A Classification and Survey of Analysis Strategies for Software Product Lines

THOMAS THÜM, University of Magdeburg, Germany
SVEN APEL, University of Passau, Germany
CHRISTIAN KÄSTNER, Carnegie Mellon University, Pittsburgh, Pennsylvania
INA SCHAEFER, University of Braunschweig, Germany
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 family of software products that share a common set of features. Software product lines challenge traditional analysis techniques, such as type checking, model checking, and theorem proving, in their quest of ensuring correctness and reliability of software. Simply creating and analyzing all products of a product line is usually not feasible, due to the potentially exponential number of valid feature combinations. Recently, researchers began to develop analysis techniques that take the distinguishing properties of software product lines into account, for example, by checking feature-related code in isolation or by exploiting variability information during analysis. The emerging field of product-line analyses is both broad and diverse, so it is difficult for researchers and practitioners to understand their similarities and differences. We propose a classification of product-line analyses to enable systematic research and application. Based on our insights with classifying and comparing a corpus of 123 research articles, we develop a research agenda to guide future research on product-line analyses.

Categories and Subject Descriptors: A.1 [**General**]: Introductory and Survey; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and interfaces*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Correctness proofs, formal methods, model checking*; D.2.9 [**Software Engineering**]: Management—*Software configuration management*; D.2.13 [**Software Engineering**]: Reusable Software—*Domain engineering*; D.3.4 [**Software Engineering**]: Processors—*Code generation, compilers, incremental compilers, parsing*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Mechanical verification, specification techniques*

General Terms: Design, Experimentation, Reliability, Theory, Verification

Additional Key Words and Phrases: Product-line analysis, software product line, program family, software analysis, type checking, static analysis, model checking, theorem proving

ACM Reference Format:
Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A classification and survey of analysis strategies for software product lines. ACM Comput. Surv. 47, 1, Article 6 (May 2014), 45 pages.
DOI: http://dx.doi.org/10.1145/2580950

6

Apel's work is supported by the German Research Foundation (DFG – AP 206/2, AP 206/4, AP 206/5, AP 206/6, and AP 206/7). Kästner's work is supported by ERC grant #203099. Schaefer's work is supported by the German Research Foundation (DFG – SCHA 1635/2). Saake's work is supported by the German Research Foundation (DFG – SA 465/34).

DOI: http://dx.doi.org/10.1145/2580950

1. INTRODUCTION

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; Apel et al. 2013a]. A *software product line* (or program family [Parnas 1976]) 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 et al. 2013a].

Software-product-line engineering has gained considerable momentum in recent years, both in industry and in academia. Companies such as Boeing, Bosch, General Motors, Hewlett Packard, Philips, Siemens, and Toshiba apply product-line technology to broaden their software portfolio, increase return on investment, shorten time to market, and improve software quality [van der Linden et al. 2007; Weiss 2008; Lutz 2007]. Software product lines have been used successfully to build automotive gasoline systems, televisions, medical devices, and even power plants [Weiss 2008]. A prominent example from the open-source community that can be considered as a software product line is the Linux kernel with more than 11.000 features [Tartler et al. 2012].

Software-product-line engineering is increasingly used in safety- and mission-critical systems, including embedded, medical, automotive, and avionic systems [Weiss 2008]. Hence, proper quality assurance that provides correctness and reliability guarantees is 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, model checking, and theorem proving, can and needs to be applied to software product lines to obtain reliable software products. A simple strategy is to generate all software products of a product line and to 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 the products of a software product line typically have similarities. This inefficiency is especially a problem if products can be generated automatically from a common code base, because such product lines often contain a large set of products. Already for a product line with 33 independent, optional features, we can generate more products than people on earth; even if the analysis runs automatically and takes only 1 second for each product, the sequential analysis of the whole product line would take more than 272 years. 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 every single product.

Recently, researchers began developing 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, to make them *aware* of the variability and the features of a product line. The emerging field of *product-line analysis* is both broad and diverse. Hence, 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, while still others analyze the product line's code base as a whole. This breadth and diversity hinder systematic research and application.

We classify existing and ongoing work in the field of product-line analyses, compare techniques based on our classification, and infer a research agenda to guide further

research in this direction. Our long-term vision is to empower developers to assess and predict the analysis effort based on static characteristics of a software product line, such as the number of features, the number of products, or the complexity of feature implementations. Our short-term goals are (a) making research more systematic and efficient, (b) enabling tool developers to create new tools based on existing 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.

While our classification applies to a wide variety of software analyses, we focus on particular analyses in our survey for clarity: we concentrate on development techniques, with which products are generated automatically based on a feature selection. In contrast to the typically low number of products when manual assembly is required, automatic generation often leads to a huge number of products and thus is especially challenging for product-line analyses. Furthermore, we survey analysis approaches that operate statically, such as type checking, model checking, and theorem proving. Analyses that focus exclusively on requirements engineering and domain analysis (e.g., feature-model analysis) or that focus only on testing are outside the scope of this article; we refer the reader to dedicated surveys on feature-model analysis [Janota et al. 2008; Benavides et al. 2010] and on product-line testing [Tevanlinna et al. 2004; Engström and Runeson 2011; Da Mota Silveira Neto et al. 2011; Oster et al. 2011; Lee et al. 2012].

In summary, we make the following contributions.

—We propose a classification of product-line analyses.
—We survey and classify 123 existing approaches for the analysis of product lines.
—We infer a research agenda based on our insights with classification and survey.
—We offer and maintain a website to support the continuous community effort of classifying new approaches.[1]

2. PRELIMINARIES

In this section, we briefly introduce the necessary background for the following discussions. In Section 2.1, we present basic concepts of software product lines. In Section 2.2, we review software analyses that are crucial to build reliable software and that have been applied to product lines, as identified in our survey. In Section 2.3, we briefly discuss how specifications can be defined for software product lines as a basis for product-line analyses. Finally, we discuss the methodology of our survey in Section 2.4.

2.1. Software Product Lines

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

There is a broad variety of implementation mechanisms used in product-line engineering. For example, the developers of the Linux kernel combine 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; Svahnberg et al. 2005; Apel et al. 2013a]; all establish and maintain a mapping between features and implementation artifacts (such as models, code, test cases, and documentation). Apel et al. [2013a] distinguish between annotation-based implementation approaches, such as preprocessors, and composition-based implementation

[1]Project website: http://fosd.net/spl-strategies/.

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: the implementation is separated into multiple composition units.

approaches, such as black-box frameworks with plug-ins. In our running example, we use feature-oriented programming as a composition-based generation mechanism. However, the analysis strategies presented in this article are largely independent of the implementation approach.

A Running 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 a basic access-control mechanism that allows a client to seal and unseal the store and thus to control access to stored objects.

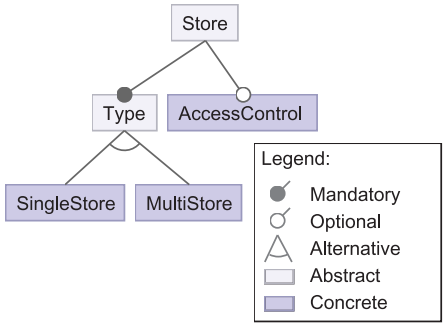
In Figure 1, we show the implementation of the three features of the object store using feature-oriented programming. In *feature-oriented programming*, each feature is implemented in a separate module called a feature module [Prehofer 1997; Batory et al. 2004]. A *feature module* is a set of classes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

Product {*MultiStore, AccessControl*}

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

Fig. 2. An object store composed from the feature modules *MultiStore* and *AccessControl*.



(a) Feature diagram

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

(b) Propositional formula

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

(c) Enumeration of all valid combinations

Fig. 3. The variability model of the object store in three alternative representations.

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 class refinement (denoted with refines class) is that a given class is extended with new methods and fields. Similar to subclassing, class refinement allows the programmer to override or extend existing methods. While the features *SingleStore* and *MultiStore* introduce only regular Java classes, feature *AccessControl* refines an existing class by adding new members. The result of the composition of the feature modules *MultiStore* and *AccessControl* is shown in Figure 2.

Variability Models. Decomposing the object store along its features gives rise to compositional flexibility; features can be composed in any combination. However, often not all feature combinations are desired; in our example, we must not select *SingleStore* and *MultiStore* in the same product. Product-line engineers typically specify constraints on feature combinations (a.k.a., *configurations*) in a variability model. In Figure 3(a), 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 each child feature depends on its parent feature [Kang et al. 1990]. We distinguish between concrete features, which are mapped to implementation artifacts, such as feature modules, and abstract features, which are only used to group other features [Thüm et al. 2011a]. In our example, each object store stores either a single object (feature *SingleStore*) or several (feature *MultiStore*). Furthermore, an object store may have the optional feature *AccessControl*. Valid feature combinations

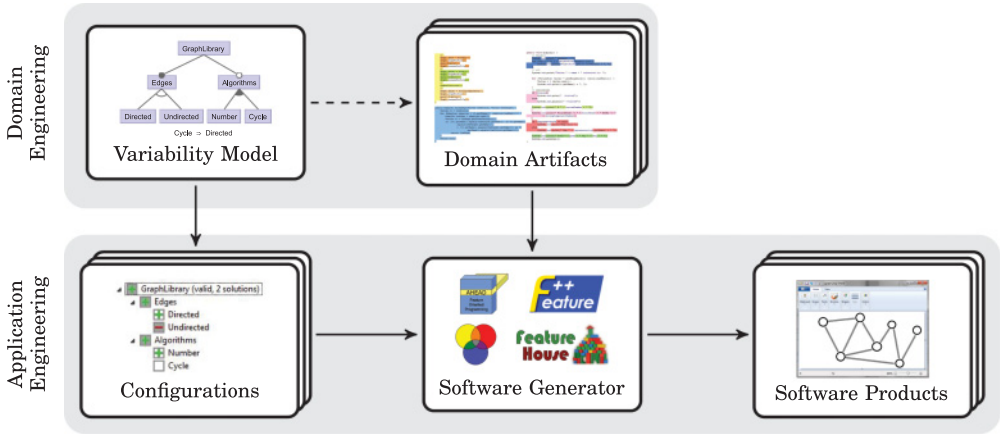


Fig. 4. In domain engineering, variability models and domain artifacts are created, which are used in application engineering to automatically generate software products based on feature selections.

can alternatively be specified using *propositional formulas* [Batory 2005], as shown in Figure 3(b); each variable encodes the absence or presence of a particular feature in the final product, and the overall formula yields *true* for all valid configurations. In our example, there are four configurations that are *valid* according to the variability model; they are enumerated in Figure 3(c)—yet another representation of a variability model, in which abstract features are usually omitted as they have no influence on generated products [Thüm et al. 2011a].

Automatic Product Generation from Domain Artifacts. In Figure 4, we illustrate the processes of domain engineering and application engineering (in a simplified form), both 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 domain artifacts. In *application engineering*, the developer determines a selection of features that serves the needs of the user best and that is valid according to the variability model. Based on this selection and the domain artifacts created during domain engineering, the software product containing the selected features is generated automatically. For example, composing the feature modules *SingleStore* and *AccessControl* results in a store tailored for a particular user.

In our 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* derives the corresponding product, without further user assistance, such as manual assembly or providing glue code. Examples of such implementation techniques are preprocessors [Liebig et al. 2010], generative programming [Czarnecki and Eisenecker 2000], feature-oriented programming [Prehofer 1997; Batory et al. 2004], and aspect-oriented programming [Kiczales et al. 1997]. The overall goal is to minimize the effort to tailor software products to the needs of the user.

Correctness of Software Product Lines. 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 feature *AccessControl* in combination with *MultiStore* as illustrated in Figure 2, the specification of feature *AccessControl* is violated; a client can access a store using method readAll even though the store is sealed.

There are several solutions to solve this misbehavior. We could modify the variability model to forbid the critical feature combination P_4, we could change the specification, or we could resolve the problem with alternative implementation patterns. For instance, we can alter the implementation of feature *AccessControl* by refining method readAll in analogy to methods read and set . While this change resolves the misbehavior when combining *MultiStore* and *AccessControl*, it introduces a new problem: the changed implementation of *AccessControl* no longer composes with *SingleStore*, because it attempts to override method readAll, which is not present in this configuration. The illustrated problem is called the *optional feature problem* [Liu et al. 2006; 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.

The point of our example is to illustrate how products can misbehave or cause type 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 combinations 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. Inconsistencies between the variability model and the implementation have repeatedly been observed in real product lines and are certainly not an exception [Thaker et al. 2007; Kästner et al. 2012a; Tartler et al. 2011; Kolesnikov et al. 2013; Medeiros et al. 2013]. Ideally, analysis strategies for software product lines are applied in domain engineering rather than application engineering to detect faults as early as possible.

2.2. Software Analyses

We briefly introduce important software analyses that have been applied and adapted to software product lines (from lightweight to heavyweight). We focus on analyses that operate statically; that is, we exclude runtime analyses and testing, because they are discussed in dedicated surveys [Tevanlinna et al. 2004; Engström and Runeson 2011; Da Mota Silveira Neto et al. 2011; Oster et al. 2011; Lee et al. 2012]. Each of the discussed analyses has its strengths and weaknesses. We argue that a wide variety of analyses are needed to increase the quality of software, in general, and software product lines, in particular. We discuss type checking, static analysis, model checking, and theorem proving. There are no clear distinctions between these analyses and gray zones between them, depending on individual definitions. For example, arguably they can all be defined as some form of abstract interpretation [Cousot and Cousot 1977]. We make a simple distinction based on commonly used terms.

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

tool-analyzing programs written in a certain language, usually part of a compiler or linker [Pierce 2002]. In model-driven development, type checking is essentially the analysis of well-formedness of a model with respect to its metamodel [Atkinson and Kühne 2003].

By means of 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 has not been declared. For our object store, we discussed that if we would call method readAll in feature *AccessControl*, then a dangling method reference would occur in product P_2. Other examples are that a programmer may have misspelled the name of a method, or the number of arguments is not correct.

A type system can be seen as a formal specification that all programs written in a certain language must conform to. Pierce [2002] argues that, in principle, types can be created to check arbitrary specifications. However, in practice, type systems are limited to properties that are efficiently statically decidable and checkable. Type checkers are typically included in compilers and scale to large programs, and then require no user input and can be fully automated.

Static Analysis. The term *static analysis* (a.k.a. program analysis) refers to analyses that operate at compile time and approximate the set of values or behaviors arising dynamically at runtime when executing a program [Nielson et al. 2010]. Examples for static analyses are traditional data-flow and control-flow analyses, but also alias analyses, program slicing, and constraint-based analyses [Weiser 1981; Muchnick 1997; Nielson et al. 2010]. A key technique in static analysis is that the undecidability of program termination due to loops or recursion is handled using approximation [Nielson et al. 2010].

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

The difference to type checking is that not every behavioral property of interest has to be encoded with types; the difference to other verification techniques, such as model checking or theorem proving, is that branches in programs are typically not interpreted and values are approximated. Similar to type checking, static analyses run automatically and often do not require user input such as providing a specification.

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

A model checker is a tool that performs a model-checking task given to a system to verify and its specification. Some model checkers require models with dedicated input languages for this task (e.g., Promela in SPIN [Holzmann 1997], CMU SMV in

NuSMV [Cimatti et al. 1999]), while others extract models directly from source code (e.g., C in Blast [Beyer et al. 2007] or CPAchecker [Beyer and Keremoglu 2011], Java in JPF[Visser et al. 2000]). After encoding a model-checking problem into the model checker's input language, the model-checking task is fully automated; each property either is stated valid or a counterexample is provided. The counterexample helps the user to identify the source of invalidity. The most severe practical limitation of model checkers is the limited size of the state space they can handle [Schumann 2001] (e.g., they may run out of time or main memory).

Model checking usually requires a model of the program input, which is not needed for type checking and static analyses. In addition, model checking usually scales only to much smaller programs than type checking and static analyses. Avoiding the state-space explosion requires manual effort for system abstraction or to configure heuristics of model checkers. Nevertheless, model checking can uncover faults that type checking and static analyses can not.

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 on them [Schumann 2001]; it assists the programmer in verifying the correctness of formulas, which can be achieved interactively or automatically. Interactive theorem provers, such as Coq [Bertot and Castéran 2004], PVS [Owre et al. 1992], and Isabelle/HOL [Nipkow et al. 2002], require the user to write commands applying inference rules. Instead, automated theorem provers, such as Prover9,[2] SPASS [Weidenbach et al. 2009], and Simplify [Detlefs et al. 2005], try to evaluate the validity of theorems without further assistance by the user. Theorem provers usually provide a language to express logical formulas (theorems). Additionally, interactive theorem provers also need to provide a language for proof commands. Automated theorem provers are often limited to first-order logic or subsets thereof, whereas interactive theorem provers are available for higher-order logic and typed logic. Theorem provers are able to generate proof scripts containing deductive reasoning that can be inspected by humans.

Theorem provers are used in many applications because of their high expressiveness and generality. In the analysis of software products, theorem provers are used to formally prove that a program fulfills its specification. Given a specification in some formal language and an implementation, a verification tool generates theorems, which are the input for the theorem prover. If a theorem cannot be proved, theorem provers point to the part of the theorem that could not be proved.

Compared to other verification techniques, the main disadvantage of theorem proving is that experts with an education in logical reasoning and considerable experience are needed [Clarke et al. 1999]. Even if the verification procedure can be fully automated in some cases, users still need experience to define formal specifications. Contrary to type checking and static analysis, model checking and theorem proving often do not scale to large programs.

2.3. Product-Line Specification

Many software analyses, such as model checking and theorem proving, require specifications defining the expected behavior of the programs to analyze. These analyses check the conformance of the actual behavior of a given program with the expected behavior. While surveying the literature, we identified different strategies to define

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

specifications for product-line analyses. We briefly present each specification strategy and will use them to classify approaches for product-line analyses in later sections.

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

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

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

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

Family-Based Specification. Finally, it is also possible to define specifications that particular subsets of all products have in common—referred to as *family-based specification*. In a family-based specification, we can specify properties of individual features or feature combinations. Basically, we can provide specifications together with a *presence condition*, which describes a subset of all valid configurations (e.g., by a propositional formula). Alternatively, features can be referenced directly in the specification. For example, in our object store, we might want to specify that objects cannot be accessed using method readAll if the store is sealed *and* the product contains the features *MultiStore* and *AccessControl*. In fact, family-based specification generalizes family-wide, product-based, and feature-based specifications, in a sense that such specifications can be expressed as special family-based specifications. With a family-based specification, we can automatically generate specifications of individual products, similar to product generation. Several family-based specifications require extensions to existing specification techniques [Asirelli et al. 2012; Classen et al. 2013], as features are referenced explicitly to model variability in properties.

2.4. Classification and Survey Methodology

Based on the introduction of software product lines, software analyses, and strategies for product-line specification, we present an overview of our classification of product-line analyses. Then, we explain the methodology used to perform our literature survey.

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

In our survey, we classify existing product-line analyses based on how they attempt to reduce analysis effort—the *analysis strategy*. We distinguish three basic strategies, indicating whether the analysis is applied to products, features, or the whole product line: product-based, feature-based, and family-based analyses. We explain the basic strategies and discuss existing approaches implementing each strategy. While surveying the literature, we found approaches that actually combine some of the basic strategies. Hence, we discuss possible combinations as well. For each strategy, we provide a definition and an example, we discuss advantages and disadvantages, and we classify existing approaches. Our main classification identifying the underlying analysis strategy is presented in Sections 3–6. Besides the main classification, we distinguish approaches also based on implementation strategies (see Section 2.1), the applied software analysis (see Section 2.2), and specification strategies (see Section 2.3).

We reached our classification in an iterative process, in which we repeatedly drafted a classification and classified articles accordingly. We collected relevant articles from research literature guided by our knowledge and experience—we have all actively worked in the field of product-line analyses for several years. In addition, we discussed analyses for software product lines at the Dagstuhl meetings 11021 and 13091, and we asked for contributions—several researchers tagged relevant articles in the online repository of `researchr.org`.[3] In our survey, we include articles independent of the time being published and the kind of publication (e.g., article in journal, conference, or technical report). The oldest articles we found were published in 2001 [Klaeren et al. 2001; Fisler and Krishnamurthi 2001; Plath and Ryan 2001; Nelson et al. 2001]. We assigned each article to, at least, two of the authors, who summarized the approach and analysis strategy; each time we sought interpersonal consensus to ensure validity. In case of doubt, we discussed the article with all authors of this survey or contacted the original authors of the article and refined the classification. We repeated the process until we reached consensus.

For clarity, we decided to remove articles subsumed by newer articles. An article is considered as subsumed if a follow-up article by the same authors is classified identically and the presented analyses are similar. As a consequence, while we classified 123 articles in total, we discuss only 90 articles in our survey. We set up a website

[3]http://researchr.org/tag/variability-aware-analysis/.

presenting our results including subsumed papers, and we invite other researchers to contribute to the ongoing process of classifying research on product-line analyses.[4]

3. PRODUCT-BASED ANALYSES

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

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

3.1. Example

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

3.2. Advantages and Disadvantages

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

A specific advantage of an unoptimized product-based analysis is the soundness and completeness with respect to the analysis that is scaled from single-system engineering (i.e., the *base analysis*). First, every fault detected using this strategy is a fault of a software product that can be detected by the base analysis (soundness). Second, every fault that can be detected using the base analysis is also detected using an unoptimized product-based analysis (completeness). Note that while the base analysis itself might be unsound or incomplete with regard to some specification and analysis goal, this strategy is still sound and complete with regard to the base analysis (i.e., it will detect the same faults).

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

The analysis results of product-based analyses refer necessarily to generated artifacts of products, and not to domain artifacts implemented in domain engineering,

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

which gives rise to two difficulties. First, a programmer may need to read and understand the generated code in order to understand the analysis results (e.g., the composed class Store in Figure 2 contains all members introduced by the features of the analyzed product). Second, if a change to the code is necessary, it must be applied to the domain artifacts instead of generated artifacts, and automatic mappings are not always possible [Kuhlemann and Sturm 2010].

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

3.3. Unoptimized Product-Based Analyses

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

In general, we found no proposal in the literature explicitly suggesting an unoptimized product-based analysis. However, we found some approaches that actually use product-based analyses for specific implementation mechanisms and do not discuss how to deal with many products; these approaches apply type checking [Apel et al. 2008a; Buchmann and Schwägerl 2012; Istoan 2013], static analyses [Klaeren et al. 2001; Scholz et al. 2011], model checking [Ubayashi and Tamai 2002; Kishi and Noda 2006; Fantechi and Gnesi 2008; Apel et al. 2010b; Istoan 2013; Bessling and Huhn 2014], and theorem proving [Harhurin and Hartmann 2008] to software product lines. The unoptimized product-based analysis strategy has been used with domain-independent specifications [Apel et al. 2008a; Buchmann and Schwägerl 2012; Istoan 2013], family-wide specifications [Ubayashi and Tamai 2002; Kishi and Noda 2006; Fantechi and Gnesi 2008; Istoan 2013], and feature-based specifications [Klaeren et al. 2001; Harhurin and Hartmann 2008; Apel et al. 2010b; Scholz et al. 2011; Istoan 2013; Bessling and Huhn 2014]. These approaches considered composition-based implementation [Klaeren et al. 2001; Ubayashi and Tamai 2002; Apel et al. 2008a; Scholz et al. 2011], composition-based design [Harhurin and Hartmann 2008; Apel et al. 2010b; Istoan 2013; Bessling and Huhn 2014], and annotation-based design [Kishi and Noda 2006; Fantechi and Gnesi 2008; Buchmann and Schwägerl 2012] as domain artifacts.

3.4. Optimized Product-Based Analyses

One reason for the success of software product lines is that new combinations of features can often be derived automatically. The effort for the development of new products is smaller than developing them from scratch. However, unoptimized product-based strategies hinder an efficient analysis of software product lines, and thus efficient development. The overall goal of product-line engineering is to scale product-line analyses to a similar degree of efficiency as implementation techniques, as the development of software product lines requires both efficient implementation strategies *and* efficient analysis strategies. Several optimized product-based strategies have been proposed to improve scalability and reduce redundant computations. Optimizations proposed in the literature focus either on detecting redundant parts in analyses or on eliminating

products that are already covered by other analysis steps, according to certain coverage criteria.

Optimized Product-Based Model Checking. Katz [2006] introduces aspect categories to optimize model checking of aspect-oriented programs. According to our classification, they discuss model checking for composition-based implementation and feature-based specification. In the first phase, a static analysis classifies aspects into spectative, regulative, and invasive aspects.[5] It is applied to individual products; Katz [2006] does not discuss how to handle many products. However, the static analysis in the first phase can save effort when model checking products in the second phase. He discusses which temporal properties can be influenced by spectative and regulative aspects. When analyzing certain properties using model checking, one can omit products that contain spectative and regulative aspects. Hence, we classify the two phases of this approach as unoptimized product-based static analysis and optimized product-based model checking. Similarly, Cordy et al. [2012d] discuss the notion of conservative features, which are features that do not remove behavior. They discuss which properties are preserved when adding conservative features to a product line. In their approach, they do not need to verify some properties for some products.

Optimized Product-Based Theorem Proving. Bruns et al. [2011] present a product-based analysis strategy for formal verification of delta-oriented software product lines. Their approach is based on contracts defined in delta modules, which we classify as feature-based specification and composition-based implementation. Delta modules are similar to feature modules but can also remove program elements. Bruns et al. [2011] generate all derivable software products and verify them incrementally using interactive theorem proving. To this end, 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 proved. However, in the end, still all products need to be generated and analyzed.

Sample-Based Analyses. Other approaches improve the efficiency of the product-based strategy by eliminating products from the set of products to analyze, because some products may already be covered by the analysis of other products. A frequently stated assumption is that most faults are caused by an interaction of only few features [Nie and Leung 2011; Kuhn et al. 2013]. Hence, those approaches retrieve a minimal set of products fulfilling a given coverage criterion and only those products are analyzed. While sampling is sound with respect to the base analysis, it is inherently incomplete (i.e., it may miss defects of not covered products). While most coverage criteria such as pair-wise or t-wise coverage are often proposed for testing of single systems [Nie and Leung 2011] and product lines [Oster et al. 2010; Perrouin et al. 2010], they have also been applied to scale-type checking [Jayaraman et al. 2007; Liebig et al. 2013], static analysis [Liebig et al. 2013], and model checking [Plath and Ryan 2001; Apel et al. 2013c] to software product lines.

In pair-wise coverage, for every pair of features (F, G), products must exist in the calculated set containing F but not G, G but not F, and both features F and G, whereas only combinations of features are considered that are valid according to the variability model. While pair-wise coverage can only detect all pair-wise interactions, t-wise

[5]Aspects can be used to implement features; the difference to feature modules is discussed elsewhere [Apel et al. 2008b].

coverage is a generalization of pair-wise coverage to detect higher-order interactions for up to t features. Sample-based analyses have been discussed for composition-based implementation [Apel et al. 2013c], composition-based design [Plath and Ryan 2001; Jayaraman et al. 2007], and annotation-based implementations [Liebig et al. 2013]. We classify their used specification strategies as domain-independent specification [Plath and Ryan 2001; Jayaraman et al. 2007; Liebig et al. 2013], family-wide specification [Liebig et al. 2013], and feature-based specification [Apel et al. 2013c]. Recent evaluations for type checking, static analysis, and model checking have shown that there are more efficient strategies for product-line analysis [Liebig et al. 2013; Apel et al. 2013c], which we discuss in the following sections.

4. FAMILY-BASED ANALYSES

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

Family-based. analyses operate on domain artifacts and valid combinations thereof, as specified by a 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 in concert, from which we can conclude that some intended properties hold for all products. Often, all implementation artifacts of all features are merged into a single virtual product (a.k.a. *metaproduct* or *product simulator*). The virtual product is not necessarily a valid product due to optional and mutually exclusive features [Thüm et al. 2012].

Definition 4.1 (*Family-Based Analysis*). An analysis of a software product line is *family based* if it (a) operates only on domain artifacts and (b) incorporates the knowledge about valid feature combinations.

4.1. Example

A family-based type checker, for instance, can analyze the code base of the object store example (i.e., all feature modules) in a single pass, although the features are combined differently in the individual products. To this end, it takes variability into account, in the sense that individual feature modules may be present or absent in certain products. Regarding method invocations, it checks whether a corresponding target method is declared in *every* valid product in which it is invoked. In Figure 5, we illustrate how a family-based type system checks whether the references of a slightly modified feature module *AccessControl* to the methods read and readAll are well typed in *every* valid product. For method read, the type system infers that the method is introduced by the feature modules *SingleStore* and *MultiStore*, and that one of them is always present (checked using a satisfiability solver; green, solid arrows).[6] For method readAll, it infers that the method is introduced only by feature module *MultiStore*, which may be absent when feature module *AccessControl* is present (red, dotted arrow). Hence, the type system reports a fault and produces a counterexample in terms of a valid feature selection that contains a dangling method invocation: {*SingleStore*, *AccessControl*}.

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

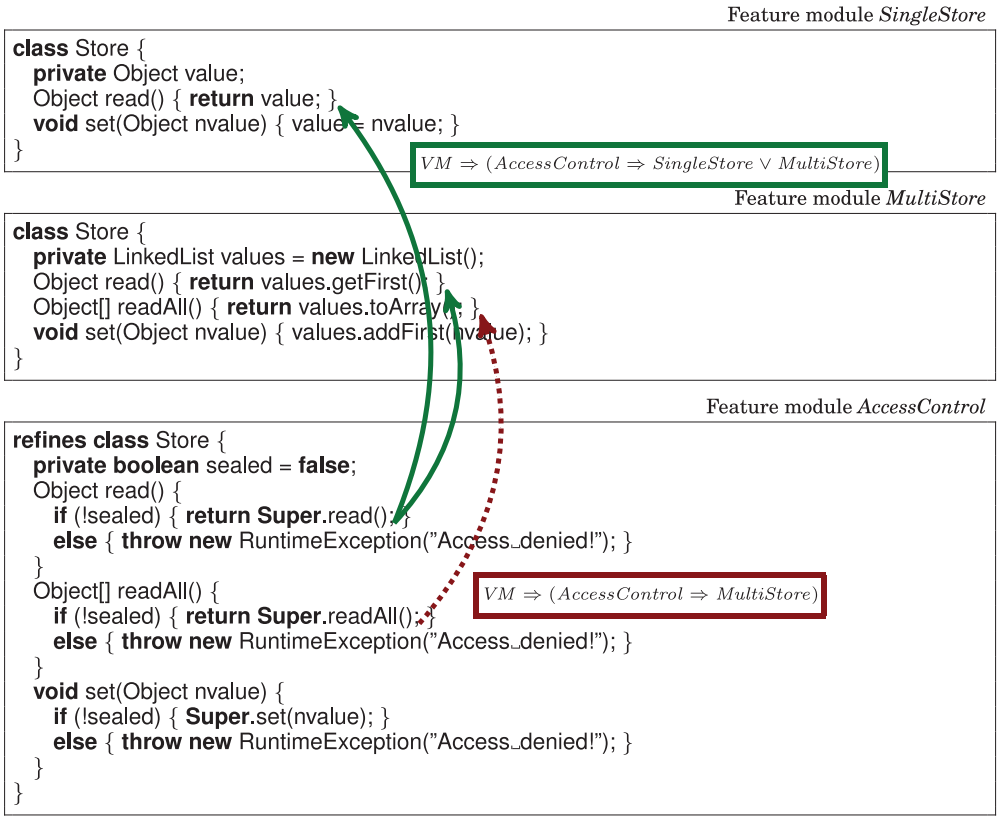


Fig. 5. Checking whether references to the methods read and readAll are well typed in *all* products. *VM* denotes the variability model of Figure 3 as propositional formula; a satisfiability solver determines whether the formulas in the boxes are tautologies (the upper formula is, but the lower is not).

4.2. Advantages and Disadvantages

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

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

However, family-based strategies also have disadvantages. Often, known analysis methods for single products cannot be used as they are. The reason is that the analysis method must be aware of features and variability. Existing analysis methods and off-the-shelf tools need to be extended, if possible, or new analysis methods need to be

developed. For some software analyses, such as model checking and theorem proving, there exist techniques to encode the analysis problem in an existing formalism or language (e.g., using a metaproduct simulating all products) and reuse off-the-shelf tools [Post and Sinz 2008; Apel et al. 2011; Thüm et al. 2012], but it is not clear whether these techniques can be used for all kinds of software analyses.

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

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

Fourth, as family-based analyses consider all domain artifacts as a whole, the size of the analysis problem can easily exceed physical boundaries such as the available memory [Apel et al. 2013c]. 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 multiteam development or software ecosystems such as Eclipse. Note, whenever we want to analyze the *whole* software product line, a closed world is required—independent of the chosen strategy.

4.3. Family-Based Syntax Checking

Although parsing detects only certain defects in source code (i.e., syntax conformance with respect to a domain-independent specification), it is a necessary step for many analyses such as type checking. While parsing is straightforward for modular product-line implementation approaches such as feature-oriented programming or aspect-oriented programming, it is complicated for product lines implemented with conditional compilation. There are several approaches for family-based parsing of C code with preprocessor directives that avoid preprocessing the code for each product separately by generating a variability-aware abstract syntax tree. Kästner et al. [2011] implemented their approach in TypeChef, and Gazzillo and Grimm [2012] presented SuperC for parsing. Gazzillo and Grimm [2012] compare the efficiency of both tools. Based on TypeChef, Medeiros et al. [2013] studied releases and commits of several open-source product lines, such as Bash, CVS, and Vim. They found defects that have remained unnoticed for years.

4.4. Family-Based Type Checking

Family-based strategies have been proposed by several authors for type checking of software product lines. The majority of work on family-based type checking is about creating variability-aware type systems (i.e., a domain-independent specification) and proving that, whenever a product line is type safe according to the type system, all derivable products are also well typed. The rules of these type systems contain reachability checks (basically implications) making sure, among others, that every program element is defined in all products where it is referenced. Variability-aware type systems have been developed for composition-based implementation [Thaker et al. 2007;

Huang et al. 2007; Kim et al. 2008; Kuhlemann et al. 2009; Delaware et al. 2009; Apel et al. 2010a, 2010c; Kolesnikov et al. 2013], composition-based design [Alférez et al. 2011], annotation-based implementation [Aversano et al. 2002; Kim et al. 2008; Kenner et al. 2010; Teixeira et al. 2011; Kästner et al. 2012a, 2012b; Liebig et al. 2013; Le et al. 2013; Chen et al. 2014], and annotation-based design [Czarnecki and Pietroszek 2006; Metzger et al. 2007; Heidenreich 2009]. For composition-based product lines, type checking ensures *safe composition* [Thaker et al. 2007; Kim et al. 2008]. Post and Sinz [2008] and Liebig et al. [2013] applied family-based type checking to parts of the Linux kernel.

Actually, there are two approaches to family-based type checking [Apel et al. 2010c; Huang et al. 2011]. *Local approaches* perform distinct reachability checks for every program element [Kim et al. 2008; Apel et al. 2010a; Kenner et al. 2010; Huang et al. 2011; Kästner et al. 2012a, 2012b; Liebig et al. 2013; Kolesnikov et al. 2013]. This results in many small satisfiability problems to solve, which can be cached efficiently [Kolesnikov et al. 2013]. *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; Teixeira et al. 2011; Alférez et al. 2011; Le et al. 2013]. This results in one large satisfiability problem to solve. Apel et al. [2010c] and Huang et al. [2011] discuss strengths and weaknesses of local and global approaches.

Type systems for product lines are often designed for explicitly typed languages, in which the expected types are given explicitly in the product line's implementation. However, when dealing with implicitly typed languages, instead of only checking whether a term is of a given type, we also need to infer types for given terms. A type system that performs type inference for an extension of the lambda calculus has been presented by Chen et al. [2014].

4.5. Family-Based Static Analysis

Recently, researchers have proposed family-based static analyses for software product lines, in particular, intraprocedural [Brabrand et al. 2013; Liebig et al. 2013; Midtgaard et al. 2014] and interprocedural [Bodden et al. 2013] data-flow analyses. Furthermore, static analyses have been proposed [Ribeiro et al. 2010; Tartler et al. 2011; Adelsberger et al. 2014; Sabouri and Khosravi 2014] that do not scale an existing static analysis known from single-system engineering, but rather focus on an analysis that is specific to product lines—which we refer to as *family-specific analyses*. Interestingly, most approaches for family-based static analysis are designed for annotation-based implementations and domain-independent specifications. As an exception, Adelsberger et al. [2014] focus on composition-based implementations with feature-oriented programming, and Sabouri and Khosravi [2014] propose a family-wide specification. Overall, these analyses support product lines implemented in C [Tartler et al. 2011; Liebig et al. 2013] and Java [Ribeiro et al. 2010; Brabrand et al. 2013; Bodden et al. 2013; Adelsberger et al. 2014]. For some of these approaches, existing tools have been extended such as Soot [Ribeiro et al. 2010; Brabrand et al. 2013; Bodden et al. 2013] and IFDS [Bodden et al. 2013].

Ribeiro et al. [2010] proposed the first family-based static analysis. Their goal was not product-line verification, but rather to support product-line development and prevent errors up-front. They show how to infer interfaces for preprocessor-based product lines using family-based data-flow analysis. Tartler et al. [2011] propose a family-based static analysis for defect detection in the Linux kernel. They analyze whether code blocks surrounded by #ifdef directives are dead (i.e., not contained in any product) or undead (i.e., contained in all products that contain the parent block).

Adelsberger et al. [2014] propose a static analysis for dynamic software product lines implemented with feature-oriented programming. The goal of their analysis is to assess the complexity of a reconfiguration at runtime. All these approaches are family-specific analyses.

In contrast, family-based analyses have also been proposed to scale existing static analyses from single-system engineering to product lines. Brabrand et al. [2013] demonstrate how to transform any standard intraprocedural data-flow analysis into a family-based data-flow analysis. They discuss three family-based approaches for this task, which differ in how they introduce variability into the underlying analysis abstractions (control-flow graph, the lattice storing the intermediate results, and the corresponding transfer function). Bodden et al. [2013] propose a similar data-flow analysis, which is, however, interprocedural and requires less intrusive changes to the internal analysis abstractions (i.e., only the control-flow graph is enriched with feature constraints). This way, one can reuse an information-flow analysis that was designed for regular programs also for software product lines, without having to change a single line of analysis code. Liebig et al. [2013] report experiences with scaling family-based data-flow analyses to real C product lines with thousands of features and millions of lines of code. Midtgaard et al. [2014] show how to systematically lift static analyses from single-system engineering to product lines. They propose variational abstract interpretation to develop family-based static analyses that are correct by construction. They exemplify their approach by means of constant propagation analysis. Sabouri and Khosravi [2014] propose a static analysis to identify features that are irrelevant for a given temporal property. These irrelevant features are then ignored during family-based model checking to reduce the state space.

Family-based static analyses show significant performance speed-ups compare to product-based static analyses. Brabrand et al. [2013] found that the family-based strategy is, on average, three times faster than the unoptimized product-based approach, without product generation and compilation, and almost eight times faster when including product generation and compilation. While their approach is intraprocedural, a comparison with an unoptimized product-based strategy is impractical for most other approaches. With a huge number of products, such an experiment would take years [Bodden et al. 2013; Liebig et al. 2013]. Nevertheless, Tartler et al. [2011] report that they found 1,776 defects in the Linux kernel in 15 minutes, which sum up to 5,129 lines of dead code and superfluous #ifdef statements. Liebig et al. [2013] compare the family-based strategy with several optimized product-based strategies, such as a single configuration containing as many features as possible (i.e., *allyesconfig*), configuration coverage [Tartler et al. 2012], and pair-wise sampling. They found that the family-based strategy was slower than checking the single configuration but faster than all other sampling strategies.

A further criterion to distinguish family-based analyses is when they consider the variability model. A family-based analysis may use the dependencies of the variability model already during the analysis—which we refer to as *early variability-model consideration*. In contrast, a family-based analysis may incorporate the knowledge about valid feature combinations only at the end of the analysis to rule out false positives—which we refer to as *late variability-model consideration*. Most family-based static analyses rely on early variability-model consideration [Ribeiro et al. 2010; Brabrand et al. 2013; Bodden et al. 2013; Tartler et al. 2011; Adelsberger et al. 2014], whereas family-based static analyses with late variability-model consideration have been proposed recently [Bodden et al. 2013; Liebig et al. 2013]. Interestingly, Bodden et al. [2013] measured that late variability-model consideration was slightly faster than early consideration (i.e., ignoring the variability model during analysis is faster). However, it is not clear whether this applies to static analyses in general.

4.6. Family-Based Model Checking

Several approaches have been proposed for family-based model checking. The overall idea is that a model of the product-line implementation is analyzed with respect to the variability model and one or more properties. For a given property, the model checker analyzes whether the property is fulfilled by all products. If not, the model checker usually returns a propositional formula specifying those products that violate the property [Gruler et al. 2008].

One distinguishing characteristic of approaches for family-based model checking is whether they operate directly on source code or on an abstraction of a system. The former is known as software model checking and the latter is referred to as abstract model checking. The majority of approaches for family-based model checking apply abstract model checking. Abstract models have been defined using I/O automata [Lauenroth et al. 2010], labeled transition systems [Gruler et al. 2008; Sabouri and Khosravi 2012; ter Beek et al. 2013; Sabouri and Khosravi 2014], modal transition systems [Fischbein et al. 2006; Asirelli et al. 2012], featured transition systems [Classen et al. 2010, 2013, 2014; Cordy et al. 2012a, 2012b, 2013a, 2013b; Sabouri and Khosravi 2013a], featured timed automata [Cordy et al. 2012c], modal sequence diagrams [Greenyer et al. 2013], and actor models [Sabouri and Khosravi 2013b]. In contrast, several authors proposed approaches for family-based software model checking. These approaches analyze product lines written in C [Post and Sinz 2008; Apel et al. 2011, 2013c] and Java [Schaefer et al. 2010; Kästner et al. 2012c; Apel et al. 2013c]. Family-based model checking has been applied to composition-based [Apel et al. 2011, 2013c; Greenyer et al. 2013; Classen et al. 2013, 2014; Sabouri and Khosravi 2013a] and annotation-based [Fischbein et al. 2006; Gruler et al. 2008; Post and Sinz 2008; Lauenroth et al. 2010; Classen et al. 2010, 2013; Schaefer et al. 2010; Asirelli et al. 2012; Cordy et al. 2012a, 2012b, 2012c, 2013a, 2013b; Sabouri and Khosravi 2012, 2013b, 2014; ter Beek et al. 2013] product lines. Tool support for family-based model checking is often built on existing tools such as CBMC [Post and Sinz 2008], ProMoVer [Schaefer et al. 2010], CPAchecker [Apel et al. 2011, 2013c], NuSMV [Greenyer et al. 2013; Classen et al. 2013, 2014], SPIN [Sabouri and Khosravi 2012; Classen et al. 2013; Cordy et al. 2013b], UPPAAL [Cordy et al. 2012c], JPF [Apel et al. 2013c], Afra [Sabouri and Khosravi 2013a, 2014], Maude [ter Beek et al. 2013], and Modere [Sabouri and Khosravi 2013b].

Besides the product line's source code or an abstraction thereof, family-based model checking requires a formalism to encode properties (i.e., specifications) to be checked. Most approaches are based on computation tree logic (CTL) [Lauenroth et al. 2010; Greenyer et al. 2013; Classen et al. 2013, 2014; Cordy et al. 2013a] or linear temporal logic (LTL) [Classen et al. 2010; Schaefer et al. 2010; Cordy et al. 2012b; ter Beek et al. 2013; Sabouri and Khosravi 2013b, 2014]. Gruler et al. [2008] and Sabouri and Khosravi [2012] use the μ-calculus as a generalization of CTL and LTL. Cordy et al. [2012c] rely on timed CTL, an extension of CTL with support for modeling real-time properties. Asirelli et al. [2012] propose the branching-time temporal logic MHML to express common dependencies of variability models in the product-line specification. Apel et al. [2011] and Apel et al. [2013c] model temporal safety properties using aspect-oriented programming and assertions. In contrast to all other approaches, Cordy et al. [2012a] propose to model the product-line implementation *and* properties each as featured transition systems, and they verify the properties by checking whether both featured transition systems are in a simulation relation. These specification techniques have been lifted to product lines using different strategies. The surveyed approaches use domain-independent [Post and Sinz 2008; Sabouri and Khosravi 2013a], family-wide [Fischbein et al. 2006; Gruler et al. 2008; Schaefer et al. 2010; Sabouri and Khosravi

2012, 2013b, 2014; Cordy et al. 2012c; Greenyer et al. 2013; ter Beek et al. 2013], feature-based [Lauenroth et al. 2010; Classen et al. 2010; Apel et al. 2011, 2013c], and family-based specifications [Asirelli et al. 2012; Cordy et al. 2012a, 2012b; Classen et al. 2013; Cordy et al. 2013a; Cordy et al. 2013b; Classen et al. 2014].

Most approaches for family-based model checking rely on early variability-model consideration. That is, the variability model is used during analysis to ignore paths for invalid feature combinations. In contrast, Classen et al. [2013] discuss family-based model checking with late variability-model consideration. The variability model is ignored during model checking, and the output of model checking is then combined with the variability model to prevent false positives. False positives can occur if a property is violated only by invalid configurations. By means of an empirical evaluation, Classen et al. [2013] found that family-based model checking with early variability-model consideration is about 7% faster than with late consideration. This is in contrast to results for static analyses, where some experiments revealed the opposite (see Section 4.5). It is up to future work to find the fundamental reasons for this difference between family-based model checking and static analysis.

4.7. Family-Based Theorem Proving

When we started working on this survey, there was no approach applying the family-based strategy to theorem proving. Based on this insight, some of the authors (Thüm, Schaefer, and Apel) proposed family-based theorem proving for product lines implemented with feature-oriented programming [Thüm et al. 2012]. Similar to approaches for family-based model checking, all feature modules are translated into a single metaproduct that can be passed to the off-the-shelf verification tool KeY. In addition to the translation of feature modules, feature-based specifications given in an extension of the Java Modeling Language are translated into a metaspecification (i.e., a family-based specification). Instead of checking that each product fulfills its specification, it is sufficient to check that the metaproduct conforms to the metaspecification, which saves 85% of the calculation time for automatic verification for the product line of bank accounts.

5. FEATURE-BASED ANALYSES

Software product lines may also be analyzed using a *feature-based* strategy. That is, all domain artifacts implementing a certain feature are analyzed in isolation (in bundles assigned to individual features) without considering other features or the 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. Similar to family-based strategies, feature-based strategies operate on domain artifacts instead of 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* with respect to the features (i.e., the analysis results cannot be invalidated by interactions of features).

Definition 5.1 (*Feature-Based Analysis*). An analysis of a software product line is *feature based* if (a) it operates only on domain artifacts and (b) software artifacts belonging to a feature are analyzed in isolation (i.e., knowledge about valid feature combinations is not used) and feature interactions are not considered.

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. 6. 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 analyzed with feature-based type checking.

5.1. Example

In the object-store example, we can analyze each of the three feature modules in isolation to *some extent*. First, we can parse each feature module in isolation to make sure that it conforms to the syntax and to create an abstract syntax tree for each feature module. For syntax checking, it is sufficient to consider each feature module in isolation, as syntactic correctness is independent of other features, and thus a compositional property. Second, the type checker uses the abstract syntax tree to infer which types and references can be resolved by a feature itself and which have to be provided by other features. As an example, all references to field sealed are internal and can be checked within the implementation of feature *AccessControl*, as illustrated in Figure 6. That is, there is no need to check this reference for every product. However, some of the references cut across feature boundaries and cannot be checked in a feature-based fashion. Well-typedness is not a compositional property. For example, references to the methods read and readAll of feature *AccessControl* cannot be resolved within the feature.

5.2. Advantages and Disadvantages

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

Nevertheless, feature-based strategies have advantages compared to product-based and family-based strategies. First, they analyze domain artifacts (similar to

family-based strategies) instead of operating on generated software artifacts, and thus there are no redundant computations across products.

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

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

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

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

5.3. Feature-Based Approaches

As indicated previously, there are only a few strict feature-based approaches. For example, 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) are compositional analyses. While parsing is a necessary task for any static analysis, it is only discussed for nonmodular 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. For most software analyses, we need to combine feature-based analyses with other strategies.

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

6. COMBINED ANALYSIS STRATEGIES

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

6.1. Feature-Product-Based Analyses

A commonly proposed combined strategy, which we identified in the literature, is the feature-product-based strategy that consists of two phases. First, features are analyzed in isolation, and second, all properties not checked feature based are analyzed for each

product. The feature-based part can only analyze features locally and the product-based part checks that features work properly in combination. The key idea is to reduce analysis effort by checking as much as possible feature locally.

Definition 6.1 (*Feature-Product-Based Analysis*). An analysis of a software product line is *feature product based* if (a) it consists of a feature-based analysis followed by a product-based analysis, and (b) the analysis results of the feature-based analysis are used in the product-based analysis.

Example. In our object store, we could start to type-check all features in isolation. As shown in Figure 6, we can check that all intrafeature references are valid and create an interface for every feature. The interface contains all methods, fields, and classes that the feature provides and also those that are required. In the second step, we take these interfaces and iterate over every valid combination of features and check whether the interfaces are compatible for every valid configuration (i.e., everything that is required in some interface is provided by another interface). This way, we can save redundant checks for intrafeature references. Especially, if some features evolve, we can omit reanalyzing unchanged features in the feature-based analysis step.

Advantages and Disadvantages. Feature-product-based strategies reduce redundant computations, compared to strict product-based strategies, but redundancies still occur for all analyses applied at the product level. For example, when some features evolve, other features need not to be reanalyzed, but all products containing any of the affected features need to be analyzed again whenever the feature interface changes. Considering that strict feature-based strategies are usually not sufficient for noncompositional 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. Feature-product-based type checking has been proposed for composition-based implementation approaches such as logic metaprogramming [Klose and Ostermann 2010], feature modules [Apel and Hutchins 2010; Kolesnikov et al. 2013], delta modules [Bettini et al. 2013], and traits [Bettini et al. 2014]. As explained for our running example, feature implementations (i.e., modules, feature modules, delta modules, traits) are type checked as far as possible in isolation in the first phase. Additionally, interfaces or constraints are generated for each feature implementation, which are used in an unoptimized product-based linking step. For each approach, a type system for a core calculus has been presented; Featherweight Record-Trait Java [Bettini et al. 2014] and Imperative Featherweight Delta Java [Bettini et al. 2013] formalize product lines in Java, whereas gDeep [Apel and Hutchins 2010] has been discussed for being used in the context of different languages such as Java, Haskell, Bali, and XML. Kolesnikov et al. [2013] evaluated this strategy against family-based type checking and found that feature-product-based type checking was significantly slower.

Feature-Product-Based Model Checking. In feature-product-based model checking, each feature implementation is model checked in isolation and an interface is generated specifying provided and assumed behavior of other features. Then, these interfaces are checked for every product to make sure that features are compatible with each other. Compared to the family-based strategy, the feature-product-based strategy was only proposed a few times, even if most approaches for feature-product-based model checking are older than first approaches for family-based model checking. In addition, the field of feature-product-based model checking is less diverse: all surveyed approaches

[Fisler and Krishnamurthi 2001; Nelson et al. 2001; Li et al. 2002, 2005; Blundell et al. 2004; Liu et al. 2011] (a) apply abstract model checking (i.e., we have not found a single approach for feature-product-based software model checking), (b) are based on finite state machines, (c) consider only product lines that are decomposed into modules (i.e., no annotation-based product lines), and (d) verify properties defined in CTL. However, some approaches rely on family-wide specifications [Fisler and Krishnamurthi 2001; Nelson et al. 2001; Blundell et al. 2004; Liu et al. 2011] and others on feature-based specifications [Li et al. 2002, 2005]. Fisler and Krishnamurthi [2001] extended the existing tool VIS, and others built new tools from scratch [Li et al. 2005; Liu et al. 2011]. We found only one empirical evaluation of feature-product-based model checking: Liu et al. [2011] measured that this strategy is 6.7 times faster than unoptimized product-based model checking for a product line with four products.

Feature-Product-Based Theorem Proving. Approaches classified as feature-product-based theorem proving are diverse. They have been used for product-line verification [Thüm et al. 2011b; Damiani et al. 2012], but also to prove type soundness for product lines of programming languages [Batory and Börger 2008; Delaware et al. 2011, 2013].

Thüm et al. [2011b] propose feature-product-based theorem proving for verification of feature modules. Features are implemented in feature modules based on Java and specified using the Java Modeling Language (JML). The verification step 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. The verification time mainly depends on writing proof scripts in the feature-based part. For a product line with 12 products, the number of handwritten proof commands was reduced by 88% compared to an unoptimized product-based strategy.

Damiani et al. [2012] propose a similar approach for feature-product-based theorem proving. They introduce a calculus representing a kind of feature module. A syntax is presented to define method contracts with uninterpreted assertions to refer to contracts of other methods. In their approach, deductive verification is achieved in two steps. First, contracts of each implementation unit are verified as far as possible locally, by only considering the uninterpreted assertions and guarantees of other methods. Second, all remaining proof obligations are proved for each generated product. This approach has not yet been evaluated empirically.

Besides the verification of product-line implementations, the strategy has also been applied in the general context of theorem proving for product lines. 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. Nevertheless, a human still needs to check that every product has a valid grammar, correctness theorems, and natural language proof.

Similarly, Delaware et al. [2011] and Delaware et al. [2013] propose feature-product-based theorem proving for a product line of type-soundness proofs. They focus on a product line of languages based on Featherweight Java, for which language features, such as generics, interfaces, or casting, can be selected independently. All eight Featherweight Java variants are proved 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. Delaware et al. [2011] measured the time the proof assistant Coq needed to verify theorems; proof checking for all features took about 4 minutes, whereas checking *all* products based on these proofs for each feature took only about 1 minute (i.e., Coq spent most of the time on compositional properties).

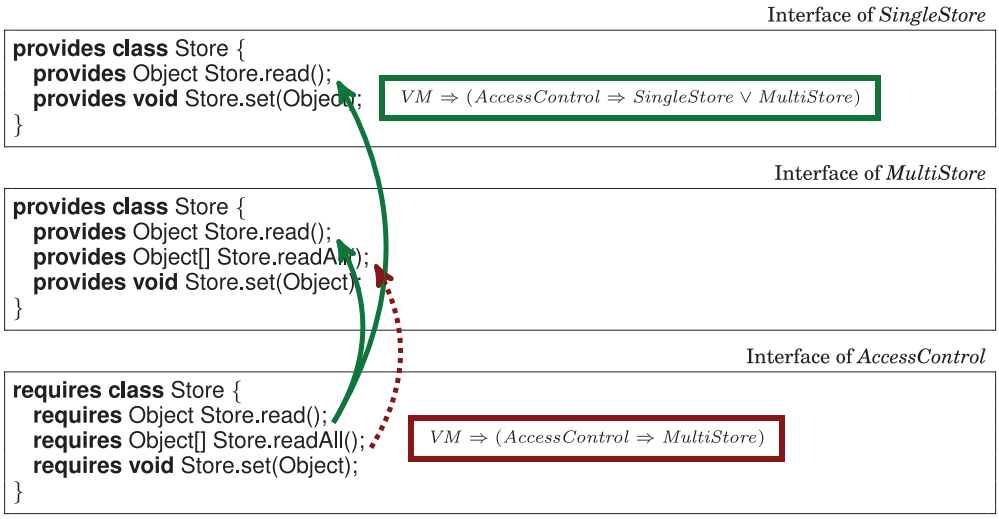


Fig. 7. Feature-family-based type checking analyzes features in isolation and applies family-based type checking on the deduced feature interfaces afterward. The references to read and readAll cut across feature boundaries and are checked at composition time based on the features' interfaces and the variability model.

6.2. Feature-Family-Based Analyses

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

Definition 6.2 (*Feature-Family-Based Analysis*). An analysis of a software product line is *feature-family based* if (a) it consists of a feature-based analysis followed by a family-based analysis and (b) the analysis effort of the feature-based analysis is used in the family-based analysis.

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 7). For example, feature *AccessControl* requires a method read, which is provided either by feature *SingleStore* or feature *MultiStore*. However, method readAll required by feature *AccessControl* is not available in all products with feature *AccessControl*. Basically, we can create a propositional formula for each reference, which can be checked using a satisfiability solver, as described in Section 4.

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 eliminated not only for feature-local analyses but also for analyses across features). Furthermore, compared to a pure family-based analysis, it better supports the evolution of software product lines, in which usually only a small set of features evolves. Finally, a feature-family-based analysis combines open-world and closed-world scenarios. That is, while the feature-based analysis does

not require knowing all features and their valid combinations, we can postpone all parts of the analysis requiring a closed world to the family-based analysis.

Feature-Family-Based Type Checking. The feature-family-based strategy has been proposed for type checking of composition-based product lines implemented with feature-oriented programming [Delaware et al. 2009] and delta-oriented programming [Damiani and Schaefer 2012]. Both approaches rely on a constraint-based type system that generates constraints for each module in isolation. The constraints describe type references and dependencies that must be fulfilled by other modules. Delaware et al. [2009] use these constraints to create a propositional formula describing the set of well-typed feature combinations, which is then compared to the variability model to retrieve whether all valid feature combinations according to the variability model are well typed (i.e., in a global approach, cp. Section 4.4). In contrast, Damiani and Schaefer [2012] construct a product family generation tree (PFGT) representing all possible generation orders of products. The constraints of the single deltas are then checked by traversing the PFGT in a single pass constituting a family-based analysis step. There are no empirical comparisons of this strategy to other strategies. However, Delaware et al. [2009] report that their approach was even faster than generating and compiling a single product.

Feature-Family-Based Theorem Proving. Hähnle and Schaefer [2012] 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. Called methods not defined in the delta itself are approximated by the specification of the first introduction of this method, either in the core product or in the first delta in the application ordering. Still, we consider this step as feature based, because only specifications of other features are incorporated and there is no implementation artifacts of other features. In the second step, all deltas are checked for conformance in a family-based fashion.

6.3. Family-Product-Based Analyses

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

Definition 6.3 (*Family-Product-Based Analysis*). An analysis of a software product line is *family-product based* if (a) it consists of a (partial) family-based analysis followed by a product-based analysis and (b) the analysis effort of the family-based analysis is reused in the product-based analysis.

Family-Product-Based Analyses. We have not found pure static approaches for this strategy. However, we discuss some approaches that combine static and dynamic analyses of product lines, because similar approaches could also be created that operate only statically.

Tartler et al. [2012] propose a heuristic for sampling that incorporates the variability model *and* the preprocessor-based code base to achieve a special code coverage. This is in contrast to approaches for sampling discussed in Section 3.4, which incorporate only the variability model. The idea is that an analysis touches each domain artifact and individual piece of code at least once. Hence, for each given preprocessor directive, they ensure that it is activated in at least one product in the resulting set of sample products. In the second step, an arbitrary software analysis can be reused in a product-based fashion. Their approach implies rather weak guarantees toward correctness and mainly targets bug finding.

Kim et al. [2010] propose a family-product-based analysis for feature-oriented programming. They apply a family-based static analysis to reduce the set of products for which safety properties need to be monitored during runtime. Safety properties are defined in AspectJ. In the first step, the family-based static analysis rules out configurations that cannot violate the safety property. The result of the static analysis is a specialized variability model representing the products that are monitored in the second step. Kim et al. [2011] generalized this work from safety properties in AspectJ to general test cases. For each test case, a set of products is calculated that is sufficient to test. They extend control-flow and data-flow analyses with variability information to trace the effect of features.

Similarly, Shi et al. [2012] propose a family-product-based analysis to analyze feature interactions. A family-based static analysis is used to calculate test cases, which are then used to test products individually. They create a dependency graph for the whole software product line while considering only valid feature combinations as specified in the variability model. Then, they use symbolic execution to compute method summaries and test cases. The number of test cases can be influenced using a coverage criterion as known from t-wise testing [Perrouin et al. 2010]. Finally, resulting test cases are executed for products individually.

6.4. 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 6.4 (*Feature-Family-Product-Based Analysis*). An analysis of a software product line is *feature-family-product based* if (a) it consists of a feature-based analysis followed by a family-product-based analysis, and (b) the analysis 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.

7. RESEARCH AGENDA

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

—What are the strengths and weaknesses of the individual strategies in practice?
—Is it meaningful to combine each strategy with each software analysis, and which combinations are useful and superior in what circumstances?
—What can we learn from strategies applied to one analysis when applying them to other analyses?

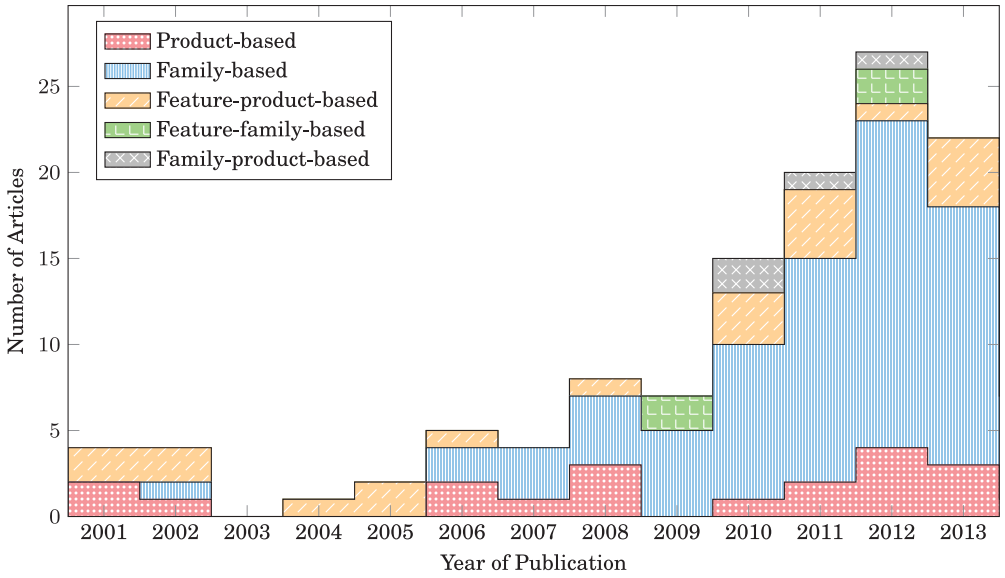


Fig. 8. Overview of the frequency of analysis strategies addressed in the research literature (before 2014). We found no approaches pursuing a feature-based or feature-family-product-based strategy.

—Are there further novel analysis strategies?
—What characteristics of a given product line affect the efficiency of the individual analysis strategies?
—Is there a principle and possibly automated way to lift a given specification and analysis technique to product lines?

Based on the classification of existing approaches in the previous sections, we discuss underrepresented research areas and specific research questions that we uncovered in our survey: In Section 7.1, we summarize advantages and drawbacks of each strategy and identify underrepresented analysis strategies. We discuss how strategies have been evaluated quantitatively and report weaknesses of existing evaluations in Section 7.2. In Section 7.3, we discuss which analysis strategies have been combined with which specification and implementation strategies. Finally, we describe future challenges for type checking, static analysis, model checking, and theorem proving of software product lines in Section 7.4–7.5.

7.1. Comparison of Analysis Strategies

In the previous sections, we have discussed three basic strategies and four combined strategies to scale software analysis to product lines. In Figure 8, we give an overview of how often each strategy was applied in the surveyed approaches and when. More than half of the approaches apply a family-based strategy, suggesting that this strategy to cope with software variability is well known. However, we also found approaches for analysis in a product-line context that do not discuss how to cope with many, similar products. Almost a third of all approaches rely on the generation of *all* products (i.e., unoptimized product-based and feature-product-based strategy), which is infeasible for large product lines. None of the surveyed analysis approaches is solely feature based, because analyzing features only in isolation is usually not sufficient (i.e., the properties of interest are not compositional). All combined strategies except for the feature-product-based strategy are underrepresented.

Table I. Summary of Advantages and Disadvantages of Analysis Strategies. A Perfect Strategy would have a "Yes" in Every Column, but There is a Tradeoff between Avoiding Redundancies in Computations and the Size of Analysis Problems

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

[1]Analysis results of product-based analysis step refers to products.

[2]Avoids redundant computations when changed domain artifacts can be verified feature-based.

[3]The family-based analysis problem may be larger or smaller than verifying a single product depending on how much feature-based part reduces the analysis problem.

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

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

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

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

7.2. Quantitative Evaluation of Analysis Strategies

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

First, we found that in almost all studies, a particular strategy is compared to an unoptimized product-based analysis [Classen et al. 2010, 2013, 2014; Schaefer et al. 2010; Cordy et al. 2012a, 2012c; Sabouri and Khosravi 2012, 2013a, 2013b; Thüm et al. 2012; Greenyer et al. 2013; Apel et al. 2013c; Chen et al. 2014] or to the analysis of a single product [Post and Sinz 2008; Delaware et al. 2009; Kenner et al. 2010; Kästner et al. 2011, 2012a; Gazzillo and Grimm 2012]. The advantage of such a standard evaluation is that we can compare different approaches more easily, even if evaluations strongly depend on the size and kind of product line being analyzed. However, the unoptimized product-based strategy is often not an option in practice (e.g., for large product lines). Recently, researchers started comparing family-based with optimized product-based [Apel et al. 2013c; Liebig et al. 2013] and feature-product-based strategies [Kolesnikov et al. 2013]. However, there are still strategies that have not been compared with any other strategy. For example, researchers proposed feature-family-based analyses [Delaware et al. 2009; Hähnle and Schaefer 2012; Damiani and Schaefer 2012], but there is no empirical comparison with a family-based or feature-product-based strategy that assesses the potential of such a strategy.

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

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

Table II. Number of Surveyed Approaches for Each Combination of Analysis Strategy and Specification Strategy as well as Analysis Strategy and Implementation Strategy

	Implementation			**Specification**				
Analysis strategy	Composition based	Annotation based		Domain-independent	Family-wide	Product based	Feature based	Family based
Product based (unoptimized)	7	3		2	3	0	5	0
Product based (optimized)	6	2		3	2	0	4	0
Family based	16	37		28	12	0	5	7
Feature based	0	0		0	0	0	0	0
Feature-product based	17	0		8	5	0	6	0
Feature-family based	3	0		2	0	0	1	0
Family-product based	2	2		1	2	0	0	0
Feature-family-product based	0	0		0	0	0	0	0
Total*	49	43		42	23	0	20	7

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

of time [Liebig et al. 2013]. In summary, for empirical evaluations, the performance of each step should be documented to improve comparability.

7.3. Product-Line Implementation and Specification

In addition to the analysis strategy, we classified product-line analyses with respect to the underlying implementation and specification strategy. We distinguish between composition-based and annotation-based implementations, and between domain-independent, family-wide, product-based, feature-based, and family-based specifications (see Section 2). In Table II, we give an overview of which analysis strategies have been applied to which kind of implementation and specification strategy, respectively.

The majority of implementation and specification strategies discussed in our survey have actually been applied. Both composition-based and annotation-based implementations have been used with similar frequency in the literature. In contrast, most approaches built on domain-independent specifications. This is natural, as many approaches consider type checking or static analysis, for which specifications are often defined independently of a particular system. In addition, many other specifications are family-wide, which means that, while the implementation contains variability, the specification does not. About the same number of approaches rely on feature-based specifications, which support variability similar to composition-based implementation. However, we found only seven approaches using family-based specification [Asirelli et al. 2012; Cordy et al. 2012a, 2012b, 2013a, 2013b; Classen et al. 2013, 2014], and none with product-based specification. An open research question is how much variability is required in product-line specifications (e.g., whether feature-based specifications are sufficient [Apel et al. 2013b]) and whether there are differences in the variability of specifications depending on the underlying software analysis. Model checking is the only software analysis to which all specification strategies (except product-based specification) have already been applied (see Section 4.6). For type checking, domain-independent specifications are sufficient, but other specification strategies shall be explored for static analysis and theorem proving.

Table III. Classification of Product-Line Type Checking

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

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

7.4. Product-Line Type Checking

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

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

Table IV. Classification of Product-Line Static Analysis

	Composition based	Annotation based	Domain-independent	Family-wide	Feature based
Product based (unoptimized)	Klaeren et al. [2001], Scholz et al. [2011]				Klaeren et al. [2001], Scholz et al. [2011]
Product based (optimized)	Katz [2006]	Liebig et al. [2013]	Liebig et al. [2013]	Liebig et al. [2013]	Katz [2006]
Family based	Adelsberger et al. [2014]	Ribeiro et al. [2010], Bodden et al. [2013], Brabrand et al. [2013], Liebig et al. [2013], Midtgaard et al. [2014], Sabouri and Khosravi [2014]	Ribeiro et al. [2010], Liebig et al. [2013], Midtgaard et al. [2014]	Bodden et al. [2013], Brabrand et al. [2013], Liebig et al. [2013], Sabouri and Khosravi [2014]	
Family-product based	Kim et al. [2010], Kim et al. [2011]	Kim et al. [2011], Shi et al. [2012]	Shi et al. [2012]	Kim et al. [2010], Kim et al. [2011]	

Furthermore, there are two competing approaches for family-based type checking, namely, local and global approaches (see Section 4.4). The main difference is whether the whole product line is encoded as a single or a large number of satisfiability problems. However, empirical evaluations are missing that compare time and space efficiency of both approaches.

7.5. Product-Line Static Analysis

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

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

Family-based static analyses have been compared empirically with optimized [Liebig et al. 2013] and unoptimized [Brabrand et al. 2013; Bodden et al. 2013] product-based analyses. Comparisons include time efficiency [Brabrand et al. 2013; Bodden et al. 2013; Liebig et al. 2013], memory efficiency [Brabrand et al. 2013], and soundness [Bodden et al. 2013]. In particular, Bodden et al. [2013] measured that it is faster to ignore than to incorporate the variability model during static analysis. Further studies will evaluate whether this is the case for all kinds of static analysis and explore the fundamental reasons. This is especially interesting, as opposite experience has been noted with model checking (see Section 4.6).

7.6. Product-Line Model Checking

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

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

7.7. Product-Line Theorem Proving

In Table VI, we summarize the strategies used for theorem proving. Compared to type checking and model checking, there are fewer approaches for theorem proving, suggesting that this research field is underrepresented. In particular, it is surprising that there is only one family-based approach for theorem proving [Thüm et al. 2012], whereas this strategy has been applied often to type checking and model checking.

For theorem proving, there is a lack of reliable evaluations comparing the strategies to each other. Optimized product-based and feature-family-based theorem proving have not been compared so far. Thüm et al. [2012] compare the time efficiency of family-based strategy with that of unoptimized product-based theorem proving. Feature-product-based theorem proving has been evaluated against an unoptimized product-based strategy, in terms of the size of handwritten proof scripts [Thüm et al. 2011b]. Delaware et al. [2011] measured the time needed for the feature-based and the product-based part in feature-product-based theorem proving. However, there is not a single evaluation of memory consumption, and many strategies have not yet been compared to each other.

8. RELATED WORK

Classifications for Quality Assurance in Software Product Lines. Pohl et al. [2005] discuss four strategies for product-line testing. In contrast to our classification, they discuss strategies incorporating tests at different levels including unit tests, integration tests, and system tests. The brute force strategy is similar to unoptimized product-based analysis, but tests are performed at all levels for all products. In contrast, for the pure application strategy, only delivered products are tested in application engineering. The sample application strategy is equivalent to the sample-based strategy in our classification. Finally, with the commonality and reuse strategy, artifacts common to all products are tested in domain engineering and then all products are tested separately. These strategies have been defined for product-line testing and do not represent all strategies that we identified in our survey.

Similarly, Metzger [2007] and Lauenroth et al. [2010] discuss three strategies for quality assurance (e.g., model checking) of product lines, namely, commonality strategy, sample strategy (similar to sample-based analysis), and comprehensive strategy (similar to unoptimized product-based analysis). The idea of the commonality strategy is to check artifacts that are common to all products. Similar to the family-based strategy, the commonality strategy uses the variability model and domain artifacts to retrieve the common artifacts. Similar to the feature-based strategy, it can only uncover

Table V. Classification of Product-Line Model Checking

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

Table VI. Classification of Product-Line Theorem Proving

	Composition based	Domain-independent	Feature based
Product based (unoptimized)	Harhurin and Hartmann [2008]		Harhurin and Hartmann [2008]
Product based (optimized)	Bruns et al. [2011]		Bruns et al. [2011]
Family based	Thüm et al. [2012]		Thüm et al. [2012]
Feature-product based	Batory and Börger [2008], Delaware et al. [2011], Thüm et al. [2011b], Damiani et al. [2012], Delaware et al. [2013]	Delaware et al. [2011], Delaware et al. [2013]	Batory and Börger [2008], Thüm et al. [2011b], Damiani et al. [2012]
Feature-family based	Hähnle and Schaefer [2012]		Hähnle and Schaefer [2012]

certain faults for a given product line. The commonality strategy is not represented in our classification. However, we have not found any approaches applying this strategy.

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

In previous work, we proposed first ideas on a classification into product-based and feature-based verification techniques [Thüm et al. 2011b]. In this survey, we extend the classification to family-based and combined strategies, generalize it from verification to software analyses in general, and actually classify existing approaches. Furthermore, we give definitions and examples and discuss advantages and disadvantages of each strategy in detail. In contrast to our early ideas, we strengthened the notion of a feature-based analysis to make researchers and practitioners aware that most analyses do not solely operate on features in isolation and that combinations with product-based or family-based analyses are usually necessary. It is worthwhile to note that von Rhein et al. [2013] already use our classification to model combinations of product-line analyses, but they do not survey the literature on product-line analyses.

Surveys on Quality Assurance in Software Product Lines. Benavides et al. [2010] survey automated analyses for variability models. These analyses consider only the variability model and can detect anomalies such as dead features or compute statistics such as the number of products. In contrast, our focus is on approaches that operate on source code or models thereof. However, many of the approaches in our survey rely on techniques from this line of research to reason about variability (e.g., for the family-based strategy).

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

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

9. CONCLUSION

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

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

Overall, we classified 123 existing analysis and specification approaches, gaining insights into the field of product-line analyses. First, whereas many approaches claim to be compositional, we distinguish feature-product-based and feature-family-based strategies to reveal how inherently noncompositional properties such as feature interactions are analyzed. Second, not all strategies have been applied to all software analyses. For example, we have not found feature-product-based static analyses, feature-family-based static analyses, and feature-family-based model checking. Third, we identified well-represented (e.g., family-based type checking, static analysis, and model checking) and underrepresented research areas (e.g., optimized product-based analyses, family-based theorem proving, and feature-family-based theorem proving). Finally, there is no compositional analysis for annotation-based product lines or family-based specifications. Based on these insights, we formulated research questions to be addressed in future work. Most notably, is there a principle and possibly automated way to lift a given specification and analysis technique to product lines for a particular analysis strategy?

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

ACKNOWLEDGMENTS

We thank Martin Kuhlemann for interesting discussions and help with the classification of product-line analyses. Furthermore, we gratefully acknowledge Eric Bodden, Márcio Ribeiro, Vander Alves, Stefania Gnesi, Maurice H. ter Beek, and Martin Erwig for their input regarding our classification. Finally, we thank Alexander von Rhein, Seyed Hossein Haeri, David Broneske, Reimar Schröter, Wolfram Fenske, and Martin Schäler for helpful comments on prior drafts of this article.

REFERENCES

S. Adelsberger, S. Sobernig, and G. Neumann. 2014. Towards assessing the complexity of object migration in dynamic, feature-oriented software product lines. In *Proc. Int'l Workshop Variability Modelling of Software-intensive Systems (VaMoS'14)*. ACM, New York, NY, 17:1–17:8.

M. Alférez, R. E. Lopez-Herrejon, A. Moreira, V. Amaral, and A. Egyed. 2011. Supporting consistency checking between features and software product line use scenarios. In *Proc. Int'l Conf. Software Reuse (ICSR'11)*. Springer, Berlin, 20–35.

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

S. Apel and D. Hutchins. 2010. A calculus for uniform feature composition. *ACM Transactions on Programming Languages and Systems* 32, 5, 19:1–19:33.

S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. 2010a. Type safety for feature-oriented product lines. *Automated Software Engineering* 17, 3, 251–300.

S. Apel, C. Kästner, and C. Lengauer. 2008a. Feature featherweight Java: A calculus for feature-oriented programming and stepwise refinement. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE'08)*. ACM, New York, NY, 101–112.

S. Apel, T. Leich, and G. Saake. 2008b. Aspectual feature modules. *IEEE Transaction on Software Engineering* 34, 2, 162–180.

S. Apel, W. Scholz, C. Lengauer, and C. Kästner. 2010b. Detecting dependences and interactions in feature-oriented design. In *Proc. Int'l Symposium Software Reliability Engineering (ISSRE'10)*. IEEE, Washington, DC, 161–170.

S. Apel, W. Scholz, C. Lengauer, and C. Kästner. 2010c. Language-independent reference checking in software product lines. In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD'10)*. ACM, New York, NY, 65–71.

S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. 2011. Detection of feature interactions using feature-aware verification. In *Proc. Int'l Conf. Automated Software Engineering (ASE'11)*. IEEE, Washington, DC, 372–375.

S. Apel, A. von Rhein, T. Thüm, and C. Kästner. 2013b. Feature-interaction detection based on feature-based specifications. *Computer Networks* 57, 12, 2399–2409.

S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. 2013c. Strategies for product-line verification: Case studies and experiments. In *Proc. Int'l Conf. Software Engineering (ICSE'13)*. IEEE, Piscataway, NJ, 482–491.

P. Asirelli, M. H. ter Beek, A. Fantechi, and S. Gnesi. 2012. A compositional framework to derive product line behavioural descriptions. In *Proc. Int'l Symposium Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'12)*. Springer, Berlin, 146–161.

C. Atkinson and T. Kühne. 2003. Model-Driven development: A metamodeling foundation. *IEEE Software* 20, 5, 36–41.

L. Aversano, M. D. Penta, and I. D. Baxter. 2002. Handling preprocessor-conditioned declarations. In *Proc. Int'l Workshop Source Code Analysis and Manipulation (SCAM'02)*. IEEE, Washington, DC, 83–92.

D. Batory. 2005. Feature models, grammars, and propositional formulas. In *Proc. Int'l Software Product Line Conf. (SPLC'05)*. Springer, Berlin, 7–20.

D. Batory and E. Börger. 2008. Modularizing theorems for software product lines: The jbook case study. *Journal of Universal Computer Science* 14, 12, 2059–2082.

D. Batory, J. N. Sarvela, and A. Rauschmayer. 2004. Scaling step-wise refinement. *IEEE Transactions on Software Engineering* 30, 6, 355–371.

D. Benavides, S. Segura, and A. Ruiz-Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Information Systems* 35, 6, 615–708.

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

S. Bessling and M. Huhn. 2014. Towards formal safety analysis in feature-oriented product line development. In *Proc. Int'l Symposium Foundations of Health Information Engineering and Systems (FHIES'14)*. Springer, Berlin, 217–235.

L. Bettini, F. Damiani, and I. Schaefer. 2013. Compositional type checking of delta-oriented software product lines. *Acta Informatica* 50, 2, 77–122.

L. Bettini, F. Damiani, and I. Schaefer. 2014. Implementing type-safe software product lines using parametric traits. *Science of Computer Programming* To appear.

D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. 2007. The software model checker blast: Applications to software engineering. *International Journal of Software Tools for Technology Transfer* 9, 5, 505–525.

D. Beyer and M. E. Keremoglu. 2011. CPAchecker: A tool for configurable software verification. In *Proc. Int'l Conf. Computer Aided Verification (CAV'11)*. Springer, Berlin, 184–190.

C. Blundell, K. Fisler, S. Krishnamurthi, and P. V. Hentenryck. 2004. Parameterized interfaces for open system verification of product lines. In *Proc. Int'l Conf. Automated Software Engineering (ASE'04)*. IEEE, Washington, DC, 258–267.

E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini. 2013. SPLLIFT: Statically analyzing software product lines in minutes instead of years. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI'13)*. ACM, New York, NY, 355–364.

T. F. Bowen, F. S. Dworack, C.-H. Chow, N. Griffeth, G. E. Herman, and Y.-J. Lin. 1989. The feature interaction problem in telecommunications systems. In *Proc. Int'l Conf. Software Engineering for Telecommunication Switching Systems (SETSS'89)*. IEEE, Washington, DC, 59–62.

C. Brabrand, M. Ribeiro, T. Tolêdo, J. Winther, and P. Borba. 2013. Intraprocedural dataflow analysis for software product lines. *Transactions on Aspect-Oriented Software Development* 10, 73–108.

D. Bruns, V. Klebanov, and I. Schaefer. 2011. Verification of software product lines with delta-oriented slicing. In *Proc. Int'l Conf. Formal Verification of Object-Oriented Software (FoVeOOS'11)*. Springer, Berlin, 61–75.

T. Buchmann and F. Schwägerl. 2012. Ensuring well-formedness of configured domain models in model-driven product lines based on negative variability. In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD'12)*. ACM, New York, NY, 37–44.

M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. 2003. Feature interaction: A critical review and considered forecast. *Computer Networks* 41, 1, 115–141.

I. D. Carmo Machado, J. D. McGregor, Y. A. C. Cavalcanti, and E. S. De Almeida. 2014. On strategies for testing software product lines: A systematic literature review. *Journal of Information and Software Technology* To appear.

S. Chen, M. Erwig, and E. Walkingshaw. 2014. Extending type inference to variational programs. *ACM Transactions on Programming Languages and Systems* 36, 1, 1:1–1:54.

A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. 1999. NuSMV: A new symbolic model verifier. In *Proc. Int'l Conf. Computer Aided Verification (CAV'99)*. Springer, London, 495–499.

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

A. Classen, M. Cordy, P. Heymans, A. Legay, and P.-Y. Schobbens. 2014. Formal semantics, modular specification, and symbolic verification of product-line behaviour. *Science of Computer Programming* 80, Part B, 416–439.

A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin. 2013. Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. *IEEE Transactions on Software Engineering* 39, 8, 1069–1089.

A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. 2010. Model checking lots of systems: Efficient verification of temporal properties in software product lines. In *Proc. Int'l Conf. Software Engineering (ICSE'10)*. ACM, New York, NY, 335–344.

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

M. Cordy, A. Classen, P. Heymans, A. Legay, and P.-Y. Schobbens. 2013a. Model checking adaptive software with featured transition systems. In *Proc. Workshop Assurances for Self-Adaptive Systems (ASAS'13)*. Springer, Berlin, 1–29.

M. Cordy, A. Classen, G. Perrouin, P.-Y. Schobbens, P. Heymans, and A. Legay. 2012a. Simulation-based abstractions for software product-line model checking. In *Proc. Int'l Conf. Software Engineering (ICSE'12)*. IEEE, Piscataway, NJ, U672–682.

M. Cordy, A. Classen, P.-Y. Schobbens, P. Heymans, and A. Legay. 2012b. Managing evolution in software product lines: A model-checking perspective. In *Proc. Int'l Workshop Variability Modelling of Software-intensive Systems (VaMoS'12)*. ACM, New York, NY, 183–191.

M. Cordy, P.-Y. Schobbens, P. Heymans, and A. Legay. 2012c. Behavioural modelling and verification of real-time software product lines. In *Proc. Int'l Software Product Line Conf. (SPLC'12)*. ACM, New York, NY, 66–75.

M. Cordy, P.-Y. Schobbens, P. Heymans, and A. Legay. 2012d. Towards an incremental automata-based approach for software product-line model checking. In *Proc. Int'l Workshop Formal Methods and Analysis in Software Product Line Engineering (FMSPLE'12)*. ACM, New York, NY, 74–81.

M. Cordy, P.-Y. Schobbens, P. Heymans, and A. Legay. 2013b. Beyond boolean product-line model checking: Dealing with feature attributes and multi-features. In *Proc. Int'l Conf. Software Engineering (ICSE'13)*. IEEE, Piscataway, NJ, 472–481.

P. Cousot and R. Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. Symposium Principles of Programming Languages (POPL'77)*. ACM, New York, NY, 238–252.

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

K. Czarnecki and K. Pietroszek. 2006. Verifying feature-based model templates against well-formedness OCL constraints. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE'06)*. ACM, New York, NY, 211–220.

P. A. Da Mota Silveira Neto, I. D. Carmo Machado, J. D. McGregor, E. S. De Almeida, and S. R. De Lemos Meira. 2011. A systematic mapping study of software product lines testing. *Journal of Information and Software Technology* 53, 5, 407–423.

F. Damiani, O. Owe, J. Dovland, I. Schaefer, E. B. Johnsen, and I. C. Yu. 2012. A transformational proof system for delta-oriented programming. In *Proc. Int'l Workshop Formal Methods and Analysis in Software Product Line Engineering (FMSPLE'12)*. ACM, New York, NY, 53–60.

F. Damiani and I. Schaefer. 2012. Family-based analysis of type safety for delta-oriented software product lines. In *Proc. Int'l Symposium Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'12)*. Springer, Berlin, 193–207.

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

B. Delaware, W. Cook, and D. Batory. 2009. Fitting the pieces together: A machine-checked model of safe composition. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE'09)*. ACM, New York, NY, 243–252.

B. Delaware, W. Cook, and D. Batory. 2011. Product lines of theorems. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'1)*. ACM, New York, NY, 595–608.

B. Delaware, B. C. d. S. Oliveira, and T. Schrijvers. 2013. Meta-theory à la Carte. In *Proc. Symposium Principles of Programming Languages (POPL'13)*. ACM, New York, NY, 207–218.

D. Detlefs, G. Nelson, and J. B. Saxe. 2005. Simplify: A theorem prover for program checking. *Journal of the ACM* 52, 3, 365–473.

E. Engström and P. Runeson. 2011. Software product line testing - A systematic mapping study. *J. of Information and Software Technology* 53, 2–13.

L. Etxeberria, G. Sagardui, and L. Belategi. 2008. Quality aware software product line engineering. *Journal of the Brazilian Computer Society* 14, 1, 57–69.

A. Fantechi and S. Gnesi. 2008. Formal modeling for product families engineering. In *Proc. Int'l Software Product Line Conf. (SPLC'08)*. IEEE, Washington, DC, 193–202.

D. Fischbein, S. Uchitel, and V. Braberman. 2006. A foundation for behavioural conformance in software product line architectures. In *Proc. Int'l Workshop Role of Software Architecture for Testing and Analysis (ROSATEA'06)*. ACM, New York, NY, 39–48.

K. Fisler and S. Krishnamurthi. 2001. Modular verification of collaboration-based software designs. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE'01)*. ACM, New York, NY, 152–163.

K. Fisler and S. Krishnamurthi. 2005. Decomposing verification around end-user features. In *Proc. IFIP Working Conf. Verified Software: Theories, Tools, Experiments (VSTTE'05)*. Springer, Berlin, 74–81.

P. Gazzillo and R. Grimm. 2012. SuperC: Parsing all of C by taming the preprocessor. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI'12)*. ACM, New York, NY, 323–334.

J. Greenyer, A. M. Sharifloo, M. Cordy, and P. Heymans. 2013. Features meet scenarios: Modeling and consistency-checking scenario-based product line specifications. *Requirements Engineering* 18, 2, 175–198.

A. Gruler, M. Leucker, and K. Scheidemann. 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, 113–131.

R. Hähnle and I. Schaefer. 2012. A liskov principle for delta-oriented programming. In *Proc. Int'l Symposium Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'12)*. Springer, Berlin, 32–46.

A. Harhurin and J. Hartmann. 2008. Towards consistent specifications of product families. In *Proc. Int'l Symposium Formal Methods (FM'08)*. Springer, Berlin, 390–405.

F. Heidenreich. 2009. Towards systematic ensuring well-formedness of software product lines. In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD'09)*. ACM, New York, NY, 69–74.

G. J. Holzmann. 1997. The model checker SPIN. *IEEE Transactions on Software Engineering* 23, 5, 279–295.

D. Hovemeyer and W. Pugh. 2004. Finding bugs is easy. *SIGPLAN Notices* 39, 12, 92–106.

S. S. Huang, D. Zook, and Y. Smaragdakis. 2007. cJ: Enhancing java with safe type conditions. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD'07)*. ACM, New York, NY, 185–198.

S. S. Huang, D. Zook, and Y. Smaragdakis. 2011. Statically safe program generation with SafeGen. *Science of Computer Programming* 76, 5, 376–391.

P. Istoan. 2013. Methodology for the derivation of product behaviour in a software product line. Ph.D. thesis, Université Rennes 1, Luxembourg.

M. Janota, J. Kiniry, and G. Botterweck. 2008. Formal methods in software product lines: Concepts, survey, and guidelines. Tech. Rep. Lero-TR-SPL-2008-02, Lero, University of Limerick. May.

P. Jayaraman, J. Whittle, A. M. Elkhodary, and H. Gomaa. 2007. Model composition in product lines and feature interaction detection using critical pair analysis. In *Proc. Int'l Conf. Model Driven Engineering Languages and Systems (MODELS'07)*. Springer, Berlin, 151–165.

E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. 2012. ABS: A core language for abstract behavioral specification. In *Proc. Int'l Symposium Formal Methods for Components and Objects (FMCO'12)*. Springer, Berlin, 142–164.

K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. 1990. Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute.

C. Kästner, S. Apel, T. Thüm, and G. Saake. 2012a. Type checking annotation-based product lines. *Transactions on Software Engineering and Methodology* 21, 3, 14:1–14:39.

C. Kästner, S. Apel, S. S. ur Rahman, M. Rosenmüller, D. Batory, and G. Saake. 2009. On the impact of the optional feature problem: Analysis and case studies. In *Proc. Int'l Software Product Line Conf. (SPLC'09)*. Software Engineering Institute, Pittsburgh, PA, 181–190.

C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. 2011. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'11)*. ACM, New York, NY, 805–824.

C. Kästner, K. Ostermann, and S. Erdweg. 2012b. A variability-aware module system. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'12)*. ACM, New York, NY, 773–792.

C. Kästner, A. von Rhein, S. Erdweg, J. Pusch, S. Apel, T. Rendel, and K. Ostermann. 2012c. Toward variability-aware testing. In *Proc. Int'l Workshop Feature-Oriented Software Development*. ACM, New York, NY, 1–8.

S. Katz. 2006. Aspect categories and classes of temporal properties. *Transactions on Aspect-Oriented Software Development* 1, 106–134.

A. Kenner, C. Kästner, S. Haase, and T. Leich. 2010. TypeChef: Toward type checking #Ifdef variability in C. In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD'10)*. ACM, New York, NY, 25–32.

G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. 1997. Aspect-oriented programming. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP'97)*. Springer, Berlin, 220–242.

C. H. P. Kim, D. Batory, and S. Khurshid. 2011. Reducing combinatorics in testing product lines. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD'11)*. ACM, New York, NY, 57–68.

C. H. P. Kim, E. Bodden, D. Batory, and S. Khurshid. 2010. Reducing configurations to monitor in a software product line. In *Proc. Int'l Conf. Runtime Verification (RV'10)*. Springer, Berlin, 285–299.

C. H. P. Kim, C. Kästner, and D. Batory. 2008. On the modularity of feature interactions. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE'08)*. ACM, New York, NY, 23–34.

C. H. P. Kim, S. Khurshid, and D. Batory. 2012. Shared execution for efficiently testing product lines. In *Proc. Int'l Symposium Software Reliability Engineering (ISSRE'12)*. IEEE, Washington, DC, 221–230.

C. H. P. Kim, D. Marinov, S. Khurshid, D. Batory, S. Souto, P. Barros, and M. D'Amorim. 2013. SPLat: Lightweight dynamic analysis for reducing combinatorics in testing configurable systems. In *Proc. Europ. Software Engineering Conf. / Foundations of Software Engineering (ESEC / FSE'13)*. ACM, New York, NY, 257–267.

T. Kishi and N. Noda. 2006. Formal verification and software product lines. *Comm. ACM* 49, 73–77.

H. Klaeren, E. Pulvermueller, A. Rashid, and A. Speck. 2001. Aspect composition applying the design by contract principle. In *Proc. Int'l Symposium Generative and Component-Based Software Engineering (GCSE'01)*. Springer, Berlin, 57–69.

K. Klose and K. Ostermann. 2010. Modular logic metaprogramming. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'10)*. ACM, New York, NY, 484–503.

S. Kolesnikov, A. von Rhein, C. Hunsen, and S. Apel. 2013. A comparison of product-based, feature-based, and family-based type checking. In *Proc. Int'l Conf. Generative Programming: Concepts & Experiences (GPCE'13)*. ACM, New York, NY, 115–124.

M. Kuhlemann, D. Batory, and C. Kästner. 2009. Safe composition of non-monotonic features. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE'09)*. ACM, New York, NY, 177–186.

M. Kuhlemann and M. Sturm. 2010. Patching product line programs. In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD'10)*. ACM, New York, NY, 33–40.

D. R. Kuhn, R. N. Kacker, and Y. Lei. 2013. *Introduction to Combinatorial Testing,* 1st ed. Chapman & Hall/CRC, London, UK.

K. Lauenroth, A. Metzger, and K. Pohl. 2010. Quality assurance in the presence of variability. In *Intentional Perspectives on Information Systems Engineering*. Springer, Berlin, 319–333.

D. M. Le, H. Lee, K. C. Kang, and L. Keun. 2013. Validating consistency between a feature model and its implementation. In *Proc. Int'l Conf. Software Reuse (ICSR'13)*. Springer, Berlin, 1–16.

J. Lee, S. Kang, and D. Lee. 2012. A survey on software product line testing. In *Proc. Int'l Software Product Line Conf. (SPLC'12)*. ACM, New York, NY, 31–40.

H. Li, S. Krishnamurthi, and K. Fisler. 2002. Verifying cross-cutting features as open systems. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE'02)*. ACM, New York, NY, 89–98.

H. Li, S. Krishnamurthi, and K. Fisler. 2005. Modular verification of open features using three-valued model checking. *Automated Software Engineering* 12, 3, 349–382.

J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. 2010. An analysis of the variability in forty preprocessor-based software product lines. In *Proc. Int'l Conf. Software Engineering (ICSE'10)*. IEEE, Washington, DC, 105–114.

J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. 2013. Scalable analysis of variable software. In *Proc. Europ. Software Engineering Conf. / Foundations of Software Engineering (ESEC / FSE'13)*. ACM, New York, NY, 81–91.

J. Liu, S. Basu, and R. Lutz. 2011. Compositional model checking of software product lines using variation point obligations. *Automated Software Engineering* 18, 1, 39–76.

J. Liu, D. Batory, and C. Lengauer. 2006. Feature oriented refactoring of legacy applications. In *Proc. Int'l Conf. Software Engineering (ICSE'06)*. ACM, New York, NY, 112–121.

J. Liu, J. Dehlinger, and R. Lutz. 2007. Safety analysis of software product lines using state-based modeling. *Journal of Systems and Software* 80, 11, 1879–1892.

R. Lutz. 2007. Survey of product-line verification and validation techniques. Tech. Rep. 2014/41221, NASA, Jet Propulsion Laboratory, La Canada Flintridge, CA.

F. Medeiros, M. Ribeiro, and R. Gheyi. 2013. Investigating preprocessor-based syntax errors. In *Proc. Int'l Conf. Generative Programming: Concepts & Experiences (GPCE'13)*. ACM, New York, NY, 75–84.

M. Mendonça, A. Wasowski, and K. Czarnecki. 2009. SAT-based analysis of feature models is easy. In *Proc. Int'l Software Product Line Conf. (SPLC'09)*. Software Engineering Institute, Pittsburgh, PA, 231–240.

A. Metzger. 2007. Quality issues in software product lines: Feature interactions and beyond. In *Proc. Int'l Conf. Feature Interactions in Software and Communication Systems (ICFI'07)*. IOS Press, Amsterdam, The Netherlands, 1–12.

A. Metzger, K. Pohl, P. Heymans, P.-Y. Schobbens, and G. Saval. 2007. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *Proc. Int'l Conf. Requirements Engineering (RE'07)*. IEEE, Washington, DC, 243–253.

J. Midtgaard, C. Brabrand, and A. Wasowski. 2014. Systematic derivation of static analyses for software product lines. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD'14)*. ACM, New York, NY, 181–192.

S. Montagud and S. Abrahão. 2009. Gathering current knowledge about quality evaluation in software product lines. In *Proc. Int'l Software Product Line Conf. (SPLC'09)*. Software Engineering Institute, Pittsburgh, PA, 91–100.

S. S. Muchnick. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA.

T. Nelson, D. D. Cowan, and P. S. C. Alencar. 2001. Supporting formal verification of crosscutting concerns. In *Proc. Int'l Conf. Metalevel Architectures and Separation of Crosscutting Concerns*. Springer, London, 153–169.

H. V. Nguyen, C. Kästner, and T. N. Nguyen. 2014. Exploring variability-aware execution for testing plugin-based web applications. In *Proc. Int'l Conf. Software Engineering (ICSE'14)*. ACM, New York, NY.

C. Nie and H. Leung. 2011. A survey of combinatorial testing. *ACM Computing Surveys* 43, 2, 11:1–11:29.

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

T. Nipkow, M. Wenzel, and L. C. Paulson. 2002. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, Berlin.

S. Oster, F. Markert, and P. Ritter. 2010. Automated incremental pairwise testing of software product lines. In *Proc. Int'l Software Product Line Conf. (SPLC'10)*. Springer, Berlin, 196–210.

S. Oster, A. Wübbeke, G. Engels, and A. Schürr. 2011. A survey of model-based software product lines testing. In *Model-based Testing for Embedded System*. CRC Press, Boca Raton, FL, 339–381.

S. Owre, J. M. Rushby, and N. Shankar. 1992. PVS: A prototype verification system. In *Proc. Int'l Conf. Automated Deduction (CADE'92)*. Springer, London, 748–752.

D. L. Parnas. 1976. On the design and development of program families. *IEEE Transactions on Software Engineering* SE-2, 1, 1–9.

G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. Le Traon. 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'10)*. IEEE, Washington, DC, 459–468.

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

M. Plath and M. Ryan. 2001. Feature integration using a feature construct. *Science of Computer Programming* 41, 1, 53–84.

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

H. Post and C. Sinz. 2008. Configuration lifting: Software verification meets software configuration. In *Proc. Int'l Conf. Automated Software Engineering (ASE'08)*. IEEE, Washington, DC, 347–350.

C. Prehofer. 1997. Feature-oriented programming: A fresh look at objects. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP'97)*. Springer, Berlin, 419–443.

M. Ribeiro, H. Pacheco, L. Teixeira, and P. Borba. 2010. Emergent feature modularization. In *Proc. Int'l Conf. Object-Oriented Programming Systems Languages and Applications Companion (SPLASH'10)*. ACM, New York, NY, 11–18.

V. V. Rubanov and E. A. Shatokhin. 2011. Runtime verification of Linux kernel modules based on call interception. In *Proc. Int'l Conf. Software Testing, Verification and Validation (ICST'11)*. IEEE, Washington, DC, 180–189.

H. Sabouri and R. Khosravi. 2012. Efficient verification of evolving software product lines. In *Proc. Int'l Conf. Fundamentals of Software Engineering (FSEN'12)*. Springer, Berlin, 351–358.

H. Sabouri and R. Khosravi. 2013a. Delta modeling and model checking of product families. In *Proc. Int'l Conf. Fundamentals of Software Engineering (FSEN'13)*. Springer, Berlin, 51–65.

H. Sabouri and R. Khosravi. 2013b. Modeling and verification of reconfigurable actor families. *Journal of Universal Computer Science* 19, 2, 207–232.

H. Sabouri and R. Khosravi. 2014. Reducing the verification cost of evolving product families using static analysis techniques. *Science of Computer Programming* 83, 0, 35–55.

I. Schaefer, D. Gurov, and S. Soleimanifard. 2010. Compositional algorithmic verification of software product lines. In *Proc. Int'l Symposium Formal Methods for Components and Objects (FMCO'10)*. Springer, Berlin, 184–203.

W. Scholz, T. Thüm, S. Apel, and C. Lengauer. 2011. Automatic detection of feature interactions using the java modeling language: An experience report. In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD'11)*. ACM, New York, NY, 7:1–7:8.

J. Schumann. 2001. *Automated Theorem Proving in Software Engineering*. Springer, Berlin.

J. Shi, M. B. Cohen, and M. B. Dwyer. 2012. Integration testing of software product lines using compositional symbolic execution. In *Proc. Int'l Conf. Fundamental Approaches to Software Engineering (FASE'12)*. Springer, Berlin, 270–284.

M. Svahnberg, J. van Gurp, and J. Bosch. 2005. A taxonomy of variability realization techniques: Research articles. *Software: Practice and Experience* 35, 8, 705–754.

R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero. 2012. Configuration coverage in the analysis of large-scale system software. *ACM SIGOPS Operating Systems Review* 45, 3, 10–14.

R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. 2011. Feature consistency in compile-time-configurable system software: Facing the Linux 10,000 feature problem. In *Proc. Europ. Conf. Computer Systems (EuroSys'11)*. ACM, New York, NY, 47–60.

L. Teixeira, P. Borba, and R. Gheyi. 2011. Safe composition of configuration knowledge-based software product lines. In *Proc. Brazilian Symposium Software Engineering (SBES'11)*. IEEE, Washington, DC, 263–272.

M. H. ter Beek, A. L. Lafuente, and M. Petrocchi. 2013. Combining declarative and procedural views in the specification and analysis of product families. In *Proc. Int'l Workshop Formal Methods and Analysis in Software Product Line Engineering (FMSPLE'13)*. ACM, New York, NY, 10–17.

A. Tevanlinna, J. Taina, and R. Kauppinen. 2004. Product family testing: A survey. *SIGSOFT Software Engineering Notes* 29, 12–17.

S. Thaker, D. Batory, D. Kitchin, and W. Cook. 2007. Safe composition of product lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE'07)*. ACM, New York, NY, 95–104.

T. Thüm, D. Batory, and C. Kästner. 2009. Reasoning about edits to feature models. In *Proc. Int'l Conf. Software Engineering (ICSE'09)*. IEEE, Washington, DC, 254–264.

T. Thüm, C. Kästner, S. Erdweg, and N. Siegmund. 2011a. Abstract features in feature modeling. In *Proc. Int'l Software Product Line Conf. (SPLC'11)*. IEEE, Washington, DC, 191–200.

T. Thüm, I. Schaefer, S. Apel, and M. Hentschel. 2012. Family-based deductive verification of software product lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE'12)*. ACM, New York, NY, 11–20.

T. Thüm, I. Schaefer, M. Kuhlemann, and S. Apel. 2011b. Proof composition for deductive verification of software product lines. In *Proc. Int'l Workshop Variability-intensive Systems Testing, Validation and Verification (VAST'11)*. IEEE, Washington, DC, 270–277.

T. Thüm, I. Schaefer, M. Kuhlemann, S. Apel, and G. Saake. 2012. Applying design by contract to feature-oriented programming. In *Proc. Int'l Conf. Fundamental Approaches to Software Engineering (FASE'02)*. Springer, Berlin, 255–269.

N. Ubayashi and T. Tamai. 2002. Aspect-oriented programming with model checking. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD'02)*. ACM, New York, NY, 148–154.

F. J. van der Linden, K. Schmid, and E. Rommes. 2007. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, Berlin.

W. Visser, K. Havelund, G. P. Brat, and S. Park. 2000. Model checking programs. In *Proc. Int'l Conf. Automated Software Engineering (ASE'00)*. Springer, Berlin, 3–12.

A. von Rhein, S. Apel, C. Kästner, T. Thüm, and I. Schaefer. 2013. The PLA model: On the combination of product-line analyses. In *Proc. Int'l Workshop Variability Modelling of Software-intensive Systems (VaMoS'13)*. ACM, New York, NY, 14:1–14:8.

C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. 2009. SPASS version 3.5. In *Proc. Int'l Conf. Automated Deduction (CADE'09)*. Springer, Berlin, 140–145.

M. Weiser. 1981. Program slicing. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, Piscataway, NJ, USA, 439–449.

D. M. Weiss. 2008. The product line hall of fame. In *Proc. Int'l Software Product Line Conf. (SPLC)*. IEEE, Washington, DC, USA, 395.

Received November 2012; revised September 2013; accepted January 2014