RWTH Aachen University
Software Engineering Group

FH Aachen, Campus Jülich
09 Medical Engineering and Technomathematics

Comparison of State-of-the-Art Analyses for Software Product Lines

Seminar Thesis

presented by

Mingers, Joshua

MatrNr: 3244971

1st Examiner: Prof. Dr. rer. nat. Bodo Kraft

2nd Examiner: David Schmalzing, MSc.

Aachen, December 31, 2021

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Seminararbeit mit dem Thema

Comparison of State-of-the-Art Analyses for Software Product Lines

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Ich verpflichte mich, ein Exemplar der Seminararbeit fünf Jahre aufzubewahren und auf Verlangen dem Prüfungsamt des Fachbereiches Medizintechnik und Technomathematik auszuhändigen.

Name: Joshua Mingers

Aldenhoven, den 31.12.2021

Unterschrift des Studenten

Abstract

Product lines are defined as “a group of closely related commodities made by the same process and for the same purpose and differing only in style, model, or size.” The concept of product lines has been established in the manufacturing industry decades ago as a means of increasing productivity.

After product lines had helped the manufacturing industry grow, the underlying concept was applied to the field of software engineering. This combination has been named software product line.
Software product lines are interesting because they provide software engineers with several potential benefits, such as reduction of development cost and increased quality. A common requirement in software engineering is that a system can be analyzed and validated (i.e., properties of a system can be proven). However, software product lines cannot simply be checked with the techniques that have been developed for single-system engineering. The techniques must be adapted, if they are applicable at all. If they are applicable, they might prove highly inefficient due to their implementation. This does of course raise questions, most prominently: Which analysis approaches and techniques should a software product line framework support and how can those be performed in an optimized way?

In this thesis, we will explore how analysis techniques from single-system engineering can be applied to software product lines. We will compare different approaches in an attempt to determine which use cases they would fit best. In the process, we will consider two perspectives: the software product line framework developer’s and the framework user’s.

Contents

1 Introduction

While knowing a formal definition of the term "product line" might not be common, an intuitive understanding of it is relatively widespread. The popular dictionary Merriam-Webster defines a product line as "a group of closely related commodities made by the same process and for the same purpose and differing only in style, model, or size."[MW] Given the linguist's definition alongside our natural understanding, it becomes apparent that product lines are all around us.
A typical example is a car model. The base for the car is always roughly the same - it has a motor, a drivetrain, a chassis, etc. But the manufacturer offers customization, of course. For example, you might choose to have a heated windshield. The configurator you are using then informs you that selecting this requires having heated side mirrors. While it might not seem like much, you have already been presented with several aspects of product lines by now.

Another example of a product line can be found in most pockets nowadays. A smartphone matches what we described above, but the device itself is not the only result of a product line. The basis for the phone's operating system, the linux kernel, can also be considered a product line [TLD+11]. To be more specific, it can be considered a software product line.

After product lines had helped the manufacturing industry grow, the underlying concept was applied to the field of software engineering. This combination has been named software product line. Software product lines are interesting because they provide software engineers with several potential benefits, such as reduction of development cost and increased quality [PBL05]. A common requirement in software engineering is that a system can be analyzed and validated (i.e., properties of a system can be proven). However, software product lines cannot simply be checked with the techniques that have been developed for single-system engineering. The techniques must be adapted, if they are applicable at all. If they are applicable, they might prove highly inefficient due to their implementation. This does of course raise questions, most prominently: Which analysis approaches and techniques should a software product line framework support and how can those be performed in an optimized way?

In this thesis, we will explore how analysis techniques from single-system engineering can be applied to software product lines. We will compare different approaches in an attempt to determine which use cases they would fit best. In the process, we will consider two perspectives: the software product line framework developer's and the framework user's.

The structure of the thesis is as follows. We begin by establishing the basics necessary to describe and discuss software product line analyses in chapter 2. In chapter 3 we describe different approaches to analyzing software product lines. The approaches are compared in chapter 4 and related work is presented in chapter 5. Lastly, a summarization of the previous chapters and a conclusion is given in chapter 6.

2 Preliminaries

In this chapter we will present the basic concepts upon which this thesis is built. Product lines will be defined more rigorously than in the introduction. We will also be introducing the analysis techniques that will be considered in chapter 3. Afterwards, specifications that can be applied to product lines will be established. Since the criteria by which we intend to qualify analyses are based on the work of Thüm et al., we will be using parts of their terminology, which includes properly introducing the respective terms [TAK+14].

2.1 Product Lines

A product line is a set of products which (a) typically share a *common core* and (b) are based on a common set of *features*. This concept has a high level of abstraction; i.e., there are no limits to what sort of product may be described by it. Since the products share a core and features, there is no need to specify each product separately. Rather, the core is defined only once, as is each feature. To get a concrete product from this, a subset of the features must be selected. Such a subset of features is called a *configuration*.

2.1.1 Variability Model

With the definition given above, the features would only be grouped in an unordered set. However, certain configurations may contain undesired feature combinations or lack required features. Thus, constraints need to be introduced, specifying which configurations are valid. This is achieved using *variability models*.

There are two common ways to define variability models. The first is a graphical representation, the *feature diagram* [KCH+90]. It is most useful for presenting variability models in a human-readable form.

As we can see in Figure 2.1, the feature diagram conforms to a tree-like structure. The root node contains the common core of all products. It can be considered a feature and is then typically referred to as the root feature. It is either a mandatory explicit selection or implicitly selected for any configuration, based on implementation. We can derive from this diagram that any valid configuration must have exactly one camera type selected. Valid configurations may also have an extra card slot selected, but not for both card types simultaneously. A restriction that cannot be represented in the tree structure is that having the PRO camera requires selecting the extra SD-Card slot. Restrictions that do not fit the tree structure can also be represented in text form below the diagram.

The second way to define a variability model is a *propositional formula*. Propositional formulae are logical statements which describe valid feature combinations. They are useful for checking the validity of a configuration using logic solvers or for finding possible partial configurations using satisfiability solvers. Any feature diagram can be converted into a

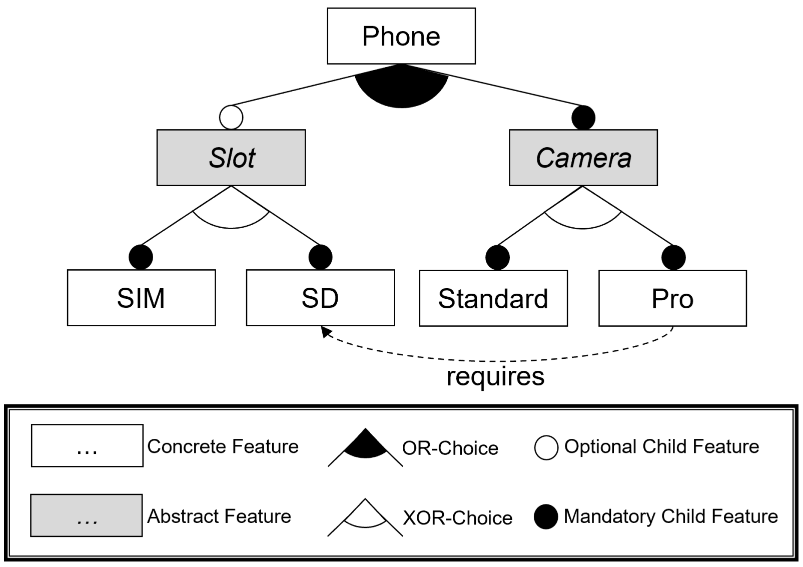

Figure 2.1: A simple example of a feature diagram, representing features of a minimally customizable mobile phone.

propositional formula and vice versa [Bat05]. The propositional formula in Equation (2.1) is equivalent to the feature diagram presented in Figure 2.1.

$$\begin{gathered} Phone \wedge \\ (Phone \Rightarrow Camera) \wedge \\ (Slot \vee Camera \Rightarrow Camera) \wedge \\ (Slot \Leftrightarrow SIMCard \vee SDCard) \wedge (\neg SIMCard \vee \neg SDCard) \wedge \\ (Camera \Leftrightarrow Standard \vee Pro) \wedge (\neg Standard \vee \neg Pro) \wedge \\ (Pro \Rightarrow SDCard) \end{gathered} \tag{2.1}$$

Each variable in the formula stands for the respective feature being selected (true) or deselected (false). If the whole expression evaluates to true, the configuration is valid. An example of a valid configuration would be one in which only the three features *Phone*, *Camera* and *Standard* are selected.

2.1.2 Software Product Lines

Having understood the concept of product lines and being familiar with software engineering, just thinking about the term software product lines will likely grant you an intuitive understanding of it. The main reason for developing software as a product line is that it can lead to an increase in productivity and quality in software development [PBL05]. The software product line approach promotes or forces reuse of code across different products by definition. This can reduce the amount of redundant code as well as the overall size of the codebase of a group of products. An additional benefit are reduced development costs for multiple products. As Figure 2.2 shows, developing a software product line comes with a high initial cost. However, once more than three products are derived from the product line, the cost per product is lower than having developed each product on its own. It has also been argued that time-to-market can be reduced significantly after an initially higher development time for the first few products [PBL05].

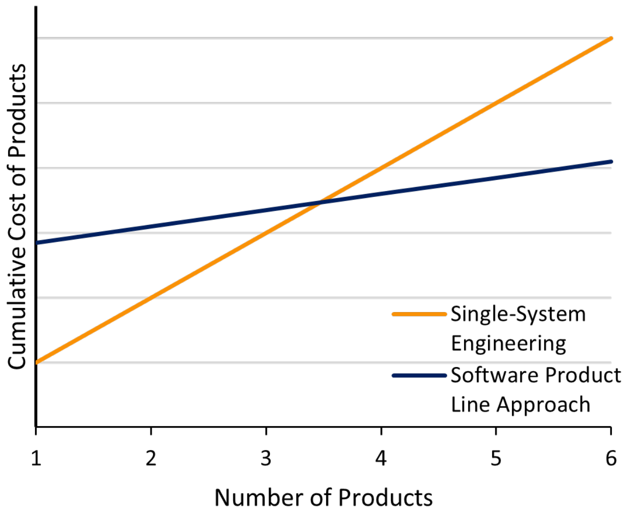

Figure 2.2: The cost of developing multiple software products in a classic single-systems approach vs. in a product-line driven approach [MNJP02].

The definition given in an influential book on software product lines is the following: “A software product line is a set of software-intensive systems sharing a common, managed set of features, [. . .] that are developed from a common set of core assets in a prescribed way.” [CN01] This definition extends the one we established for product lines in that it calls for the assets to be developed in a prescribed way. This is a necessary addition since the core implementation and all the feature implementations need to interact with one another. Ensuring such a seemingly basic property can become difficult when software is being developed in an arbitrary manner. Outside of this context several loosely related projects might be developed simultaneously. These then communicate with one another via APIs. While the APIs’ specifications need to be met, how they are implemented does not matter as much, since each project is its own system. When developing software to be used in a product line environment, a high degree of coupling can occur and might even be intended between the implementations of some features. This does already hint at problems that may arise when trying to analyze product lines. We will elaborate on them shortly.

Outlining the complete process of creating a software product line and deriving products from it would be far beyond the scope of this thesis. However, knowing the process and the artifacts it creates will be of importance when discussing analyses later. Thus, we will simplify or abstract the steps. As a premise it is important to know that product lines are typically developed in an according framework, not unlike most other software. Said framework may combine different tools: a generator and analysis tools for product lines are typical examples.

The steps necessary to create software products from a software product line can be as-

signed to two roles, the product line developer and the application developer. First, a product line developer creates a variability model based on specifications and/or requirements that have been determined beforehand. They are typically knowledgeable in the domain the product line targets, but do not usually have a specific application in mind. The variability model is defined early in the process, it may however be subject to change. This is due to flexibility being a major requirement, just like in other areas of software engineering. As we will see later, some approaches to analyzing software product lines handle this kind of openness better than others.
Next, the product line developer will start implementing features they defined in the variability model as well as the common core. The common core and feature implementations are referred to as *domain artifacts*. The tasks the product line developer performs can be grouped as *domain engineering*.
When the variability model is complete, an application developer can consult with users to create configurations that best suit their needs. The configurations they create must of course be valid according to the variability model.
After the domain artifacts and configurations have been completed, they can be passed to a generator which is typically bundled with the framework used to define the product line. The generator will create a software product for each given configuration. The steps that do not fall under domain engineering are considered to be part of *application engineering*.
Now that we have established how software product lines work, we will go over some of the unique problems that may arise.

Problem: An Ever-Growing Number of Configurations

A software product line is always designed with the intention of being flexible to allow several distinct products to be created. To this end, a large number of features might be introduced in the variability model. This can quickly cause the variability model to become unwieldy. In addition, the number of possible configurations might grow in an exponential manner. The worst case would be a variability model in which no feature is mandatory and no two features are mutually dependent, neither by exclusion nor requirement. For n features, this leads to 2^n valid configurations. We can also create a highly restricted example. For example, consider a variability model with n features, arranged in a binary tree, with each pair of features being mutually exclusive but no pair must remain unselected. In this example, the number of configurations grows linearly, n features lead to $\lceil n/2 \rceil$ valid configurations. Considering however, that product lines from industry applications have had more than 1000 features in the past, even linear growth can become overwhelming. While the number of valid combinations cannot be reduced without changing the variability model, techniques have been developed to extract key information even from large-scale models. These techniques are commonly referred to as feature model analyses [BSRC10]. They can assist developers in keeping an overview of a complex model which would otherwise become hardly manageable.

Problem: The Optional Feature Problem

As described above, in software product line development, the variability model and concrete implementation of features may not be developed simultaneously or by the same person. Discrepancies might arise between the inter-feature dependencies declared in the variability models and those present in the implementations of different features. While a dependency declared in the variability model without an equivalent in the implementation does not cause problems, this does not hold vice versa. Assume some feature `A` has already been implemented. If we now implement feature `B` and have a program dependency on feature `A`, said dependency must be reflected in the variability model in some way. Otherwise, we can likely create a valid configuration in which feature `B` was selected while feature `A` was not. This can cause a multitude of problems, ranging from type errors or the program not compiling to the product showing inexplicable or erratic behavior. A discrepancy between the variability model and the implementation of this sort is referred to as the optional feature problem [LBL06]. Detecting the optional feature problem can be a challenge in itself, though certain analyses presented in later chapters can help with this. Resolving this problem can be achieved in several ways, modifying either the variability model or the respective features' implementations, or a combination of both [KAuR+09].

Problem: General Flaws in the Variability Model

This subsection does not refer to a singular problem but to a group of problems. The optional feature problem is a member of this group, possibly the most prominent example. One might already conclude from said example that the variability model does not always allow only correctly functioning products. Even if it has been designed with great care, the variability model might permit combinations of features causing unwanted behavior. This is commonly caused by oversights during creation of the variability model, interactions of certain features' implementations or previously unknown specifics of a deployment environment. A myriad of other reasons might also cause unwanted behavior to occur. Approaches to detecting and resolving these problems are as varied as the different forms of flaws that fall into this category.

2.2 Software Analyses

We will be considering four types of software analyses which have all been applied successfully in single-system engineering. While they do not exactly share the same area of application, they do all prove useful in analyzing software product lines.

2.2.1 Type Checking

Type checking is implemented by default in all modern programming languages. While the specifics may vary greatly from language to language, the basics are the same. A type checker bases the analysis of a program on the specified type system. It can evaluate the type of expressions and determine if variables are assigned values of the correct type (e.g., determining `String s = 3 + 2.5;` would mean assigning a floating-point number to a string variable). Another common task is determining whether a type that is being used is defined in that scope (e.g., the type `ArrayList` cannot be used in a Java class unless it has been imported). Finally, checking if a variable, method, or function is defined in a given scope is an important part of type checking [Mit02].

2.2.2 Static Analysis

Two defining characteristics of static analyses are that they operate at compile time and that they use approximation so as not to exceed the boundaries of computability [NNH04]. Scenarios in which approximation is required are those in which undecidability of program termination occurs, e.g., loops or recursion. Examples of static analyses are data- and control-flow analyses, alias analyses and program slicing [NNH04, Muc98, Wei81]. Static analyses have been integrated into compilers or published in standalone tools. A popular example of one such standalone tool for Java would be FindBugs [HP04] or its successor SpotBugs [Tea]. Other areas of application include integration into IDEs, e.g., JetBrains' IntelliJ IDEA [JB20].

2.2.3 Model Checking

Model checking is a verification technique in which a given system is first described formally. Certain properties of the system are then proved or disproved, based solely on the formal description. Model checking is performed by a model checker which determines the languages that can be used. The entire verification process can be subdivided into three phases [BK08]. First, a system model and properties are described in the specified languages. A simulation can then be run to roughly assess if the model describes the system as intended. The second step is checking the validity of a property. If the property is satisfied, the next one is checked. If it is violated, the third phase starts. The model checker provides a counterexample which can be analyzed by the developer. The system design, model or property are then modified accordingly. Afterwards, the second step is restarted with the first property. If the model checker runs out of memory during phase two, the system model needs to be reduced. The phase can then be restarted. It must be noted that model checking cannot be implemented or applied without some expertise in the field of formal logic and verification. Another major drawback of model checking is the limited scaling capabilities on larger systems. The checker may run out of time or memory due to a well-known problem referred to as the "state-space explosion" [Sch01].

2.2.4 Theorem Proving

Theorem proving is the use of theorem provers and verification tools with the purpose of ascertaining that a given implementation fulfills its specification. Verification tools are able to generate theorems from a specification in a formal language and an implementation. These theorems are then processed by the theorem prover by applying inference rules on them. A prover can be either interactive or automated [Sch01]. An interactive theorem prover requires user input in the form of commands applying inference rules. In contrast, an automated prover does not require any input other than the logical formulae. While automated provers offer increased ease of use, interactive provers are generally more powerful. Automated provers are typically limited to first-order logic while interactive ones can also reason about higher-order and typed logic. If a theorem cannot be proved, a theorem prover will indicate which part of the theorem in particular could not be proved. A major drawback of theorem proving is the amount of expertise that is required to apply it, regardless of the type of prover used [CGK$^+$18]. Overall, theorem proving can be considered akin to model checking because both model checkers and theorem provers can be grouped as inference systems [Sch01].

2.3 Terminology for Specifications and Scopes

As we have established, product lines can become quite complex. In order to be able to concisely express which part of the product line or its products we want to refer to, we will introduce some terms. These terms were established by Thüm et al. as specification

strategies for software product line analyses [TAK$^+$14]. They will, however, not only be used to describe in what way a specification applies to a product line. They can define which parts of the product line artifacts are considered in an analysis.
In this section, we will be using the expression "(valid) configurations" for the sake of brevity. This means that the statement typically only applies to valid configurations, but certain contexts or applications may find it useful to allow invalid configurations as well.

Domain-Independent
The term domain-independent is based on the concept of domains, in this context referring to the thematic environment in which the software product line is based. This means anything domain-independent is independent of any concrete product line. Consider type safety for a given software product line framework. Being domain-independent means that the requirement of type safety holds for all product lines developed using said framework. Note that we limited our example to a single framework only for the purpose of clarity. Theoretically, domain-independent can also mean independent of framework, imagining an implementation for that might however become difficult.

Family-Wide
The term family-wide refers to all (valid) configurations and/or resulting products of one specific product line. Consider the example of mobile phones as a product line. A possible family-wide specification would be that all products must allow the user to make a phone call. To illustrate why we need a term that is slightly more specific than domain-independent: If this specification was made domain-independent in the respective framework, it would also apply to a product line of washing machines developed in the same framework.

Product-Based
The term product-based refers to a singular (valid) configuration and/or its resulting product. We can construct an abstract product-based specification with respect to the model presented in Figure 2.1: The product generated with the configuration $Phone \land Camera \land Standard$ must fulfill some property. Said property does not have to be fulfilled by the product originating from the configuration $Phone \land Camera \land Standard \land Slot \land SIM$.

Feature-Based
The term feature-based refers to all (valid) configurations and/or resulting products in which a specific feature has been selected. Referring to Figure 2.1 again, an example of a feature-based grouping would be all products originating from configurations in which the feature Pro was selected. For this variability model, said feature-based selection might be equivalent to a product-based selection because there is only one valid configuration containing the feature Pro. Analogously, a feature-based selection of products with feature $Phone$ selected might be equivalent to a family-wide selection since all valid configurations must contain the root feature.

Family-Based
The term family-based refers to all (valid) configurations and/or resulting products in which a specific set of features has been selected. Considering a model covering a bigger diversity of features compared to the variability mode introduced in Figure 2.1 allows us to reason better about the family-based specification. Selecting certain subsets of features can make family-based equivalent to feature-based (only one feature selected) or product-based (one complete configuration selected). These special cases should normally be avoided and the term feature-based or product-based should be used for clarity's sake. Still, family-based is the most flexible way of defining scopes.

3 Analysis Techniques for Product Lines

In this chapter we will present combinations of the analyses and specifications introduced in section 2.2 and section 2.3. Considering every available combination of analyses and specifications leads to a vast number of possibilities, especially because we will consider combined specifications. To this end, we decided to focus mostly on strategies present in literature that we would consider promising as well as introducing those strategies required to better explain the former. Finally, we want to talk about daisy-chained analyses as a contrast to multi-stage analyses.

3.1 Single-Stage Analyses

Single-stage analyses are approaches to analyzing software product lines in which only a single scope (as presented in Section 2.3) is considered. Single-stage is not supposed to mean that every analysis described in the following section can be completed in a single step. Rather, there is only one abstract step specific to implementing the approach for software product lines and no techniques are combined.

3.1.1 Product-Based Analyses

Applying known analysis techniques from single-system engineering to products of a software product line is called product-based analysis. Obtaining products from a product line is typically achieved using a generator that comes with the software product line framework. Any product-based approach must necessarily operate on generated artifacts. Mapping analysis results to domain artifacts can become difficult or even impossible in some cases [KS10]. Thus, searching for the source of an error can become a tedious task.

Unoptimized Approach

The simplest type of product-based approach is called exhaustive, comprehensive, or brute-force. This means applying an analysis technique to every possible product. It offers a number of benefits. First, completeness can be guaranteed for the analysis (i.e., each product that is invalid per this analysis will be found). While this may not seem like a significant advantage at first sight, we will see that this property is not common in other approaches. Additionally, any fault in the analysis (e.g., invalid products not being detected) can most likely be traced back to a fault in the underlying analysis tool (rather than the implementation specific to the product line). Another advantage is the minimal amount of analyses required if a change in the product line affects only a small number of products. Finally, it can be considered advantageous that such a product-based approach can be applied without knowledge of the complete product line (i.e., variability model, feature implementations and configuration of the product under consideration).
However, there are also severe drawbacks to the approach. As we have established, products are supposed to share features, which leads to a significant portion of computations

becoming redundant as more and more products are analyzed. Furthermore, a brute force approach does not scale well with growing variability models because of the rapidly growing number of products, as described in section 2.1.2. The approach might even be rendered completely unusable if the number of products becomes too great.
We could not find publications in which only unoptimized product-based analysis is proposed. This is most likely due to the severe disadvantages. The approach is still important in order to define a baseline regarding overall efficiency as well as soundness and completeness with respect to the underlying single-system analyses.

This approach may seem attractive to software product line framework developers because it is easy to implement and is very comprehensive. If a well-established tool is used to perform the actual analyses, the risk of false positives is also very low. Performing this analysis on a complete product line may of course take a long time, possibly limiting usability severely. From a software product line framework user's point of view, this approach might seem unattractive as it may run for a long time for bigger product lines. Since it is particularly simple, easy usage can be considered a plus. Viable use cases are (a) a software product line with a very limited variability model allowing only a handful of products and (b) applications in which completeness with regard to a certain base analysis must be guaranteed.

Optimized Approach

The product-based approach can be improved by reducing redundancy and the number of operations that need to be performed. The first way this can be achieved is by reusing results or intermediary results from previous analyses. Analyses can then be performed incrementally for subsequent products. However, deciding which results can be reused is not trivial. It requires knowledge of the variability model and configurations of previously analyzed models. A firm grasp of the applied analysis technique is needed to gauge which properties checked with the given analysis do not change for two given configurations. Furthermore, off-the-shelf tools do not necessarily support incremental analysis. A resulting increase in development costs is another potential disadvantage.
This optimized product-based analysis has been described more concretely for model checking in the past [Kat06, CSHL12b]. A preparatory product-based step was required for both, in which feature implementations are categorized. This categorization is then used to determine which results from previous analyses can be used in the analysis of certain other products. Both succeeded in identifying checks that did not need to be recomputed. However, neither research measured nor approximated how much computation time would be saved compared to an unoptimized approach if their approach was to be implemented. The concept of an optimized approach to product-based theorem proving has been explored by Bruns et al. [BKS10]. They managed to reduce the number of proofs required once a base product has been verified completely. A reduction of computation time between an unoptimized and an optimized approach has not been quantified here either.
The second option to reduce the number of operations required is to reduce the number of products that are being analyzed. Performing analyses on a subset of all possible products is called sample-based analysis. A typical coverage criterion is pair-wise coverage, which can be achieved for any pair of features (F, G). The set of selected products must contain products whose configurations include F and G, F but not G, and G but not F. This does of course assume that F and G are not mutually dependent. Applying the same concept to n-tuples of features leads to n-wise coverage. Operating under the popular assumption that most failures are caused by interaction of only a handful of parameters makes this approach seem very reasonable [KKL16]. However, completeness with regard to the base

analysis can of course not be guaranteed.

For software product line framework developers a sample-based approach is nearly as easy to implement as the unoptimized approach discussed above. The only difference is that the framework must allow the user to specify a subset of products to be used as a sample. Optimization by incremental analysis, on the other hand, is potentially significantly tougher to implement due to lack of tooling. For the software product line framework user the sample-based approach is most likely not particularly attractive because it cannot ensure completeness with regard to the base analysis. The incremental approach has only been presented for model checking and theorem proving thus far, neither of which are particularly easy to use.

3.1.2 Family-Based Analyses

Applying analysis techniques to domain artifacts of a software product line and incorporating knowledge of the variability model into the analysis is called family-based analysis. The variability model is converted to a logic formula (see Equation (2.1)) to make it accessible to the respective analysis tools. In some approaches, all feature implementations are merged into a single virtual product, referred to as "metaproduct", which is not necessarily a valid product.
The performance of family-based analyses is proportional to the number of features and the size of feature implementations. This is because they aim to avoid redundant computations by considering the variability model. Another positive aspect of family-based approaches is that they operate solely on domain artifacts. Thus, problems found by the analyses can be mapped to domain artifacts and the source of each problem can be identified.
A disadvantage of family-based approaches is that known analysis methods cannot be used exactly as they are, since such approaches intend to use knowledge of the variability model, information which classic tools are not typically able to interpret in a meaningful way. Some analysis problems have been encoded from software product lines into an existing formalism in order to allow off-the-shelf tools to be used, but it is unclear whether this is even possible for all the analysis techniques mentioned in this thesis [ASW^{+}11]. Such encodings have been performed for model checking and theorem proving in the past. Moreover, the usability of family-based approaches is limited in evolving product lines as even minor changes in either variability model or domain artifacts usually result in a new analysis of the complete product line. The impact of this can be mitigated by caching analysis results, yet significant amounts of work may still be required. Another drawback is that the size of the analysis problem may exceed physical boundaries (e.g., available memory) because the analysis may consider all domain artifacts at once. The final problem we want to consider is that a family-based approach operates in what is known as a "closed-world scenario", meaning that the complete variability model must be known in order to perform any analysis.
Several analyses have already been implemented for software product lines based on a family-based approach. A criterion by which they can be differentiated further is the point in time at which the variability model is considered. An implementation can use the variability model during the analysis to identify analyses that can only lead to false positives (i.e., problems found by the analysis that cannot occur in a valid configuration). This is called early variability model consideration. In some cases, however, the variability model is only used afterwards to identify false positives. This is called late variability model consideration.

Type checking for software product lines has already been performed successfully using a family-based approach for several kinds of variability implementations. The goal of the analysis is to prove that, if the product line is type safe according to the check, all products that can be derived are also type safe. In most implementations, the type system has been extended to incorporate knowledge of the variability model. Kolesnikov et al. describe key aspects of family-based type checking software product lines in comparison to single-system type checking [KvRHA13]. First, a syntax tree of the metaproduct is created. In order to represent the whole product line, it is enriched with variability information. The information stored alongside each program element contains the feature it belongs to. Furthermore, information about feature dependencies is included in the syntax tree. Then a special family-based type checker analyzes the enriched syntax tree. Said type checker is not an off-the-shelf tool. It must be able to work with variability as it is encoded in the given syntax tree. Kolesnikov et al. cache results of the SAT solver used to determine dependencies between features. For some systems they tested their implementation on, caching reduced the time required to typecheck the whole product line by more than two orders of magnitude. Most of the product lines they tested their approach on had coarse-grained features (i.e., a low number of features with rather large implementations). They suspect that the performance improvement gained by caching will be not be as great in systems with fine-grained features (i.e., many features with relatively small implementations).
Since type checking is a very basic analysis and a staple for most modern programming languages, it can be regarded as an expected feature of a software product line framework. This expectation is only furthered by type safety being a domain-independent specification. For this reason, every software product line framework developer should consider integrating type checking into the default checks of their tool, possibly in a family-based approach. The extent of the implementation workload as well as the specifics of how type checking is performed are of course highly dependent on the framework and the language used to specify the product line.

The family-based approach has also been used as a basis for static analyses specific to software product lines and as a means of applying existing static analyses to software product lines. One example of a static analysis specific to software product lines was proposed by Adelsberger et al. [ASN14]. Their analysis applies to dynamic software product lines and aims to evaluate the complexity of reconfiguration at runtime. As an example of already existing static analyses being applied to product lines, we want to briefly present the approach of Bodden et al. [BTR+13]. Their tool SPL$^{\text{LIFT}}$ works with analyses formulated in the IFDS framework for interprocedural dataflow analysis [RHS95]. Any analysis formulated for IFDS can be lifted to work with software product lines. The new analysis uses the related framework IDE [SRH96]. During the lifting process, means of considering the variability model are added to the analysis. While producing correct results obviously necessitates knowledge of the variability model, it was not used for early variability model consideration. SPL$^{\text{LIFT}}$ was tested on four existing product lines. The tool performed better in almost all experiments using late variability model consideration.
Static analyses are widespread and well-established in single-system engineering, and they can be useful tools for finding errors in a program. While this makes them attractive, the amount of work required might deter framework developers from implementing them in their software product line framework. The tool presented by Bodden et al. in [BTR+13] is advertised as relatively easy to use, which might lower the effort significantly. As we have not tested SPL$^{\text{LIFT}}$ we cannot say with certainty how easy it is to use. If it is as easily usable as we understand it, implementing family-based static analyses would become more

feasible. A software product line framework user might expect static analyses to be a staple of every framework because they are so common in tools for single-system engineering. For a family-based static analysis, we can ensure soundness and completeness with regard to the base analysis. Therefore, there should not be a reason not to use a family-based approach from the framework user's point of view.

Family-based model checking has been implemented in a number of ways. One of the most prominent characteristics distinguishing the implementations is whether the analysis operates on source code or on an abstraction of the system. System abstractions have been defined as, e.g., I/O automata [LMP10], transition systems [GLS08], or featured timed automata [CSHL12a]. Software model checking has been performed on software product lines written in C [AvRW^{+}13, ASW^{+}11] and Java [AvRW^{+}13, KvRE^{+}12]. The implementations have demonstrated that existing tools for model checking can be used or extended in order to perform model checking on software product lines in a family-based approach. Across the implementations, different formalisms were used to describe the specifications that were to be checked. The specification types ranged from domain-independent to feature-based and family-based. Thüm et al. state that most family-based model checking approaches consider the variability model during the analysis in order to skip analyses of invalid feature combinations [TAK^{+}14].
From a software product line framework developer's point of view, implementing family-based model checking as a default in the framework could be considered worthwhile because model checking is a powerful technique. The framework might offer some default domain-independent specifications and allow for users to easily add their own specifications of any type. One could expect to be able to use existing tools in order to more easily implement the approach. Still, some experience in the field of model checking would be required. From a software product line framework user's point of view, the advantage of model checking being a powerful tool still applies of course. A major drawback is that using this analysis technique requires expertise in the field of model checking in order to define custom specifications and possibly in order to properly interpret the results.

3.1.3 Feature-Based Analyses

Applying analysis techniques to domain artifacts while considering features and their implementations only in isolation is called feature-based analysis. Any form of feature interaction is disregarded in this approach.
The amount of work performed for feature-based analyses grows linearly, in proportion to the number of features and the size of the feature implementations. Considering features exclusively in isolation leads to only small amounts of effort required when a product line changes. Only changed features need to be reanalyzed and changes in the variability model do not result in the analyses having to be run again. Feature-based analyses can also be used in an "open-world scenario" since the variability model is not used in the analyses. Another advantage is that this approach can be seen as a whole, or it can be divided into several small analysis tasks which can be executed independently of each other. Thus, the risk of running out of memory is relatively low and the tasks can be run in sequence or in parallel. Finally, feature-based analyses operate on domain artifacts, similar to family-based approaches. This allows for problems found by the analysis to be eradicated at their root.
A major disadvantage of feature-based approaches is that they cannot be used to reason about problems occurring in the interaction of features. They are therefore also not able

to identify the optional feature problem we outlined in section 2.1.2. Furthermore, which kind of base analysis can be used in a feature-based approach has to be considered carefully. The property to be checked must be compositional across features (i.e., the results cannot be invalidated by feature interaction), in order to ensure soundness of the analysis approach with respect to its base analysis. Whether a given property is compositional is highly dependent on the way the concept of variability is implemented.

Software product line analyses implemented in a strictly feature-based manner are rarely found. The reason for this is the approach's inability to detect problems in feature interactions being such an impactful disadvantage. This needs to be counteracted by combining it with other approaches, a concept which we will be exploring in section 3.2. For the same reason, we do not consider the feature-based approach a feasible candidate for a single-stage analysis. It has been introduced in order to more accurately describe certain multi-stage analysis approaches.

Since features and the combining of those is a key aspect of using software product lines, analyzing combinations of features must be a key aspect of analyzing software product lines as a whole. A standalone feature-based approach is incapable of doing exactly that. Therefore, it should appeal neither to a software product line framework developer, nor to its users.

3.2 Multi-Stage Analyses

Multi-stage analyses are approaches to analyzing software product lines in which multiple scopes (see Section 2.3) are considered. To distinguish the concept of multi-stage analyses from single-stage or daisy-chained analyses, it is important to note that the different analyses will not run independent of each other. Rather, subsequent steps use the results of previous analyses in order to increase the overall performance of the approach in some way.

3.2.1 Feature-Product-Based Analyses

As per the definition given in the introduction to this section, a feature-product-based analysis approach consists of multiple stages, the kinds of which can of course be deduced from the term alone. The first stage is a feature-based analysis, the results of which are to be used in the following product-based analysis. We want to motivate this combined approach as an extension to the product-based approach. The idea of this extension is the reduction of redundant computations with the bigger goal of reducing the overall analysis effort.
While the preceding feature-based analysis can reduce redundancies in the product-based step, they cannot be avoided completely. A benefit of a feature-product-based approach is that the feature-based approach's inherent insufficiencies regarding non-compositional properties and feature interactions can be compensated by the product-based counterpart. The extent of products analyzed in the second step determines if non-compositional properties are checked completely. A feature-product-based approach operates in a combination of an open-world and a closed-world scenario. Thus, the whole approach should be considered a closed-world scenario, i.e., the complete variability model must be known in order to perform the analysis. Software product line evolution is handled moderately well. If a

feature implementation changes or a feature is added, not much work has to be done in the feature-based step, as explained in section 3.1.3. However, all products affected by a change need to be analyzed anew in the product-based step.

Feature-product-based type checking has been implemented previously [KvRHA13]. In the first step, intra-feature dependencies are typechecked and interfaces for further inter-feature type checking are created. These interfaces would, for example, contain information about provided and required fields, methods, and classes. The inter-feature type checking is performed next. The second step follows a product-based approach, either unoptimized or optimized in some form (e.g., using pair-wise sampling). Depending on the range of products examined in the second step, completeness can be guaranteed with regard to the base analysis.
The same principle has been applied to elevate the single-system analysis technique of model checking to software product lines. First, feature implementations are model checked and an interface is generated for each feature. Said interface is composed of behavior provided by the feature and behavior required from other features. Inter-feature compatibility is then checked in the product-based step.

The total implementation effort for a feature-product-based approach can be broken down into parts. A sensible combination of analyses has to be selected, and it must be determined how the results of the first step are handed to the second. Next, both analyses have to be implemented with the respective approach. The implementations are most likely customized and specific to the chosen combination of analyses. This customization also determines the implementation effort required for each individual analysis. Thus, the feature-product-based approach might seem unattractive to framework developers due to potentially high implementation effort and limited usability of off-the-shelf tools. Completeness with regard to base analyses cannot be guaranteed unless an unoptimized product-based approach is chosen in the second step. This makes the approach less viable for most software product line framework users searching for validation methods. It might however prove useful in applications where completeness is not as crucial, for instance in detecting bugs during development.

3.2.2 Feature-Family-Based Analyses

A feature-family-based analysis approach is characterized by a feature-based analysis followed by a family-based analysis which uses the results of the first step. The fact that redundant computations could not be eliminated fully in the feature-product-based approach is what motivates this approach. Since the redundancies occur only in the second step, it is logical to consider other options to replace it. Following up on a feature-based step with a family-based analysis helps mitigating drawbacks of both approaches.
The deficiencies of a feature-based approach are partially addressed by the family-based follow-up, which is able to reason about interactions across feature implementations. While the family-based approach is not particularly suited to evolving product lines, this can be partly compensated by the first step which deals with evolving product lines better. The risk of excessive memory consumption in family-based approaches is inherited by this combination. Like the feature-product-based approach, a feature-family-based approach combines open- and closed-world scenarios and should thus be considered to work completely only if the variability model is known.

Type checking has been implemented in a feature-family-based approach for composition-based product lines in the past. In both approaches presented by Thüm et al., constraintbased type systems were used [TAK+14]. Constraints were generated in a feature-based step, in which type references and dependencies were stored for each feature. The constraints were then used in the following family-based step in order to complete the process of type checking.

The attractiveness of this approach to software product line framework developers can only be reasoned about when considering an equivalent single-stage family-based approach. This is because a family-based approach can check everything a feature-based approach can cover. The increased implementation effort has to be outweighed by an increase in performance. Otherwise, preceding the family-based analysis with a feature-based step is not justified. To a framework user, there is no reason not to consider a feature-family-based instead of a family-based approach as long as it is performant.

3.2.3 Family-Product-Based Analyses

A family-product-based approach consists of a family-based step whose results are used in the second analysis which follows a product-based approach. Unlike previous approaches presented, completeness with regard to a base analysis can be guaranteed in the first step. This raises the question of why a second step is even needed. The combination can be motivated in two ways, both useful in explaining different advantages.
First, consider a product-based approach as the baseline. As presented in section 3.1.1, a major drawback of this approach is that a growing number of products renders it inefficient, if not unusable, as the product line evolves. It can be preceded with an appropriate family-based analysis in order to reduce the number of products that are checked.

To this end, the variability model and codebase can be considered in concert, as proposed by Tartler et al. [TLD+11]. The goal is determining a set of products such that each domain artifact and each piece of code are part of at least one analysis in the second step, effectively maximizing code coverage. The second step can then be any product-based analysis. This implementation trades the assurance of completeness with regard to the base analysis for better performance. This is because not all valid feature combinations are checked, thus some feature interactions causing unwanted behavior might go unnoticed. Therefore, it is more suited to bug detection than to validation.
Kim et al. [KBBK10, KBK11] developed a family-based approach used to determine which products could violate a certain specification. They elevated control-flow and data-flow analyses to software product lines in order to determine which features had what effect. The analysis resulted in a specialized variability model representing all models that needed to be checked. This approach was simply prepended to a product-based analysis step. The analysis was later generalized from safety properties to general test cases.

We may also consider a family-based approach the starting point. As we have established in section 3.1.2, such an approach may run the risk of exceeding memory limitations when the whole product line is considered at once. Thüm et al. [TAK+14] suggest a family-product-based approach as a possibility of addressing this issue. They propose partial family-based analyses, but no example of a concrete implementation nor an idea for such was found. A partial family-based analysis followed by a product-based approach checking whatever properties were not checked by the family-based analysis might avoid the family-based analysis running out of memory. However, the product-based follow-up would still have to be an approach that is optimized in some way, otherwise the ever-growing number of products may render the whole approach unusable again. The lack of empirical evidence makes it hard to reason about feasibility, we can only say that the approach seems not to be plucked out of thin air due to its solid theoretical basis.

From a framework developer standpoint, implementing the approach might again seem unattractive due to the high amount of work required. Whether the problem of exceeding memory can be solved with this approach is not clear, and completeness is dependent on the set of products selected for the second step. Tartler et al. [TLD+11] might have proposed the most interesting implementation for framework developers because their approach seems to be suited to supporting software product line development rather than just validation. The approach does not appeal to software product line framework users requiring completeness of analyses. It is likely interesting to users requiring certain specifications to be checked for a subset of all products, as the work by Kim et al. [KBBK10] shows.

3.2.4 Feature-Family-Product-Based Analyses

One might consider combining the three single-stage analyses into one process. While there are 6 possible orderings for the three approaches, only one is presented in [TAK+14]. The proposed ordering is a feature-based analysis followed by a family-based step and finally a product-based approach. This is the only sensible ordering for the following reasons: The product-based approach should always come last as its results are the least useful to other analyses. Furthermore, the product-based approach benefits greatly from the results of previous stages because the number of products that are being analyzed can be reduced. The feature-based analysis should take place before the family-based step because the feature-based analysis would become obsolete otherwise. While this combination may seem like a powerful one-for-all solution, it had actually never even been proposed before Thüm et al. published their work.

To the best of our knowledge, only one implementation following this approach has been presented since. Castro et al. presented several model checking approaches for software product lines in their work, including the first and so far only feature-family-product-based analysis [CLA+18]. They suggested that the feature-family-product-based approach is an alternative to the family-product-based approach in model checking under two conditions. First, the model under consideration is compositional. Otherwise, the feature-based approach could not be used to its fullest extent. Second, the model to be considered in the family-based step is too big to be analyzed efficiently. Since this is the only feature-family-product-based approach currently proposed, empirical data on its performance is not available. Therefore, discussing advantages and disadvantages of this approach cannot go beyond speculation.

This approach does not seem to exist outside a singular academic application. Thus, we suggest only software product line framework developers invested in researching analysis techniques should consider implementing it for now. It will most likely not appeal to software product line framework users for the same reason. If the concept gains popularity and empirical data on its performance becomes available, it may become attractive for practical applications.

3.3 Daisy-Chained Analyses

The term daisy-chaining analyses describes the sequential execution of analyses of all kinds. In contrast to the previously presented multi-stage approaches, results are not used in subsequent steps. This approach is rather simplistic, thus we do not intend to discuss it excessively. The reason for mentioning it nonetheless is equally simple: We do not want to give the impression that the multi-stage approach is a replacement for daisy-chaining analyses.

When daisy-chaining analyses, a choice has to be made for each step. Should subsequent steps be executed if the analysis finds problems in the program? The answer to this question depends on the use case and the specific analyses executed. The first example we want to consider is a static analysis used for finding unreachable blocks of code followed by a model checker verifying that the program fulfills some specification. In this case, there is no reason not to execute the second step if the first analysis finds unreachable code. This is because unreachable code is highly unlikely to hinder execution of a model checker or to invalidate the results. Now consider syntax checking as the first analysis. There is no need to execute another analysis if the syntax checker finds problems. Any subsequent analysis will most likely not be able to interpret the code correctly, analyses based on bytecode will not work because errors are thrown by the compiler.

Implementing the daisy-chaining of analyses is almost as simple as the approach itself. It is not intended to enhance performance of analyses or produce new results. Rather, it is a common feature, the lack of which might impede user experience. It can be a viable alternative to multi-stage analyses in situations where a less performant approach is favorable due to significantly reduced implementation time. For these reasons we believe a software product line framework developer should add simple means to daisy-chain analyses.

4 Comparison

In section 3.1 and section 3.2, we presented approaches for applying different software analyses to software product lines. In this chapter, we will compare the analyses based on the characteristics presented beforehand with the goal of ascertaining if there is an optimal strategy. We will not be considering the daisy-chaining analysis approach from section 3.3 in this comparison as it is more of a side note.

4.1 Comparing Single-Stage Approaches

Single-stage analyses were introduced comprehensively in section 3.1. We presented three approaches for applying analyses from single-system engineering to software product lines (product-based, section 3.1.1; family-based, section 3.1.2; feature-based, section 3.1.3). Exemplary implementations for each base analysis presented in section 2.2 were found in literature. Not every combination of analysis and approach has been proposed.

The easiest approach to reason about is arguably the feature-based one. As it prevents analysis of feature interaction, a standalone feature-based approach is almost unusable. The only possible application of a feature-based approach is syntax checking, which is hardly considered a software analysis in itself. The idea of defining an explicitly feature-based syntax checker might even be considered far-fetched, since the information which part of the code belongs to which feature does not offer any benefits.

Choosing between product-based and family-based is more dependent on the scenario.
If time consumption is of no concern and the main requirement is either easy implementation, integration of preexisting analysis tools, or definitive completeness, there is no better choice than a product-based approach, most likely unoptimized. The unoptimized product-based approach can be especially attractive for software product lines with small variability models and which are not intended to evolve to allow for more products. If absolute completeness is not required, an optimized product-based approach can be implemented almost as easily. Time consumption could be reduced if enough computational resources are available by running analyses on several products in parallel. It has to be noted that product-based approaches cannot operate on domain artifacts, possibly making the search for the root of an error significantly more complex.
The family-based approach is better suited to product lines with a larger number of products, there is however an upper bound. The approach may experience problems during analysis execution if the size of the metaproduct that is being analyzed exceeds the available memory. In order to properly incorporate variability information, analysis methods need to be adapted, thus off-the-shelf tools cannot be used. Carefully implementing the base analysis is imperative in order to be able to guarantee soundness. Manually extending single-system analyses and the associated risk of mistakes can be avoided if lifting tools are available. A family-based approach can guarantee completeness with respect to its base analysis as long as the approach was implemented correctly.

4.2 Comparing Multi-Stage Approaches

Multi-stage analyses were introduced in detail in section 3.2. We presented four ways of combining single-stage approaches (feature-product-based, section 3.2.1; feature-family-based, section 3.2.2; family-product-based, section 3.2.3; feature-family-product-based, section 3.2.4). Exemplary analysis implementations were given for each base analysis presented in section 2.2. Some, but not all combinations of approaches and analyses have been implemented in the past.

We start with the easiest aspect to compare. Not all approaches operate exclusively on domain artifacts. Three of the four presented approaches end in a product-based step, the results of which cannot be mapped to domain artifacts easily.
The remaining comparisons will depend on certain attributes of the properties under analysis.
First, the way product line evolution is handled is examined. None of the approaches are able to handle product line evolution well in every possible scenario. The family-product-based approach can never avoid redundant computations if domain artifacts change. For all other approaches starting with a feature-based step, it depends on whether the changed domain artifacts can be verified in the first step, how well they handle evolution. If that is not the case, the other approaches will perform redundant computations as well.
Next, we consider compositional properties. Advantageously, all combined approaches manage to avoid redundant computations. The family-product-based approach is however less favorable than other options because compositional properties are analyzed in the family-based step, leading to the risk of exceeding memory limits.
Finally, we inspect how non-compositional properties are handled. The feature-product-based approach relies on product-based analysis in order to check non-compositional properties. This leads to a tradeoff in either completeness or problem size (and thus runtime), depending on whether the product-based approach is optimized or not. The family-product-based approach must check non-compositional properties in a family-based step, again leading to the risk of exceeding memory limits. For the feature-family-based and feature-family-product-based approach, this risk can possibly be reduced by the preceding feature-based step. The greater the share of work that can be completed in the first step, the lower the risk of exceeding memory limits becomes.

4.3 Single-Stage vs. Multi-Stage Approaches

One might assume that multi-stage approaches are always superior to single-stage approaches because certain disadvantages and shortcomings of the latter are mitigated by combining them. However, the clear-cut lines drawn by the distinct advantages and disadvantages of single-stage approaches become blurred when combining the approaches. That is to say, while it is somewhat difficult to order the single-stage approaches by general quality or usability estimates, it only becomes harder for the multi-stage approaches. The comparison attempted in this section is complicated further as two sets of approaches which are not precisely ordered themselves are to be sorted together.
Comparing individual with combined approaches brings out what makes comparing any of these approaches so intricate. The contexts in which the approaches are to be used, the types of analyses that can be applied in accordance with the approaches, and the specifications the analyses are then intended to validate span a vast multidimensional space.

Consequently, pointing out for each pair of approaches along which axes of said multidimensional space they differ in what manner might turn out to be an endlessly tedious task. In general, we refer the reader to the arguments made in section 4.1 and section 4.2 in order to ascertain which approach is suitable for the scenario at hand.

We will be considering a handful of common scenarios (i.e., combinations of certain contexts, specifications, or analyses). They will be kept as generic as possible while still allowing meaningful statements about which approach or approaches would be best suited to them.
In scenarios where the number of products is very low, for example due to a small or severely constrained variability model, an unoptimized product-based approach is a valid choice. Implementation is quick, soundness can be guaranteed by a preexisting implementation of the base analysis and completeness is guaranteed by design.
Next, imagine a scenario with a complex variability model and a resulting large number of products, where completeness is not the highest priority. In this case, a feature-product-based or a family-product-based approach with product sampling applied in the second step is most likely the best option.
In a scenario with a complex variability model and a resulting large number of products, where completeness is required, our suggestion changes. A feature-family-based approach allows completeness to be achieved in the family-based step, however care should be taken to use the first step to its fullest extent in order to avoid running out of memory in the second step.
As a final example, we want to consider a medium-sized variability model with an extraordinarily low number of mutually exclusive features, leading to a large number of products. One could contemplate implementing a simple family-based approach. It allows ensuring completeness and should not run into memory problems with a medium-sized variability model. This way, the increased implementation effort of a multi-stage approach is avoided.

4.4 Comparison of Analysis Implementations in Different Approaches

When we planned the structure of the thesis, we had intended to compare how efficient a given analysis is implemented with different approaches. However, this type of comparison is not possible for most analyses due to a lack of data. Most works are not aimed at comparing approaches. Many are either propositions for implementations or proofs of concept for a single analysis in one specific approach. Works intending a comparison are often considering implementations of different analyses, not different ways of implementing the same analyses. One might consider comparing quantitative results from different papers. That is however not feasible because the experiments would have to be performed on the exact same product line. Results for one specific product line or a small set or product lines cannot easily be normalized or generalized.

5 Related Work

This chapter is used to present theses and topics related to this thesis in some way. We present more sources for theoretical foundations, additional analysis strategies that can be applied to software product lines and a source for examples of industry applications.

First, we want to mention a book that lays the foundation for software product lines as they are understood today [CN01]. In their work, Clements and Northrop present important aspects of product line engineering. They start with basic terms and present skill areas in which an organization should be capable in order to develop software product lines.
The framework of practices they presented in 2001 has since been updated [CN12]. Another comprehensive work dealing with software product line engineering was published in 2005 by Pohl et al. [PBL05].

Configurability is obviously the central feature of software product lines. This property is made possible by using the concept of variability. Thus, understanding variability and its implementation can help in understanding software product lines as a whole. Bachmann and Clements offer an overview of variability implementations for software product lines as well as guidance on how to select variation mechanisms [BC05].

Variability in software product lines is of course represented in a variability or feature model. Said model can become quite unwieldy, thus the concept of feature model analysis has been established. Benavides et al. present work regarding automated feature model analysis from 1990 to 2010 in a literature review [BSRC10]. A more recent publication is an extensive mapping study by Galindo et al., which includes Benavides' literature review [GBT^{+}19].

In addition to feature model analyses and the software analyses presented in this thesis, systematic testing of software product lines is a major field of research. A general overview of testing strategies is offered in a literature review by do Carmo Machado et al. [dCMMCd14]. They present approaches, but comparison results are limited due to a lack of empirical evidence from industry applications.

To see product lines as more than just a subject of research, practical examples are useful. We suggest "Software Product Lines in Action" by v. d. Linden et al. as a source of examples [LSR07]. In their book, they present ten case studies from industry applications of varying sizes on 150 pages. They do not only describe the product line but also the transition from developing in a single-system approach to developing in a product line driven approach.

6 Conclusion

In this thesis, we presented approaches to applying analyses known from single-system engineering to software product lines. We introduced the reader to the concept of software product lines and explained common analyses from single-system engineering. Three single-stage approaches were described and then combined, yielding an additional four multi-stage approaches. Different combinations of approaches and analyses were presented. At the same time, we highlighted various advantages and disadvantages of the approaches and certain combinations of approaches and analyses. The presented characteristics were then used in a comparison where we attempted to identify an optimal approach for applying analyses to software product lines.

As it turns out, whether an approach is suitable for a given scenario depends on a multitude of factors. Therefore, no "general best" approach could be identified. Instead, we opted to explain by example which approach is the best under which conditions. Furthermore, we provided the reader with knowledge with which they should be able to identify a suitable approach for a given situation themselves.

For future works, different areas of software product line engineering come to mind.
Staying close to the topic of this thesis, certain analyses could be implemented in different approaches with the intention of producing empirical data on the performance impact of the approaches.
This thesis was focused mainly on existing analyses from single-system engineering. The field of software product line testing as well as analyses specific to software product lines surely offer some refreshing new ideas.
Feature model analyses were disregarded for the most part in this thesis. Exploring how they can be used to improve the performance of software product line analyses might be a worthwhile endeavour.
Finally, we suggest researching ways of developing software product line frameworks. It may be interesting to see how much customizability can be offered by analyses integrated into the framework.

Bibliography

[ASN14] Stephan Adelsberger, Stefan Sobernig, and Gustaf Neumann. Towards assessing the complexity of object migration in dynamic, feature-oriented software product lines. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS '14. Association for Computing Machinery, 2014.

[ASW$^+$11] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. Detection of feature interactions using feature-aware verification. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, page 372–375. IEEE Computer Society, 2011.

[AvRW$^+$13] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. Strategies for product-line verification: Case studies and experiments. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 482–491, 2013.

[Bat05] Don S. Batory. Feature models, grammars, and propositional formulas. In *SPLC*, 2005.

[BC05] Felix Bachmann and Paul Clements. Variability in software product lines. Technical Report CMU/SEI-2005-TR-012, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2005.

[BK08] C. Baier and J.P. Katoen. *Principles of Model Checking.* MIT Press, 2008.

[BKS10] Daniel Bruns, Vladimir Klebanov, and Ina Schaefer. Verification of software product lines with delta-oriented slicing. In *Formal Verification of Object-Oriented Software. Papers presented at the International Conference, June 28-30, 2010, Paris, France*, pages 61–75. Karlsruher Institut für Technologie (KIT), 06 2010.

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.

[BTR$^+$13] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. Spl-lift: Statically analyzing software product lines in minutes instead of years. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 355–364. Association for Computing Machinery, 2013.

[CGK$^+$18] E.M. Clarke, O. Grumberg, D. Kroening, D. Peled, and H. Veith. *Model Checking, second edition.* Cyber Physical Systems Series. MIT Press, 2018.

[CLA$^+$18] Thiago Castro, André Lanna, Vander Alves, Leopoldo Teixeira, Sven Apel, and Pierre-Yves Schobbens. All roads lead to rome: Commuting strategies for product-line reliability analysis. *Science of Computer Programming*, 152:116–160, 2018.

[CN01] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 2001.

[CN12] Paul Clements and Linda Northrop. A framework for software product line practice, version 5.0. `https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=495357`, 2012. White Paper.

[CSHL12a] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. Behavioural modelling and verification of real-time software product lines. page 66–75. Association for Computing Machinery, 2012.

[CSHL12b] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. Towards an incremental automata-based approach for software product-line model checking. In *Proceedings of the 16th International Software Product Line Conference - Volume 2*, SPLC '12, page 74–81. Association for Computing Machinery, 2012.

[dCMMCd14] Ivan do Carmo Machado, John D. McGregor, Yguaratã Cerqueira Cavalcanti, and Eduardo Santana de Almeida. On strategies for testing software product lines: A systematic literature review. *Information and Software Technology*, 56(10):1183–1199, 2014.

[GBT$^+$19] José Galindo, David Benavides, Pablo Trinidad, Antonio Gutierrez, and Antonio Ruiz-Cortés. Automated analysis of feature models: Quo vadis? *Computing*, 101, 05 2019.

[GLS08] Alexander Gruler, Martin Leucker, and Kathrin Scheidemann. Modeling and model checking software product lines. In *Formal Methods for Open Object-Based Distributed Systems*, pages 113–131. Springer Berlin Heidelberg, 2008.

[HP04] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39:92–106, 2004.

[JB20] IntelliJ IDEA Blog Jet Brains. Explore your program with static analysis. `https://blog.jetbrains.com/idea/2020/10/explore-your-program-with-static-analysis/`, 2020. Accessed 27 Dec. 2021.

[Kat06] Shmuel Katz. Aspect categories and classes of temporal properties. *Lecture Notes in Computer Science*, 3880:106–134, 01 2006.

[KAuR$^+$09] Christian Kästner, Sven Apel, Syed Saif ur Rahman, Marko Rosenmüller, Don Batory, and Gunter Saake. On the impact of the optional feature problem: Analysis and case studies. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, page 181–190. Carnegie Mellon University, 2009.

[KBBK10] Chang Hwan Peter Kim, Eric Bodden, Don Batory, and Sarfraz Khurshid. Reducing configurations to monitor in a software product line. In *Runtime Verification*, pages 285–299. Springer Berlin Heidelberg, 2010.

[KBK11] Chang Hwan Peter Kim, Don S. Batory, and Sarfraz Khurshid. Reducing combinatorics in testing product lines. In *Proceedings of the Tenth International Conference on Aspect-Oriented Software Development*, page 57–68. Association for Computing Machinery, 2011.

[KCH$^+$90] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Software Engineering Institute, Carnegie Mellon University, 1990.

[KKL16] D. Kuhn, Raghu Kacker, and yu Lei. *Introduction to Combinatorial Testing.* Chapman and Hall/CRC Press, 04 2016.

[KS10] Martin Kuhlemann and Martin Sturm. Patching product line programs. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development, FOSD'10*, pages 33–40, 01 2010.

[KvRE$^+$12] Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. Toward variability-aware testing. page 1–8. Association for Computing Machinery, 2012.

[KvRHA13] Sergiy Kolesnikov, Alexander von Rhein, Claus Hunsen, and Sven Apel. A comparison of product-based, feature-based, and family-based type checking. *SIGPLAN Not.*, 49(3):115–124, Oct. 2013.

[LBL06] Jia Liu, Don Batory, and Christian Lengauer. Feature oriented refactoring of legacy applications. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, volume 2006, pages 112–121, 05 2006.

[LMP10] Kim Lauenroth, Andreas Metzger, and Klaus Pohl. Quality assurance in the presence of variability. In *Intentional Perspectives on Information Systems Engineering*, pages 319–333. Springer, 2010.

[LSR07] Frank Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering.* 01 2007.

[Mit02] John C. Mitchell. *Concepts in Programming Languages.* Cambridge University Press, 2002.

[MNJP02] J.D. McGregor, L.M. Northrop, S. Jarrad, and K. Pohl. Initiating software product lines. *IEEE Software*, 19(4):24–27, 2002.

[Muc98] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers Inc., 1998.

[MW] Online Dictionary Merriam-Webster. Definition of 'product line'. `https://www.merriam-webster.com/dictionary/product%20line`. Accessed 9 Dec. 2021.

[NNH04] Flemming Nielson, Hanne Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer Berlin Heidelberg, 2004.

[PBL05] Klaus Pohl, Günter Böckle, and Frank Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques.* Springer, 01 2005.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '95, page 49–61. Association for Computing Machinery, 1995.

[Sch01] J.M. Schumann. *Automated Theorem Proving in Software Engineering.* Springer Berlin Heidelberg, 2001.

[SRH96] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *Selected Papers from the 6th International Joint Conference on Theory and Practice of Software Development*, TAPSOFT '95, page 131–170. Elsevier Science Publishers B. V., 1996.

[TAK$^+$14] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, 47(1), jun 2014.

[Tea] SpotBugs Team. Spotbugs homepage. `https://spotbugs.github.io/`. Accessed 11 Dec. 2021.

[TLD$^+$11] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. Configuration coverage in the analysis of large-scale system software. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*, PLOS '11. Association for Computing Machinery, 2011.

[Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, page 439–449. IEEE Press, 1981.