



# The PLA Model: On the Combination of Product-Line Analyses

Alexander von Rhein  
University of Passau  
Germany

Sven Apel  
University of Passau  
Germany

Christian Kästner  
Carnegie Mellon University  
USA

Thomas Thüm  
University of Magdeburg  
Germany

Ina Schaefer  
University of Braunschweig  
Germany

## ABSTRACT

Product-line analysis has received considerable attention in the last decade. As it is often infeasible to analyze each product of a product line individually, researchers have developed analyses, called *variability-aware analyses*, that consider and exploit variability manifested in a code base. Variability-aware analyses are often significantly more efficient than traditional analyses, but each of them has certain weaknesses regarding applicability or scalability. We present the Product-Line-Analysis model, a formal model for the classification and comparison of existing analyses, including traditional and variability-aware analyses, and lay a foundation for formulating and exploring further, combined analyses. As a proof of concept, we discuss different examples of analyses in the light of our model, and demonstrate its benefits for systematic comparison and exploration of product-line analyses.

**Categories and Subject Descriptors:** D.2.4 [Software]: Software Engineering—*Software/Program Verification*; D.2.13 [Software]: Software Engineering—*Reusable Software*

**General Terms:** Measurement, theory, verification

**Keywords:** Software product lines, product-line analysis, PLA model

## 1. INTRODUCTION

Software product-line engineering is an approach to develop a set of similar software products based on a common code base for a particular domain. Differences between products can be specified and communicated by means of *features*, which are end-user visible units of behavior or other product characteristics that are of interest to stakeholders [9]. A user can automatically derive different products by

selecting a set of desired features. The set is called a *configuration*. Each configuration defines a different product with specific functional characteristics.

To gather (or maintain) knowledge about the characteristics of the products, it is imperative to use software analyses, which range from simple statistics (e.g., complexity metrics) to complex analyses such as type checking, model checking, deductive verification, and testing [17]. A naive approach to product-line analysis is to create and analyze all possible products individually. Even for a relatively small product line with 39 features (SQLite, [16]) and over 3 million products, this is clearly infeasible. Even if we could analyze all products, the analyses would involve redundant computations caused by the similarities among products.

In an ongoing endeavor to make product-line analyses feasible, researchers have developed approaches that consider variability and exploit similarities among products during analysis [17]. These *variability-aware* analyses have been shown to be faster than comparable analyses that do not consider variability [3, 4, 7, 8, 11, 18].

We and others observed that contemporary variability-aware analyses rely on a set of recurring patterns or strategies which we describe in detail in Section 2. While these strategies are the key to their success, we found that each has disadvantages that limit scalability and efficiency of the analysis [17]. For example, one strategy is to build a product simulator that can simulate the behavior of all individual products of a product line, and to analyze the simulator in a single pass (called a *family-based* strategy). While this approach is much faster than analyzing all or even only some products (e.g., [4, 7, 8]), we also made the experience that the analysis of a simulator induces a considerably higher memory consumption than the analysis of a single product [4] (because the simulator can get very large). Another strategy is to analyze features and their implementations as far as possible in isolation (called a *feature-based* strategy), which saves analysis effort. Feature-based analyses require additional analysis effort to detect interactions between features [14, 17]. Depending on the implementation approach and the analysis used, this approach may be infeasible, especially, when features crosscut the entire code base at a fine grain.

Based on a discussion of individual strengths and weaknesses of the strategies used in existing product-line analyses, we propose the *Product-Line-Analysis* model, a novel,

formal model for comparison of product-line analyses. It is based on an existing classification [17], and it includes traditional and variability-aware analyses. Our preliminary investigation suggests that, based on individual strengths and weaknesses, it is beneficial to combine existing strategies in different ways, depending on the considered analysis problem.

The contributions of our paper are:

- The Product-Line-Analysis model (PLA model), a formal model for the classification and comparison of product-line analyses
- A notation for the visualization of product-line analyses based on the PLA model
- An application of the model on three product-line analyses and a summary of our initial experiences with describing the analyses in the context of our model
- A discussion of perspectives that arise from the PLA model for the field of product-line analysis

Our model is based on a recent survey on product-line analysis [17]. It improves over previous work by formalizing and discussing combinations of existing and future product-line analyses, and considering new combinations of analyses. This paper is meant to provide a guideline and inspiration for ongoing and further developments in the field of product-line analysis.

## 2. PRODUCT-LINE ANALYSES

In this section, we discuss software product-line analyses and introduce concepts that are used in the paper. As a running example, we use a very simplistic printing-device product line. In our example, features are implemented by means of FEATUREHOUSE-style feature modules and composed using superimposition [2]. But our classification abstracts from feature representation and composition, and is even compatible with variability implemented using preprocessor directives and conditional compilation. We discuss one such application (TypeChef) in Section 4.

### 2.1 Terminology and Running Example

The printing-device product line consists of four features that implement different printing and scanning functionalities (Figure 1). Feature *BasicPrinter* supports printing a single page as well as manual duplex printing (i.e., the user has to turn the page). This feature is the basis for each product of the product line. The optional features *Duplex* and *Scan* implement additional functionality for automatic duplex printing and scanning. Optional features do not depend on the presence of other features. The fourth feature *Copy* depends on feature *Scan*. It can only be included in configurations that contain feature *Scan*. Typically, these constraints are defined by a *feature model* [10], which describes relationships and constraints between features. The feature model of the printer product line has two constraints: first, the feature *BasicPrinter* is mandatory, it must be enabled in all products, and, second, feature *Copy* can only be selected if feature *Scan* is present.

The configurations that satisfy the feature model are called *valid* configurations. Their features can be composed to form products. In our example, {*BasicPrinter*} and {*BasicPrinter*, *Duplex*} are valid configurations, but {*Scan*, *Duplex*} is not valid because it does not contain *BasicPrinter*. A set of features is called *partial* configuration if it can be extended to a valid configuration by adding features. In this

Feature *BasicPrinter*

```

1 class Printer {
2   // basic printing method
3   public void print(Page p) {
4     ...
5   }
6   // manual duplex printing
7   public void printDuplex(Page front, Page back) {
8     print(front);
9     ... // ask user to turn and re-insert page
10    print(back);
11  }
12 }

```

Feature *Duplex*

```

1 class Printer {
2   // automatic duplex printing
3   public void printDuplex(Page front, Page back) {
4     ...
5   }
6 }

```

Feature *Scan*

```

1 class Printer {
2   // scanning of one page
3   public Page scan() {
4     ...
5   }
6 }

```

Feature *Copy*

```

1 class Printer {
2   // scans one page and prints it
3   public void copy() {
4     print(scan());
5   }
6 }

```

Figure 1: A feature-oriented implementation of a small printing device driver (the class declarations of *Duplex*, *Scan*, and *Copy* refine the corresponding declaration of *BasicPrinter* like mixins)

simple notion of configurations, a configuration is a set of enabled features. Disabled features are not stated explicitly. Figure 2 shows a product composed based on a valid configuration. The product contains the features *BasicPrinter*, *Duplex*, and *Scan*. Method `printDuplex(Page,Page)` from feature *BasicPrinter* is not included because it has been overwritten by the method from feature *Duplex* with the same signature.

### 2.2 Product Line Analysis Strategies

A product-line analysis takes as input the domain implementation (feature modules in our example) and the feature model of a product line, and attempts to make a statement about some property of the product line. Such properties include static code properties (software metrics), some form of correctness (well-typedness or functional correctness), or even non-functional properties (e.g., average throughput).

A simple approach to run analyses on product lines is to select a subset of all valid configurations, generate all corresponding products and analyze them. Normally, the generated products are expressed in a general purpose programming language, so one can use off-the-shelf analysis tools. The generation of the products (e.g., the product in Figure 2), is done with standard product-line generator tools that are also used for the actual deployment. We call

---

Product {*BasicPrinter*, *Scan*, *Copy*}

```

1 class Printer {
2   // basic printing method
3   public void print(Page p) {
4     ...
5   }
6   public void printDuplex(Page front, Page back) {
7     print(front);
8     ... // ask user to turn and re-insert page
9     print(back);
10  }
11  // scanning of one page
12  public Page scan() {
13    ...
14  }
15  // scans one page and prints it
16  public void copy() {
17    print(scan());
18  }
19 }

```

---

Figure 2: A product of the printing-device product line including the features *BasicPrinter*, *Scan*, and *Copy*.

this approach *sample-based*, because it is based on a sample set selected from the set of all possible products. However, the sample-based approach has two drawbacks: First, if not all valid configurations have been considered, the analysis only checks a subset of products and cannot make a general statement about the whole product line (except for statistical statements). Second, it generates many similar products and the analysis has to analyze similar code or behavior repeatedly. In a type analysis of our example, the basic printing method of feature *BasicPrinter* would be type checked in every product, six times if all products are checked.

*Variability-aware* analyses aim at reducing analysis effort by exploiting similarities among products in the sense that shared code or behavior is not re-checked multiple times. Instead, variability-aware analyses run directly on the feature implementations or pre-process the features and their implementations to produce a representation suitable for efficient analysis. Variability-aware analyses consist of various steps that either combine features (*integration step*), process features or feature combinations to produce a result (*processing step*). An overall analysis that consists of multiple integration and processing steps is called henceforth *compound analysis*.

A simple compound analysis is product-based testing. The first step is an integration step where different products are built from the features' source code. In the next step (processing step), all products are tested separately and for each product one verdict is produced. In the final step, again an integration step, all verdicts are combined to determine which features or feature interactions cause exceptional behavior (c.f., [16] for an example of this step).

A more sophisticated compound analysis is *family-based* analysis [17], which takes features and variability information and creates a product simulator, which includes code of all features and that can simulate the behaviors of all valid products (integration step). Normally, the feature selection is *fixed* after product generation at compile time (features cannot be deactivated any more). In a product simulator, some features can still be activated or deactivated, even at runtime (the simulator contains runtime variability). Therefore, the simulator retains the selection of enabled features until analysis time and can simulate a whole set of valid

---

Product simulator {*BasicPrinter*, *Duplex*}

```

1 class Printer {
2   static boolean _FeatureDuplex_enabled;
3   // basic printing method
4   public void print(Page p) {
5     ...
6   }
7   public void printDuplex(Page front, Page back) {
8     if (_FeatureDuplex_enabled) {
9       //code from Duplex feature
10    } else {
11      print(front);
12      ... // ask user to turn and re-insert page
13      print(back);
14    }
15  }
16 }

```

---

Figure 3: A product simulator of the printing-device product line covering the features *BasicPrinter* and *Duplex*

products instead of only one. Figure 3 shows a simulator of the printer product-line, where the *Duplex* feature is variable. In this case, the variability is implemented with a conditional statement that selects the code depending on a boolean variable (Line 9). As this product simulator can switch between the behaviors of all encoded valid products, a variability-aware analysis can reason about all products by analyzing the simulator once (processing step). This saves time compared to the separate analysis of all individual products [3,4,6–8,11,18] because shared parts have to be analyzed only once.

### 3. CLASSIFICATION OF ANALYSES

Our model of product-line analyses is motivated by the observation that variability-aware analyses can be expressed as combinations of integration and processing steps. The model can be used to classify analyses along three dimensions, which represent different combination patterns underlying different variability-aware analyses (Section 3.1).

Most of these analyses are rather complex and difficult to describe without a suitable vocabulary. One benefit of our formal model is that it helps to explain these complex analyses. Another advantage is that we can describe the individual steps of the analysis in isolation and discuss tradeoffs of the different approaches, which will help in the development of novel, efficient analyses. The formal model describes combinations of integration and combination steps algebraically (Section 3.2).

#### 3.1 Dimensions of Classification

In previous work, we have identified three dimensions of the classification of product-line analyses [17], which form the basis of our model. The dimensions give rise to a cube, which we call Product-Line-Analysis cube (PLA cube), as illustrated in Figure 4.

**Sampling.** The first dimension is “sampling” (e.g., edge **A–B**); it represents the fraction of products of a product line that are subject to analysis. For a very small product line, it is feasible to analyze all products; for large product lines, developers have to resort to sampling (i.e., selecting a proper subset of products based on a sampling heuristics, e.g., pairwise feature coverage or code coverage) [5, 15, 16]. For the printer product line, we might decide that only products

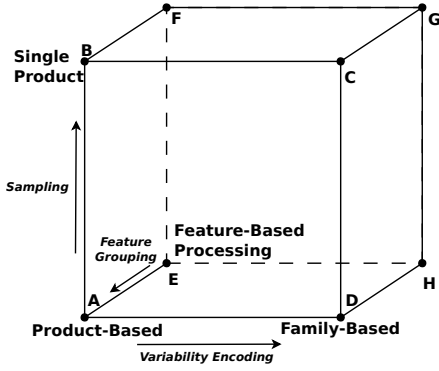


Figure 4: The PLA cube – visualization of the space of possible combinations of product-line analyses

with feature *Duplex* have to be tested, as other products are currently not requested by customers. So, only 3 of 6 products must be analyzed, saving roughly half of the analysis time (compared to analyzing all products). However, sampling means that not all valid products are considered, which may influence the conclusions one can draw from the analysis results (e.g., we may have missed a bug). Extreme cases of the sampling dimension are “all valid products” (A) and “a single product only” (B). In contrast to previous work [17], we consider sampling in combination with all other dimensions, not just with product-based analyses.

**Feature Grouping.** The second dimension is “feature grouping” (e.g., edge A–E); it represents the granularity and size of feature combinations that is considered during analysis. A corner case represents analyses considering features in isolation (vertex E), which are called *feature-based*. Examples include collecting metrics of code associated with individual features or certain intra-procedural static analyses that can operate on isolated code snippets [17]. Analyses whose results may depend on runnable programs or feature interactions are difficult to implement this way. To check for feature interactions, we could use a feature-based step to extract “feature interfaces”. In the next step, we would check whether the interfaces fit each other (linker check, [1, 13]). Another corner case are analyses that consider only entire, valid products (A), as done in the product-based approach. Cases inbetween include analyses that strategically group features to form compound units and analyze these units as a whole. Examples are analyses that search for critical feature interactions among combinations of the most common features.

**Variability Encoding.** The third dimension is “variability encoding” (edge A–D); it represents the extent to which variability is preserved and used during analysis. At one end, we have the traditional approach, where each product is created and analyzed separately with standard tools (*product-based*, A). At the other end (D) is the *family-based* approach that encodes all features, information on variability, and valid configurations into one analysis subject that simulates all products of the product line (e.g., for verification as in Figure 3). Because the simulator contains information about all products, only a single analysis pass (e.g., model checking) is necessary. To implement a family-based analysis, we need an analysis tool that can handle the vari-

ability information in the simulator. Some tools such as model checkers can handle the variability out-of-the-box by an exhaustive state-space exploration (e.g., [3]), but in other cases, a special analysis tool might be needed. In the range in between A and D, the analysis divides the product line in several sub product-lines and builds a smaller simulator for each sub product-line. This way, still the whole product line is covered, but the simulators are smaller and the analysis might be feasible even if a simulator for the whole product line exceeds system resources (such as main memory).

**Combinations.** The remaining vertices of the cube (F, C, G and H) represent combinations of the described analysis patterns. To combine the approaches, we apply different techniques successively.

Vertex F represents an analysis that targets only one single feature. For example, if we found that feature *Duplex* contains bugs, we would analyze this feature in isolation. In this example, we first used sampling to select one feature and then used feature-based analysis.

Vertex C combines sampling and family-based analysis. For example, we could run a family-based analysis on a product line to determine erroneous products. Subsequently, we would run a detailed debugging analysis on only one of the identified products, assuming that the identified bugs also appear in other products.

Vertex H combines feature-based and family-based analyses. For example, an analysis could count the lines of code per feature (feature-based) and aggregate them with variability encoding (family-based) to print the size of all possible products.

Vertex G represents an analysis that uses family-based techniques to analyze a single feature. This corner is probably not useful in practice, however, there are very useful analyses on the way to vertex G. For example, we could ask a domain expert to select groups of possibly interacting features, build simulators from them, and analyze the simulators for feature interaction detection. This is a cost-effective analysis, because we consider only a subset of features, and we can focus on the points where they interact. This analysis is situated in the middle of the cube near vertex G.

## 3.2 Formal Model

Next, we present the PLA model, a formal model for the description and comparison of product-line analyses. The model defines four operators that can be used recursively to build complex expressions. The operators are based on the dimensions of the PLA cube and denote integration and processing steps. The expressions denote compound analyses (i.e., combinations of integration and processing steps). First, we introduce the operators informally and, second, we provide a formalization.

The operators are general in the sense that they can be used to describe any product-line analysis. However, they are not complete. To describe complex analyses, the operators are extended by textual descriptions. The model helps to describe and illustrate the core concepts of the analyses.

**Operators.** We define four operators for the description of product-line analyses. The operators work either on basic features or on the results of other operators, as illustrated in Table 1. We call the input and output items of operators *objects*. We introduce the operators for two purposes: first, to



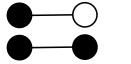
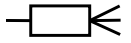
Operator	Visual	Definition	Signatures	Description
Fixed combinator		$\rho_C(A_1, A_2, \dots)$ $\rho_V(A_1, A_2, \dots)$	$\rho_C : \mathbb{C}^n \rightarrow \mathbb{C}$ $\rho_V : \mathbb{V}^n \rightarrow \mathbb{V}$	Combines all operands into a new object where the operands are fixed
Variability combinator		$\nu(A_1, A_2, \dots, m)$	$\nu : ((\mathbb{C} \cup \mathbb{V})^n \times M) \rightarrow \mathbb{V}$	Generates a variable object from operands (variable or fixed) with variability model $m \in M$
Variability restriction		$\dagger_C(A, sel)$ $\dagger_V(A, sel)$	$\dagger_C : (\mathbb{V} \times S) \rightarrow \mathbb{C}$ $\dagger_V : (\mathbb{V} \times S) \rightarrow \mathbb{V}$	Restricts the variability according to a feature selection $sel \in S$ . The sink circle is unfilled iff no variability remains
Processing step		$\tau(A)$	$\tau_C : \mathbb{C} \rightarrow \mathbb{C}^n$ $\tau_V : \mathbb{V} \rightarrow \mathbb{V}^n$	Applies a processing step to a fixed or variable object; the results are fixed or variable depending on the input object

Table 1: Operators of the Product-Line-Analysis model

give abstract descriptions of existing complex analyses and, second, to develop and describe new analyses. We describe briefly our experience with the tradeoffs of each operator.

The **fixed combinator** ( $\rho$ ) takes two objects and combines them to a (partial) product, in which both objects are fixed. In the resulting product, both operands are always activated and cannot be deactivated any more. If the operands contain variability (e.g., two existing product simulators), the variability is preserved in the result (e.g., a composed product simulator). An example of the fixed combinator is the composition of two feature modules using a code generator such as FEATUREHOUSE [2]. The result contains the artifacts from both operands as shown in Figure 2. Normally, features cannot be extracted from the result of a fixed combination. Our experience is that an analysis that uses only the fixed combinator is relatively cheap, because the analyzed object contains little or no variability. The downside is that one needs to build and analyze many products (exponential explosion). We use solid lines leading to a circle as visual notation for the fixed combinator. The circle is filled if and only if the resulting object contains variability.

The **variability combinator** ( $\nu$ ) takes two objects and a feature model, and combines the objects into one object. The combinator retains variability information according to the given feature model. The resulting object is a simulator in which each operand can be switched on and off at a later point in time (e.g., as in Figure 3 at runtime). An analysis that uses only the variability combinator with an unmodified feature model produces one simulator that includes the variability of the entire product line. Our experience is that the analysis of the simulator is efficient, because the analysis examines shared parts only once. However, the analysis of the simulator is more expensive than the analysis of a single product, which can cause problems with limited system resources such as main memory. We use dashed lines as visual notation for the variability combinator. The lines lead to a filled circle representing an object with variability information.

The operator **variability restriction** ( $\dagger$ ) is used when existing variability needs to be fully or partially eliminated. The operator takes one argument that contains variability (e.g., a variable piece of code or a product simulator) and restricts it according to a given feature selection. A well-known example of this pattern is the C-preprocessor CPP.

If one applies CPP to a product line that contains features implemented with `#ifdef` annotations, CPP eliminates all annotations and leaves only the artifacts that correspond to a given feature selection. This operator has essentially the same tradeoffs as the fixed combinator (relatively simple analysis of resulting objects, but potentially many objects that have to be analyzed). In contrast to the fixed combinator, which combines multiple objects, variability restriction takes one object with variability and restricts variability. In the visual notation the variability-restriction operator is denoted by a solid line leading from a filled to a filled or unfilled circle, depending on whether the result still contains variability information.

The operator **processing step** ( $\tau$ ) represents an analysis or a pre processing step that is performed on objects (features or combined structures). The processing step produces results that can, again, be aggregated with other transformation operators. An example processing step takes feature source code and filters the code for some properties. In the context of a locking protocol, we are interested only in the calls to functions such as `lock` or `unlock`. So, we can write a processing step that filters these calls from the source code. The filtered features are then processed with other operators. The main analysis can be executed more efficiently, when uninteresting information has been filtered as early as possible. Our experience is that the efficiency of the processing step depends on whether it is applied to an object with or without variability and whether the analysis tool is variability-aware, that is, the tool can recognize parts of the analyzed object that are not influenced by variability and analyze these parts only once. We use a box as visual notation for a processing step.

**Combination of Operators.** The four operators can be combined to form complex and powerful analyses that combine the advantages of several operators while reducing the disadvantages. If we want to run an analysis on the printer product-line introduced in Section 2, we could use any of the patterns shown in Figure 5 and more.

- The left pattern creates all six products by applying the fixed combinator in a brute-force fashion and analyzes the resulting products individually. This product-based approach leads to many analysis runs.

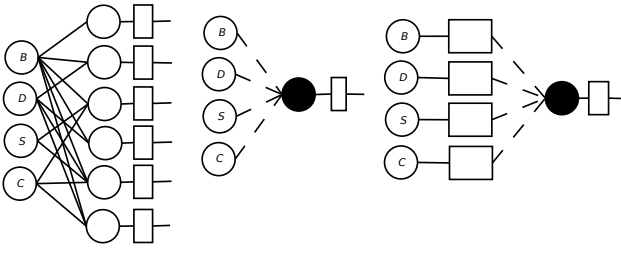


Figure 5: Three possible operator combinations for the printer product line (features are abbreviated with first letter). The operators are always applied from left to right.

- The pattern in the middle uses the variability combinator to generate a product simulator incorporating all features. The analysis of the simulator requires special tools that can cope with the variability information. The product simulator captures the behavior of all products and therefore it might get very large and costly to analyze.
- The right pattern applies a processing step on each feature before combining the results with the variability combinator. This can lead to a small simulator that can be analyzed more efficiently. For example, we can use the processing step to filter interesting functions (e.g., locking and unlocking). The simulator would be much smaller and contain only the locking behavior that we are interested in. However, implementing this analysis pattern requires three tools: an abstraction mechanism for filtering, a tool to combine the abstraction results into a simulator, and a tool that can analyze the abstract simulator. Few off-the-shelf tools provide these functionalities, so this analysis requires special tools. This implementation effort causes a higher upfront investment for the right analysis pattern than for the left or the middle one.

**Formalization.** We denote the set of all feature implementations by  $\mathbb{F}$  (every element  $f \in \mathbb{F}$  represents all code artifacts belonging to feature  $f$ ). Derived objects are partial products  $c_1, c_2, \dots \in \mathbb{C}$  and simulators  $v_1, v_2, \dots \in \mathbb{V}$ . Each partial product  $c \in \mathbb{C}$  and each simulator  $v \in \mathbb{V}$  is built from a fixed set of features  $fs \subseteq \mathbb{F}$ . Each simulator can simulate the behaviors of all valid products that can be built from the respective set of features. Each single feature implementation is also a partial product (with only one feature), so  $\forall f \in \mathbb{F} : \{f\} \in \mathbb{C}$ . The subset  $\mathbb{P} \subseteq \mathbb{C}$  denotes the set of valid products according to the feature model  $FM$  of the product line. In #ifdef product lines, the implementation artifacts already contain variability. In terms of our model, these implementation artifacts are product simulators (elements of  $\mathbb{V}$ ). Again, we use the printer product line to illustrate the meanings of the sets  $\mathbb{F}$ ,  $\mathbb{C}$  and  $\mathbb{P}$ :

- $\mathbb{F} = \{BasicPrinter, Duplex, Scan, Copy\}$
- $\mathbb{C} = \mathcal{P}(\{BasicPrinter, Duplex, Scan, Copy\})$
- $\mathbb{P} = \{\{BasicPrinter\}, \{BasicPrinter, Duplex\}, \{BasicPrinter, Scan\}, \{BasicPrinter, Scan, Duplex\}, \{BasicPrinter, Scan, Copy\}, \{BasicPrinter, Scan, Duplex, Copy\}\}$

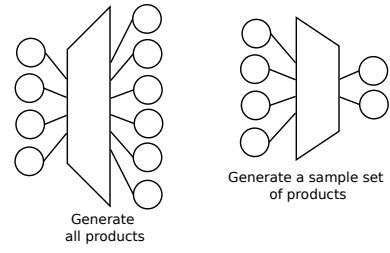


Figure 6: Simplified notation for recurring product-line analysis patterns. The notation on the left side represents the generation of all valid products and the notation on the right side represents the generation of a subset of all valid products (sampling).

Based on the basic objects, we define operators as formalized in Table 1. The fixed combinator ( $\rho$ ) is applied to objects that may contain variability ( $\mathbb{C}$  or  $\mathbb{V}$ ) and combines them into a new object that always covers the behavior of both operands. So far, we discussed only operators with binary cardinality, but they can be generalized for application to more than two objects. The variability combinator ( $\nu$ ) is applied to objects with variability ( $\mathbb{V}$ ) or without variability ( $\mathbb{C}$ ). The resulting object is always variable ( $\mathbb{V}$ ) (the behavior encoded in the operands can be switched on and off). The variability restriction ( $\dagger$ ) operator takes one object with variability (one element of  $\mathbb{V}$ ) and eliminates variability according to a given feature selection. A processing step ( $\tau$ ) is applied to any object in  $\mathbb{C}$  or  $\mathbb{V}$ . The result of the processing step is a set of objects which can contain variability information if the operand is in set  $\mathbb{V}$ .

**Sampling by Combining Operators.** In our model, sampling can be accomplished in several ways.

When using the fixed combinator, a sample set can be build by constructing only some valid products.  $\tau(\rho(B, S))$  is a sampling analysis of the printer product line based only on the features *BasicPrinter* and *Scan*.

The variable combinator has an operand that determines the variability model of the resulting object. This operand is used to restrict the analyzed products to a set of interesting configurations. To this end a restricted variability model that only allows the interesting configurations is used as operand. If only processing steps on features are used, then sampling can be accomplished by analyzing only a subset of features.

### 3.3 Notational Sugar

We noticed two patterns that occur in many analysis descriptions. Given that they are quite cumbersome to draw, we introduce notational sugar that can be used as shortcut. The first pattern is the generation of all valid products from a given set of features. In our model, this pattern is expressed by multiple applications of the fixed operator (shown on the left side of Figure 5). As shortcut, we denote this pattern by a trapezoid, as shown on the left side of Figure 6. The second pattern is the application of a sampling heuristic where only a subset of all valid products are generated. We denote this pattern by a reversed trapezoid shape that reflects that only few products are generated, as shown on the right side of Figure 6.

## 4. ANALYSIS EXAMPLES

In this section, we discuss existing analyses by means of our formal model, to demonstrate the capabilities of our model for systematic description and comparison of product-line analyses. Our experience is that it was relatively easy to express these high-level descriptions of complex product-line analyses using our model. Also, the descriptions, including visualization, are very space efficient (we fit three complex analyses on roughly one page).

**Product-line verification with variability encoding.** In the past, verification of functional properties of software product lines was often limited to the verification of abstract models or to the verification of a sample set of products. In recent years, several researchers [3,7,8,11,18] have developed analysis techniques that can be summarized as variability-encoding techniques. They aim at encoding the functional behavior of all valid products of the product line in one product simulator and use model checking or theorem proving to verify the correctness of the simulator. If the simulator can be proven correct, then all products satisfy the functional properties. Features that are unknown when the simulator is generated cannot be included subsequently, so this is a closed-world approach.

In terms of our model, the techniques use a variability combinator to build the simulator and then use a processing step for model checking. In the work of Classen et al. [7, 8] and Thüm et al. [18], the first step was done manually, whereas Apel et al. [3, 4] used an automatic combination tool. In terms of our model, both analysis approaches are expressed by the schema shown in Figure 5 (middle), and they are located on vertex **D** in the PLA cube (Figure 4).

**Modular verification of product lines.** Li et al. [13] used modular verification to check the correctness of features in an open-world setting. In an open world, features are developed in isolation, possibly by different teams. Yet, the features have to satisfy specifications when working together. To prove feature compatibility, Li et al. have used a feature-based processing step, in which they analyze individual features and extract interfaces. During this processing step, they already search for specification violations in the features. Then, the feature-based interfaces are aggregated in a combination step (fixed combination) for a subset of products. The resulting abstractions of products are analyzed to detect feature interactions that cause violations of the the product-line specifications. This approach incurs less effort than analyzing all concrete products. This analysis approach can be expressed in our model as shown in Figure 7. The analysis uses two sub analyses. The first sub analysis is feature-based (vertex **E** in the cube) and the second is product-based (vertex **A**).

**Detection of non-functional feature interactions.** Siegmund et al. [16] have developed an analysis technique to automatically detect feature interactions based on performance measurements. In this context, a feature interaction between two features is a change in application performance that is only measured when both features are present. For their analysis, Siegmund et al. use a combination of feature-based and product-based sampling strategies, which is shown in Figure 8. In the PLA cube (Figure 4), it is

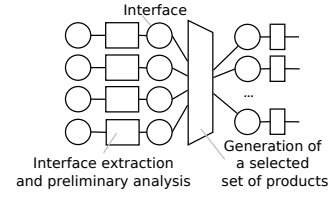


Figure 7: Pattern of the analysis of Li et al. [13]. In the first step, all features are analyzed and interfaces of the features are extracted. Then, these interfaces are combined and analyzed to find critical feature interactions.

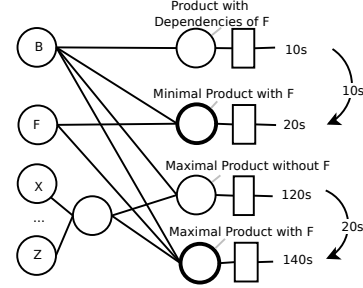


Figure 8: Pattern of the analysis of Siegmund et al. [16].  $F$  denotes the feature under analysis,  $B$  is the base product  $F$  depends on, and  $X$  to  $Z$  are all other features. The circles with bold frame denote products that contain feature  $F$

located on the plane between vertices **A**, **B**, **F** and **E**, because grouping and sampling are combined. The analysis determines whether there is an interaction between feature  $F$  and any other feature in the product line. The analysis calculates the set  $B$  of features that  $F$  depends on. It measures the performance of, (1) the product that contains only the features  $B$ , and (2) the product that contains the features  $F$  and  $B$ . The difference between these measurements is used as prediction for the performance of  $F$ . In this example, the influence which the feature  $F$  has on performance is estimated to be 10 seconds. To detect interactions, the analysis builds another pair of products that contain more features (again the only difference between the products is  $F$ ). If the difference between the measurements of the second pair (20 seconds in this example) is different from the first pair, then there exists an interaction between  $F$  and some other feature. The analysis is sample-based, because it uses a systematically chosen set of four products. To determine which feature is interacting, one can systematically evaluate more pairs of products [16].

**Type checking variability-aware ASTs.** Kästner et al. [12] developed an analysis for type checking of large annotation-based product lines in C. The analysis is used to check files of the Linux kernel. The analysis can be expressed in terms of our formal model as shown in Figure 9. Each file contains C-code with variability annotated in CPP directives. As first step, Kästner et al. parse and type check each file separately. The type checking step produces possible type errors and a type interface of the file. In the second step, the type interface is checked for compatibility with type interfaces of other files (linker check). The analysis uses a fixed combinator (interface combination) on objects containing variability.

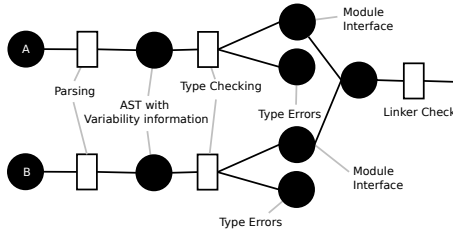


Figure 9: Visualization of an analysis by Kästner et al. [12]. *A* and *B* denote the files under analysis. Both contain variability information (in `#ifdef` annotations). The analysis executes parsing and typechecking on the files and then checks the compatibility of the computed type interfaces.

Therefore the analysis developed by Kästner et al. is located between vertices **A** and **D** in the PLA cube.

## 5. CONCLUSION AND FUTURE WORK

As it is often infeasible to analyze each product of a product line individually, researchers have developed analyses specially tailored to take the variability in product lines into account. To investigate patterns for efficient variability-aware analyses, we identified reoccurring patterns in product-line analyses and built a model for classification and comparison of analyses (called PLA model).

We demonstrated the usefulness of the model for the comparison of analyses by means of existing product-line analyses. Our initial experience with the model suggests that it is helpful for the description of complex analyses. We also expect it to be beneficial when designing new analyses, because one can select from a catalog of clearly structured analysis patterns. Our work is a step forward in making product-line analyses easier to understand and to simplify the development of future product-line analyses.

In future work, one interesting analysis seems to be the application of a family-based analysis on semantic interfaces that are extracted from features in a preprocessing step. It would also be interesting to develop guidelines for positioning of a given analysis on the PLA cube. Another area we want to explore is the relation between analysis patterns expressed in our model using rewrite rules on the patterns. For example, we could investigate under which assumptions a rewrite rule that replaces an analysis on each product by an analysis on a simulator holds.

## Acknowledgments

This work is supported by the DFG grants AP 206/2, AP 206/4, and SCHA 1635/2-1. We thank Sergiy Kolesnikov for fruitful discussions on the formal model and the anonymous reviewers for their helpful comments.

## 6. REFERENCES

- [1] S. Apel and D. Hutchins. A calculus for uniform feature composition. *ACM TOPLAS*, 32(5):19:1–19:33, 2008.
- [2] S. Apel, C. Kästner, and C. Lengauer. FeatureHouse: Language-Independent, Automated Software Composition. In *Proc. of ICSE*, pages 221–231. IEEE, 2009.
- [3] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Detection of Feature Interactions using Feature-Aware Verification. In *Proc. of ASE*, pages 372–375. IEEE, 2011.
- [4] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for Product-Line Verification: Case Studies and Experiments. In *Proc. of ICSE*. IEEE, 2013. to appear.
- [5] I. Cabral, M. Cohen, and G. Rothermel. Improving the testing and testability of software product lines. In *Proc. of SPLC*, pages 241–255. Springer, 2010.
- [6] Sheng Chen, Martin Erwig, and Eric Walkingshaw. An error-tolerant type system for variational lambda calculus. In *Proc. of ICFP*, pages 29–40. ACM, 2012.
- [7] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic model checking of software product lines. In *Proc. of ICSE*, pages 321–330. ACM, 2011.
- [8] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems: Efficient verification of temporal properties in software product lines. In *Proc. of ICSE*, pages 335–344. ACM, 2010.
- [9] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.
- [10] K. Czarnecki and U. Eisenecker. *Generative programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
- [11] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *ACM TOSEM*, 21(3):14:1–14:39, 2012.
- [12] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. A variability-aware module system. In *Proc. of OOPSLA*, pages 773–792. ACM, 2012.
- [13] H. Li, S. Krishnamurthi, and K. Fisler. Modular verification of open features using three-valued model checking. *Automated Software Engineering*, 12(3):349–382, 2005.
- [14] Harry C. Li, Shriram Krishnamurthi, and Kathi Fisler. Interfaces for modular feature verification. In *Proc. of ASE*, pages 195–204. IEEE, 2002.
- [15] S. Oster, M. Zink, M. Lochau, and M. Grechanik. Pairwise feature-interaction testing for SPLs: Potentials and limitations. In *Proc. of SPLC*, volume 2, pages 6:1–6:8. ACM, 2011.
- [16] N. Siegmund, S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake. Predicting performance via automated feature-interaction detection. In *Proc. of ICSE*, pages 167–177. IEEE, 2012.
- [17] T. Thüm, S. Apel, C. Kästner, M. Kuhlemann, I. Schaefer, and G. Saake. Analysis strategies for software product lines. Technical Report FIN-004-2012, University of Magdeburg, 2012.
- [18] T. Thüm, I. Schaefer, S. Apel, and M. Hentschel. Family-based deductive verification of software product lines. In *Proc. of GPCE*, pages 11–20. ACM, 2012.