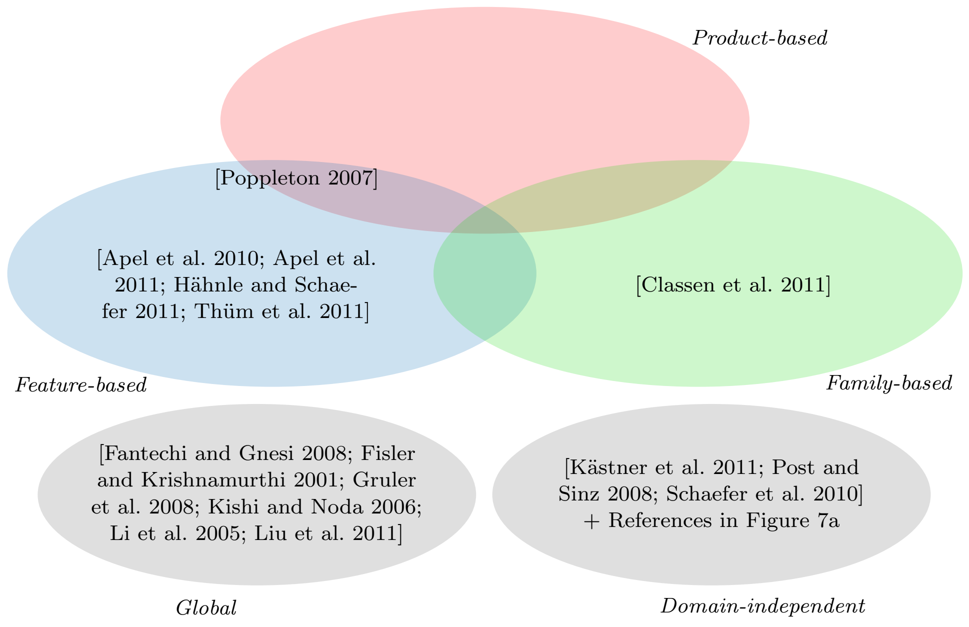

Product-based
[Poppleton 2007]
[Apel et al. 2010; Apel et al. 2011; Hähnle and Schaefer 2011; Thüm et al. 2011]
[Classen et al. 2011]
Feature-based
Family-based
[Fantechi and Gnesi 2008; Fisler and Krishnamurthi 2001; Gruler et al. 2008; Kishi and Noda 2006; Li et al. 2005; Liu et al. 2011]
[Kästner et al. 2011; Post and Sinz 2008; Schaefer et al. 2010] + References in Figure 7a
Global
Domain-independent

Fig. 8. Classification of specification strategies for software product lines.

ing should be developed and evaluated to assess the feasibility of this strategy.

4. SPECIFICATION STRATEGIES FOR SOFTWARE PRODUCT LINES

So far we discussed how to apply software analyses to software product lines. Some of these analyses, such as model checking and theorem proving, require specifications of the intended product behavior. Thus, we need to adapt specification approaches to software product lines, as well. We apply our classification schema also to specification approaches. In Figure 8, we give an overview the classified specification approaches, which we discuss in the following.

4.1 Product-based Specification

A software product line can be specified by specifying each software product individually. We call this strategy *product-based specification.* Clearly, specifying the behavior for every product scales only for software product lines with few products. As for product-based analyses, an optimization could be to specify and analyze only a subset of all products, which is applicable if only this subset is used productively. We did not find any strict product-based specification approach in the literature, but every specification approach for software may be applied to single products. Product-based specifications may be useful if the product specifications are largely disjunct, and thus there is a low potential to reuse specifications over several products.

4.2 Feature-based Specification

A more common strategy for the specification of software product lines is *feature-based specification.* Every feature is specified without any explicit reference to other features. Hence, feature-based specification can be used to define the expected behavior of each feature in isolation without referring to other features. Note that feature-based specifications can be used to verify properties across features (e.g., feature interactions can be detected [Apel et al. 2010]). Feature-based specifications were used for model checking [Apel et al. 2010; Apel et al. 2011], theorem proving [Thüm et al. 2011], and multiple analyses [Thüm et al. 2012]. Poppleton [2007] propose a feature-product-based specification that allows to specify each feature and to enrich the derived specification for every product manually.

4.3 Family-based Specification

The idea of family-based analyses can also be applied to specifications. Given that we know all features in advance (closed-world assumption), we can provide a specification that covers the entire product line while it has variable parts referring to individual features or feature combinations. Basically, we can provide specifications together with an *application condition* which is a propositional formula on the features. Alternatively, features can be referenced directly in the specification. For example consider our object store, we might want to specify that objects cannot be accessed using method `readAll()`, if the store is sealed *and* the product contains the features *MultiStore* and *AccessControl*. Thus, we can define specifications not only depending on the presence of a single feature, but also for any subset of products (e.g., all products containing the features *MultiStore* and *AccessControl*). In fact, this approach generalizes product-based and feature-based specifications, in a sense that each product-based and each feature-based specification is a family-based specifications per definition. Family-based specifications are used for model checking [Classen et al. 2011].

4.4 Global Specification

Besides applying our analysis strategies to specification, there is another strategy to define specifications for software product lines. A *global specification* is a specification that all products of a software product line need to fulfill. For example in a product line of pacemakers, all products have to admit to the same specification stating that a heart beat is generated whenever the heart stops beating [Liu et al. 2007]. Global specifications were used to define specifications of software product lines for model checking [Fisler and Krishnamurthi 2001; Li et al. 2005; Kishi and Noda 2006; Liu et al. 2011].

4.5 Domain-Independent Specification

While a specification is typically tailored to a certain software product line, for some analyses, it is sufficient to define one specification for all domains. A *domain-independent specification* is a specification that is usually specific to a certain programming language and is assumed to hold for all programs in that language. A prominent example for a domain-independent specification is a type system, since it is assumed to hold for every software product line written using a particular product-line implementation technique and programming language. Hence, all discussed approaches for type checking software product lines apply a domain-independent specification (see Figure 7). Further examples for domain-independent specifications are parsers (i.e., syntax conformance) [Kästner et al. 2011], the absence of runtime exceptions [Post and Sinz 2008], or that every program statement in a software product line appears in at least one product [Tartler et al. 2011].

5. CONCLUSION AND FUTURE WORK

Software-product-line engineering aims at the development of similar software products in an efficient and coordinated manner. Implementation artifacts are rigorously reused in a planned way. While there are several efficient methods to implement software product lines, current research seeks to scale software analyses such as type checking, statical analyses, model checking, or theorem proving from single products to entire software product lines. The field of product-line analyses is diverse, and proposed approaches are often hard to compare.

We proposed a classification of product-line analysis into three main strategies: product-based, feature-based, and family-based analysis. Combined they result in a total of seven different strategies. We classified 38 existing analysis and specification approaches gaining insights into the field of product-line analyses. First, feature-based analyses are usually only applicable in combination with product-based or family-based analyses, because of non-compositional properties. Second, while there are many family-based analyses for type checking and model checking, there is not a single approach for family-based theorem proving. Third, we identified several combined strategies such as feature-family-based analyses that are under-represented research areas and also proposed feature-family-product-based analysis as a new strategy that is not yet existent in the literature.

Our experience with surveying the literature on the analysis and specification of software product lines is that it is not easy to find a proper classification. A classification should help to distinguish and group analysis approaches according to the essential ideas behind each approach. Of course, this is not a trivial task. We refined our classification several times while classifying existing approaches. We believe a community effort is necessary to agree on a common classification and to synchronize research efforts in order to develop novel product-line analyses – with our classification, we make a first, but well-thought proposal. We refer interested readers to our website to follow the progress of our classification effort.[2]

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

—Which strategies are not yet applied to which software analysis?

—What can we learn from strategies of one analysis for other analyses?

—What are the strengths and weaknesses of the analysis strategies and what is their synergistic potential?

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

—Are there sound theoretical foundations and reliable empirical results for every approach?

—Are there proper tools available for every combination of strategy and software analysis?

—Which combinations of strategies and software analyses are of interest to research and practice of product-line analysis?

—Which strategies can be combined and what are useful combinations?

—Are there further novel analysis strategies?

For example, we envision further dimensions of our classification, which have to be discussed in the community: Which properties of a product line can we check efficiently? Which artifact types should be considered by the analyses?

A classification of product-line analysis approaches helps developers to choose and combine proper analysis tools for a given problem. For example, type checking feature implementations in isolation is especially beneficial if the features are large and refer only sporadically to other features. Of course, practitioners rely on a community endeavor of "filling" all dimensions, explore the individual strengths and weaknesses of each of them (e.g., in a family-based approach all features have to be known in advance), and develop proper tools that can be combined at wish.

We hope this article can raise awareness of the importance and challenges of product-line analysis, initiate a discussion on the future of product-line analysis, motivate researchers to explore and practitioners to use product-line analysis methods, and help to form a community of researchers, tool builders, and users interested in product-line analysis.

ACKNOWLEDGMENTS

Apel's work is supported by the German Research Foundation (DFG – AP 206/2, AP 206/4, and LE 912/13). Kästner's work is supported by ERC grant #203099. Saake's work is supported by the German Research Foundation (DFG – SA 465/34).

REFERENCES

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

Apel, S. and Kästner, C. 2009. An Overview of Feature-Oriented Software Development. *J. Object Technology (JOT) 8,* 5, 49–84.

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

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

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

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

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

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

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

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

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

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

Berger, T., She, S., Lotufo, R., Wasowski, A., and Czarnecki, K. 2010. Variability Modeling in the Real: a Perspective from the Operating Systems Domain. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*. ACM, New York, NY, USA, 73–82.

Bettini, L., Damiani, F., and Schaefer, I. 2010. Implementing Software Product Lines Using Traits. In *Proc. ACM Symposium on Applied Computing (SAC)*. ACM, New York, NY, USA, 2096–2102.

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

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

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

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

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

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

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

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

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

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

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

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

Fisler, K. and Krishnamurthi, S. 2005. Decomposing Verification Around End-User Features. In *Proc. IFIP Working Conf. Verified Software: Theories, Tools, Experiments (VSTTE)*. Springer, Berlin, Heidelberg, New York, London, 74–81.

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

Hähnle, R. and Schaefer, I. 2011. A Liskov Principle for Delta-oriented Programming. In *Proc. Int'l Conf. Formal Verification of Object-Oriented Software (FoVeOOS)*. Technical Report 2011-26, Department of Informatics, Karlsruhe Institute of Technology, Karlsruhe, Germany, 190–207.

Heidenreich, F. 2009. Towards Systematic Ensuring Well-Formedness of Software Product Lines. In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD)*. ACM, New York, NY, USA, 69–74.

Heidenreich, F., Kopcsek, J., and Wende, C. 2008. FeatureMapper: Mapping Features to Models. In *Companion Int'l Conf. Software Engineering (ICSEC)*. ACM, New York, NY, USA, 943–944.

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute.

Kästner, C. 2010. Virtual Separation of Concerns: Toward Preprocessors 2.0. Ph.D. thesis, University of Magdeburg.

Kästner, C., Apel, S., Thüm, T., and Saake, G. 2012. Type Checking Annotation-Based Product Lines. *Trans. Software Engineering and Methodology (TOSEM)*. To appear.

Kästner, C., Apel, S., ur Rahman, S. S., Rosenmüller, M., Batory, D., and Saake, G. 2009. On the Impact of the Optional Feature Problem: Analysis and Case Studies. In *Proc. Int'l Software Product Line Conference (SPLC)*. Software Engineering Institute, Pittsburgh, PA, USA, 181–190.

Kästner, C., Giarrusso, P. G., Rendel, T., Erdweg, S., Ostermann, K., and Berger, T. 2011. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, New York, NY, USA, 805–824.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. 1997. Aspect-Oriented Programming. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*. Springer, Berlin, Heidelberg, New York, London, 220–242.

Kim, C. H. P., Batory, D. S., and Khurshid, S. 2011. Reducing Combinatorics in Testing Product Lines. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*. ACM, New York, NY, USA, 57—68.

Kishi, T. and Noda, N. 2006. Formal Verification and Software Product Lines. *Comm. ACM 49*, 73–77.

Kuhlemann, M., Batory, D., and Kästner, C. 2009. Safe Composition of Non-Monotonic Features. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*. ACM, New York, NY, USA, 177–186.

Kuhlemann, M. and Sturm, M. 2010. Patching Product Line Programs. In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD)*. ACM, New York, NY, USA, 33–40.

Lauenroth, K., Pohl, K., and Toehning, S. 2009. Model Checking of Domain Artifacts in Product Line Engineering. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*. IEEE, Washington, DC, USA, 269–280.

Li, H. C., Krishnamurthi, S., and Fisler, K. 2002. Interfaces for Modular Feature Verification. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*. IEEE, Washington, DC, USA, 195–204.

Li, H. C., Krishnamurthi, S., and Fisler, K. 2005. Modular Verification of Open Features Using Three-Valued Model Checking. *Automated Software Engineering (ASE) 12,* 3, 349–382.

Liu, J., Basu, S., and Lutz, R. 2011. Compositional Model Checking of Software Product Lines using Variation Point Obligations. *Automated Software Engineering (ASE) 18,* 1, 39–76.

Liu, J., Dehlinger, J., and Lutz, R. 2007. Safety Analysis of Software Product Lines using State-based Modeling. *J. Systems and Software (JSS) 80,* 11, 1879–1892.

Mendonca, M., Wasowski, A., and Czarnecki, K. 2009. SAT-based Analysis of Feature Models is Easy. In *Proc. Int'l Software Product Line Conference (SPLC)*. Carnegie Mellon University, Pittsburgh, PA, USA, 231–240.

Nielson, F., Nielson, H. R., and Hankin, C. 2010. *Principles of Program Analysis.* Springer, Secaucus, NJ, USA.

Oster, S., Markert, F., and Ritter, P. 2010. Automated Incremental Pairwise Testing of Software Product Lines. In *Proc. Int'l Software Product Line Conference (SPLC).* Springer, Berlin, Heidelberg, New York, London, 196–210.

Perrouin, G., Sen, S., Klein, J., Baudry, B., and le Traon, Y. 2010. Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines. In *Proc. Int'l Conf. Software Testing, Verification and Validation (ICST).* IEEE, Washington, DC, USA, 459–468.

Pierce, B. C. 2002. *Types and Programming Languages.* MIT Press, Cambridge, Massachusetts, USA.

Pohl, K., Böckle, G., and van der Linden, F. J. 2005. *Software Product Line Engineering : Foundations, Principles and Techniques.* Springer, Berlin, Heidelberg, New York, London.

Poppleton, M. 2007. Towards Feature-Oriented Specification and Development with Event-B. In *Proc. Int'l Working Conf. Requirements Engineering: Foundation for Software Quality (REFSQ).* Springer, Berlin, Heidelberg, New York, London, 367–381.

Post, H. and Sinz, C. 2008. Configuration Lifting: Software Verification meets Software Configuration. In *Proc. Int'l Conf. Automated Software Engineering (ASE).* IEEE, Washington, DC, USA, 347–350.

Prehofer, C. 1997. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP).* Springer, Berlin, Heidelberg, New York, London, 419–443.

Rubanov, V. V. and Shatokhin, E. A. 2011. Runtime Verification of Linux Kernel Modules Based on Call Interception. In *Proc. Int'l Conf. Software Testing, Verification and Validation (ICST).* IEEE, Washington, DC, USA, 180–189.

Schaefer, I., Bettini, L., Bono, V., Damiani, F., and Tanzarella, N. 2010. Delta-Oriented Programming of Software Product Lines. In *Proc. Int'l Software Product Line Conference (SPLC).* Springer, Berlin, Heidelberg, New York, London, 77–91.

Schaefer, I., Bettini, L., and Damiani, F. 2011. Compositional Type-Checking for Delta-Oriented Programming. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD).* ACM, New York, NY, USA, 43–56.

Schaefer, I., Gurov, D., and Soleimanifard, S. 2010. Compositional Algorithmic Verification of Software Product Lines. In *Proc. Int'l Symposium on Formal Methods for Components and Objects (FMCO).* Springer, Berlin, Heidelberg, New York, London, 184–203.

Schumann, J. 2001. *Automated Theorem Proving in Software Engineering.* Springer, Berlin, Heidelberg, New York, London.

Sincero, J., Schirmeier, H., Schröder-Preikschat, W., and Spinczyk, O. 2007. Is The Linux Kernel a Software Product Line? In *Proc. Int'l Workshop Open Source Software and Product Lines (OSSPL)*, F. van der Linden and B. Lundell, Eds. IEEE, Washington, DC, USA, 9–12.

Tartler, R., Lohmann, D., Dietrich, C., Egger, C., and Sincero, J. 2012. Configuration Coverage in the Analysis of Large-Scale System Software. *ACM SIGOPS Operating Systems Review 45,* 3, 10–14.

Tartler, R., Lohmann, D., Sincero, J., and Schröder-Preikschat, W. 2011. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *Proc. Europ. Conf. Computer Systems (EuroSys).* ACM, New York, NY, USA, 47–60.

Thaker, S., Batory, D., Kitchin, D., and Cook, W. 2007. Safe Composition of Product Lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE).* ACM, New York, NY, USA, 95–104.

Thüm, T., Batory, D., and Kästner, C. 2009. Reasoning about Edits to Feature Models. In *Proc. Int'l Conf. Software Engineering (ICSE).* IEEE, Washington, DC, USA, 254–264.

Thüm, T., Schaefer, I., Kuhlemann, M., and Apel, S. 2011. Proof Composition for Deductive Verification of Software Product Lines. In *Proc. Int'l Workshop Variability-intensive Systems Testing, Validation and Verification (VAST).* IEEE, Washington, DC, USA, 270–277.

Thüm, T., Schaefer, I., Kuhlemann, M., Apel, S., and Saake, G. 2012. Applying Design by Contract to Feature-Oriented Programming. In *Proc. Int'l Conf. Fundamental Approaches to Software Engineering (FASE)*. Springer, Berlin, Heidelberg, New York, London, 255–269.

Weiss, D. M. 2008. The Product Line Hall of Fame. In *Proc. Int'l Software Product Line Conference (SPLC)*. IEEE, Washington, DC, USA, 395.