SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2017; **00**:1–28
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/

Composing Annotations Without Regret? Practical Experiences Using FeatureC

Jacob Krüger[1,2*], Marcus Pinnecke[1], Andy Kenner[3], Christopher Kruczek[3], Fabian Benduhn[1], Thomas Leich[2,3], Gunter Saake[1]

[1] *Otto-von-Guericke University Magdeburg, Germany*
[2] *Harz University of Applied Sciences Wernigerode, Germany*
[3] *METOP GmbH Magdeburg, Germany*

SUMMARY

Software product lines enable developers to derive similar products from a common code base. Existing implementation techniques can be categorized as composition-based and annotation-based, with both approaches promising complementary benefits. However, annotation-based approaches are commonly used in practice despite composition allowing physical separation of features and, thus, improving traceability and maintenance. A main hindrance to migrate annotated systems towards a composition-based product line is the challenging and time consuming transformation task. For a company it is difficult to predict the corresponding costs, and a successful outcome is uncertain. To overcome such problems, a solution proposed by previous work is to use a hybrid approach, utilizing composition and annotation simultaneously. Based on this idea, we introduce a step-wise migration process from annotation-based towards composition-based approaches to lower the adoption barrier of composition. This process itself is independent of used implementation techniques and enables developers to incrementally migrate towards composition. We support our approach with detailed examples by partially migrating a real-world system. In detail, we describe *i)* our migration process, *ii)* its application on a real-world system, and *iii)* discuss practical challenges we faced. We implemented the proposed approach and show that appropriate tool support helps to migrate towards composition-based product lines. Based on the case study, we show that hybrid product lines work correctly and can compete with the performance of the original annotated system. However, the results also illustrate open issues that have to be solved to apply such migrations in practice.
Copyright © 2017 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: software product line; composition; annotation; migration; integration; refactoring

1. INTRODUCTION

Software product lines are a systematic reuse approach to create similar systems from a common base [1, 2]. Product lines are defined by their *features* that describe common and variable functionality, and can be implemented using several techniques with different pros and cons [3, 4]. In this article, we distinguish annotation-based and composition-based approaches [4, 5]. Implementing variability by annotating source code, for example with preprocessor directives, is often used in practice [4, 5, 6, 7]. Preprocessors provide a low effort and ad-hoc mechanism to add fine-grained adaptations. While this is an effective way to implement variability, code and feature traceability, as well as modularity are poorly supported or even unintended [5, 8, 9]. Also, type-checking [10] all possible configurations of a product line is challenging for annotation-based implementations [11, 12].

*Correspondence to: Faculty of Computer Science, Otto-von-Guericke University, Universitätsplatz 2, D-39106 Magdeburg, Germany. E-mail: jkrueger@ovgu.de

In contrast, composition-based approaches, such as feature-oriented programming (FOP), avoid those problems, by using physical separation of features [4, 5, 13, 14, 15]. With feature-oriented programming, each feature is encapsulated into a module and serves as a configuration option. Those modules are combined to generate a customized variant. Due to this physical separation, feature and code traceability are straightforward, which facilitates maintaining and extending the product line.

Despite these benefits composition-based approaches are rarely adopted in practice [4, 5, 16]. There are some reasons that hinder their application. For example, using composition is challenging and error-prone, and corresponding tools have to meet high requirements and are hard to integrate in existing development processes [17]. In contrast, annotation-based approaches are supported by established tools, for instance the C preprocessor [7], and allow ad-hoc changes without much preplanning effort [4]. Thus, they are widely used and accepted in practice. Finally, migrating legacy applications from annotation to composition is time-consuming and costly [15, 18]. For such reasons, annotation-based implementations are the dominant implementation approach in practice.

To overcome such problems, Kästner and Apel [5] introduce the idea to combine (*integrate*) annotation and composition into a *hybrid* approach. They envision to utilize advantages of both techniques, using step-wise migrations. However, they only discuss the characteristics of such a combination but do not investigate the actual migration from an annotated system towards composition. In an earlier paper, we built on their idea to develop a simple migration concept and provided small examples [17]. For this article, we refine and extend our approach considerably. We specify a full migration process and analyze possibilities to automate tasks, which reduces the adoption barrier [18, 19]. Furthermore, we present new and detailed insights into a partial migration of BERKELEY DB[†], a preprocessor-based database system, to review and assess our migration process. As a result, we present challenges and pitfalls, for example, undisciplined annotations and required tooling, that companies may face during such migrations. Overall, our approach is not limited to migrating product lines towards a composition-based or combined approach. Another application scenario is the extraction of a product line from variable stand-alone systems.

More detailed, we make the following contributions:

- We propose a migration process to integrate composition into annotation-based approaches. Besides technical concerns, we also address automation for each step.
- We provide detailed examples of our approach based on a migration of BERKELEY DB. Therefore, we introduce FEATUREC, an extension of FEATUREHOUSE [20, 21], suitable for feature-oriented programming with C and the C preprocessor.
- We identify and discuss technical, conceptual, and organizational challenges we faced during the migration of Berkeley DB. For instance, we provide insights into practical migrations of a large-scale product line and required efforts.

The remaining article is structured as follows. In Section 2 we introduce topics that are required for the understanding of this article. We describe our research approach in Section 3 to define the scope of our work. Afterwards in Section 4, we introduce FEATUREC that we use for our migration process, which we describe in Section 5. Within Section 6, we provide detailed information on the practical application of our process on BERKELEY DB. We discuss results and further experiences we gained in Section 7 and Section 8. Finally, we provide a brief overview on related work in Section 9 before we conclude in Section 10.

2. BACKGROUND

To implement software product lines, several implementation techniques exist [3, 4]. They can be separated into *annotation-based* and *composition-based* approaches [4, 5, 22]. In the following, we introduce both categories as we combine them in our work. Additionally, we provide background on *variability modelling* as a common concept to manage product lines.

[†]`http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html`, 07.09.2016

```java
public class Main{
 public static void main(String[]
     args){
  /*if[Hello]*/
  System.out.print("Hello");
  /*end[Hello]*/
  /*if[Beautiful]*/
  System.out.print(" beautiful");
  /*end[Beautiful]*/
  /*if[Wonderful]*/
  System.out.print(" wonderful");
  /*end[Wonderful]*/
  /*if[World]*/
  System.out.print(" world!");
  /*end[World]*/
 }
}
```

(a) Annotated source code.

```java
public class Main{
 public static void main(String[]
     args){
  System.out.print("Hello");
  System.out.print(" beautiful");
  System.out.print(" world!");
 }
}
```

(b) Preprocessed source code.

Figure 1. Annotation-based implementation using MUNGE. If the token `Wonderful` is not defined for the annotated code (Figure 1a), the encapsulated code fragment is removed in the preprocessed code (Figure 1b).

2.1. Annotation-Based Approaches

In practice, variability in software product lines is commonly enabled with annotations [4, 5, 6, 7]. Typically, this approach is associated with the C preprocessors' `#ifdef` statements, other techniques being, for example, *XVCL* [23] or *Spoon* [24]. To include or exclude code during compilation, the corresponding fragments are explicitly encapsulated by *conditional compilation conditions*. Conditional compilation is achieved by combining two techniques: *macro substitution* and *conditional inclusion*. We introduce both techniques in terms of the widely used C preprocessor [7, 25]:

- **Macro substitution**: A token along with a replacement text is defined (i.e., `#define <token> <replacement>`). During preprocessing, all occurrences of the `token` are substituted by its `replacement`. The replacing text may contain textual statements or can be empty.
- **Conditional inclusion**: Allows to include or exclude annotated source code depending on the evaluation outcome of an expression (i.e., `#if <expression> <code fragment> #endif`). If the `expression` is evaluated to zero, the encapsulated code is removed.

Many variations and patterns for the C preprocessor are known and applied to overcome limitation of the C programming language. For example, upper bounds for data type ranges depend on the computer architecture[‡] and can be adopted using conditional compilation. In the context of software product lines, an expression of a conditional inclusion normally asks for the existence of a token (i.e., `#ifdef <token>`). This technique is commonly used to implement variability within a system.

In Figure 1, we illustrate a basic `Hello World` example provided in FEATUREIDE [26]. There, we use the simplistic JAVA preprocessor MUNGE. The base code in Figure 1a is annotated within the comments to encapsulate variable behaviour. Selecting a valid set of features leads to preprocessed code from which all undesired variability is removed. For example, in Figure 1b, the feature `Wonderful` is not selected and, thus, not part of the instantiated variant.

To control a product line's variant space, which is implied by the existence or absence of a set of tokens, several techniques are used in practice. These range from conditional definition of tokens depending on the presence-condition of other tokens (e.g., `#ifndef <token1> #define`

[‡]ISO/IEC 9899:TC3, Section 5.2.4.2.1 *Sizes of integer types <limits.h>*, online available at `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf`, 06.10.2016

```java
// Base Module
public class Main{
 protected void print(){
   System.out.print("Hello");
 }
 public static void main(String[] args){
   new Main().print();
 }
}

// Feature Module Beautiful
public class Main{
 protected void print(){
   original();
   System.out.print(" beautiful");
 }
}

// Feature Module Wonderful
public class Main{
 protected void print(){
   original();
   System.out.print(" wonderful");
 }
}

// Feature Module World
public class Main{
 protected void print(){
   original();
   System.out.print(" World!");
 }
}
```

(a) Modularized source code.

```java
public class Main{

 private void print__wrappee__Hello(){
   System.out.print("Hello");
 }

 private void print__wrappee__Beautiful(){
   print__wrappee__Hello();
   System.out.print(" beautiful");
 }

 protected void print(){
   print__wrappee__Beautiful();
   System.out.print(" World!");
 }

 public static void main(String[] args){
   new Main().print();
 }

}
```

(b) Composed source code.

Figure 2. Composition-based implementation using FEATUREHOUSE. If `Wonderful` is selected as feature (Figure 1a), the method is added and called accordingly in the composed code (Figure 1b).

`<token2>`), over definition of tokens with compiler flags (e.g., `gcc -Dtoken`) and make files (e.g., `make`), to high level build systems (e.g., KBUILD).

The research community criticizes the concept of annotation-based approaches as they hinder *traceability* and *physical separation* of features [4, 5, 6, 7]. However, there is still ongoing discussion whether annotations make it more difficult to develop systems or not [8, 27, 28]. Several researchers suggest to restrict the power of conditional inclusion, leading to a more *disciplined usage* of preprocessors [27]. In addition, to overcome the limitations regarding feature traceability and physical separation of concerns, composition-based approaches were proposed. However, these do not allow fine-grained adaptations and require preplanning [4, 16].

2.2. *Composition-Based Approaches*

Composition is an established technique to merge software artefacts by composing their substructure [4, 21]. In contrast to annotation-based approaches, compositions are aware of the structure of the affected classes and methods. Hence, the source code to be modified is enriched with more semantics by *feature syntax trees*. A feature syntax tree is a generalized view on the structure of objects that contain information to cover the modularity of an artefact. It is defined by its set of nodes, where each node maps into the language-dependent structure and defines a syntactic category and name. For instance, a data type `struct S` in the C language might be represented by a node

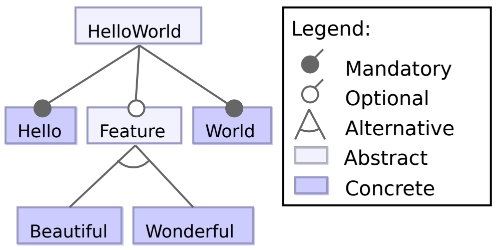


Figure 3. Basic feature model for the previous code examples.

called `S` with the type `struct`. This mapping inside feature syntax trees enables merging artefacts at structural level and, thus, *refining* base code (i.e., `S`).

We show a *`Hello World`* example from FEATUREIDE [26] for a composition-based approach (i.e., FEATUREHOUSE [20, 21]) in Figure 2. In contrast to Figure 1, we see that additional source code is necessary to represent context information. Each refinement in Figure 2a is placed in an own module, respectively class and method, with the same names. During composition, the features are merged into the base code. As we show in Figure 2b, the class is completely merged while the methods are only added and call each other.

Clearly, since syntax trees are independent of a certain language, due to their generalized view on structures, they can be applied for a variety of languages. In this article, we focus on FEATUREHOUSE [20, 21], an AHEAD-based [14] framework and tool chain for feature-oriented programming [13]. However, there are several alternatives to implement composition, for instance aspect-oriented [29] or delta-oriented programming [30]. While our basic example already uses FEATUREHOUSE, we will introduce this technique more detailed in Section 4.

2.3. *Variability Modelling*

Software product lines tend to cover a broad variability space of a domain. The complexity grows exponential, hence, for n optional and independent features 2^n possible configurations exist. To achieve reusability of features, quality guarantees, and to model the variability space (i.e., the dependencies of features), the research community introduced variability management using feature models [4, 31, 32]. Managing variability is divided into two tasks: *domain engineering*, and *application engineering* [4, 33].

Domain engineering includes defining a product line's variability and corresponding dependencies. For this purpose, several representations have been proposed or adapted, for example, feature models, decision models, delta models, UML, or natural language [34, 35, 36, 37, 38]. Despite this variety of representations, feature models and decision models are established in both, academia and industry [35, 36, 39]. Due to our own experiences and only minor differences between these two representations [37], we rely on feature models to describe variability in a product line.

A feature model is typically represented as a feature diagram [34]. We illustrate a basic example that corresponds with the intended variability in the previous code snippets in Figure 3. Feature models are important to define the variability and dependencies in a product line that is not represented in the source code. For example, in Figure 3 *`Beautiful`* and *`Wonderful`* are alternatives. Hence, only one of these two is intended to be selected for a product, while the source code itself does allow any combination. Other constrains might be parent-child relationships, optional, or mandatory selections of features inside a group, implications or exclusions. Besides the model itself, several code artefacts are developed during domain engineering. For instance, in the *`Hello World`* application the outputs *`Beautiful`* and *`Wonderful`* are modelled as two features. The associated code for these contains all code necessary to implement the refinement (compare with Figure 1 and Figure 2).

In application engineering, the variability space, implied by the feature model, is used to specify *variants*. A variant is a valid selection of features according to the constraints defined in the domain engineering. Since the dependencies in the feature model guarantee compatibility of several features selections, a variant can easily be generated. This generation process assembles corresponding artefacts to a fully functional system. How the generation actually proceeds, depends on the used implementation technique. For instance, a composer defines the tasks needed to assemble artifacts, and typically considers either runtime or compile-time variability. Runtime can be achieved by using plug-in systems or global configuration classes. In contrast, for compile-time composition-based or annotation-based approaches can be used.

A composer refines classes orthogonal to class inheritance of the object-oriented programming paradigm. State-of-the-art composers only consider either annotation or composition. In contrast, we propose a composer that can manage both simultaneously within the same code base.

3. RESEARCH APPROACH

In this section, we provide an overview on our research approach to scope the contribution of this article. Therefore, we define our *research goal and methodology*, define corresponding *evaluation criteria*, and describe the used *tools*.

3.1. Research Goal and Methodology

Overall, we aim to provide a process that supports the migration from annotation-based towards composition-based implementations. We especially aim to facilitate the practical application of this process to support practitioners in using composition. Hence, we focus on partly migrating a legacy system towards a hybrid system, integrating composition into annotation, to limit risks and potential pitfalls. More precisely, our research questions are as follows:

RQ-1 How can we migrate an annotation-based towards a composition-based implementation?
RQ-2 How does the resulting hybrid system perform compared to the original system?
RQ-3 Which open issues and potential pitfalls do exist?

To answer these research questions, we partly migrate BERKELEY DB from its annotated C version towards feature-oriented programming [13]. We focus on a partial migration to better reflect the practical application of our approach. It seems reasonable for a company to only migrate some features towards composition to limit costs, focus on regularly updated features, or because fine-grained [16, 40] variability may not be useful to migrate. Hence, in a hybrid approach three different sets of features can exist:

- Implemented only with annotations.
- Implemented solely with composition.
- Implemented with both, composition and annotations.

During our case study, we explicitly aim to cover these sets. Thus, we fully migrate some features towards feature-oriented programming and partly migrate some others, mainly to cover feature interactions. We emphasize that the scope of our work is not to fully migrate annotations towards composition but enable a step-wise and partial integration into a hybrid approach.

3.2. Evaluation Criteria

An evaluation is challenging to perform for our process. Still, we answer our first research questions based on the conducted case study. We report results on the migrated source code and argue that our process is suitable for step-wise migrations. This provides an overview on the process's applicability and open challenges.

To answer our second research question, we apply two evaluations similar to related work [41]: Firstly, we assess whether the binary size (a.k.a. footprint) of migrated configurations stay the same. Secondly, we use a test suite and a performance suite with several test runs provided with BERKELEY DB to evaluate the correctness and measure performance differences. In both cases, similar results should be achieved for the hybrid system, indicating that our migration is implemented correctly.

To answer our third research question, we discuss observations we made during our case study. We derive open issues based on problems we faced ourselves and that we were only partly able to

```java
@MethodObject
static class Txn_traceCommit{
 // [...]

 void execute(){
 }

 // [...]
}
```

(a) Base module.

```java
public class Txn{
 @MethodObject
 static class Txn_traceCommit{
  void execute(){
   logger=envImpl.getLogger();
   original();
  }
 }
}
```

(b) Feature module.

```java
@MethodObject
static class Txn_traceCommit{
 // [...]

 void execute(){
  logger=envImpl.getLogger();
  execute__wrappee__base();
 }

 // [...]
}
```

(c) Composed base module.

Figure 4. FEATUREHOUSE composition in the JAVA version of BERKELEY DB.

resolve. Hence, we provide starting points for further research, especially to improve the practical applicability of composition.

3.3. Tools

For our case study, we relied on existing tools, namely:

- ECLIPSE[§] as integrated development environment, extended with the C/C++ DEVELOPMENT TOOLING[¶] (CDT), for C development, and FEATUREIDE [17, 42, 43], for software product line development and variability modeling, plug-ins.
- FEATUREHOUSE [20, 21] as tool chain for feature-oriented programming and software composition.
- BERKELEYDB as the subject system for our case study and its test and performance suites for our evaluation.

In addition, we developed FEATUREC to enable FEATUREHOUSE to compose any annotated source code, as we describe in the next section.

4. FEATUREC

To combine annotation-based and composition-based approaches, we need a suitable composer. For our purpose, we developed FEATUREC, a feature-oriented extension of C that also supports preprocessor directives. We base our implementation on FEATUREHOUSE [20, 21], which provides a

[§]`https://eclipse.org/`, 06.09.2016
[¶]`https://eclipse.org/cdt/`, 07.09.2016

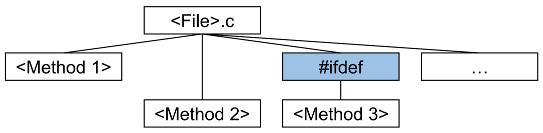


Figure 5. Depiction of a grammatical change (highlighted in blue) to support annotations in FEATUREC.

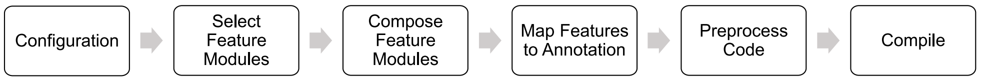


Figure 6. Instantiation of a variant in a combined annotation-based and composition-based product line.

tool chain to compose software. It can be applied on different programming languages by defining a suitable grammar. This grammar describes how the language is parsed and composed.

We illustrate a code example from BERKELEY DB with FEATUREHOUSE in Figure 4. In Figure 4a, an empty method within the base code is defined (`execute`). The *Txn* feature in Figure 4b, refines this method by applying a logging function. Calling `original` refers to the position at which the variability is added. In our example, the feature code comes first and only afterwards the original implementation follows. As we illustrate in Figure 4c, the previously empty `execute` is refined with two additional lines of code. The annotation (`@MethodObject`) specifies, how the artefacts are represented to the composer.

Applying FEATUREHOUSE directly on the C version of BERKELEY DB is possible. However, the standard grammar does not support all kinds of annotations and, thus, limits our possibilities. Therefore, we defined FEATUREC to overcome such shortcomings. To use composition and annotations in concert, enabling a *hybrid* product line, our grammar has to:

- specify how to parse the programming language,
- define composition rules, and
- also support annotations.

Parsing and composing are already included in FEATUREHOUSE but require adaptation for C. More challenging is the introduction of some preprocessor annotations. We are able to avoid some grammatical changes through source-code discipline [27, 28]. In C, only annotations on *a)* entire functions, *b)* type definitions, *c)* entire statements, or *d)* elements inside type definitions are considered to be disciplined [27]. Still, it is not possible to solve all problems this way. For example, annotations that encapsulate a complete method are problematic. During composition, such methods get lost and result in variants with missing source code. We display this grammar change in Figure 5. The composer must be aware of the possibility that annotations may indicate variability for a whole method. However, annotations within methods are unproblematic to compose.

Overall, we can create a customized variant for a hybrid approach using FEATUREC. All annotated code is located in specific feature modules. Then, we can configure, compile, and execute different systems. In Figure 6, we illustrate how we compose a specific variant. The first step is to select a valid set of features, the configuration, and provide it to the composer. Afterwards, FEATUREC selects the correct feature modules that are then composed. Before our composer can preprocess the remaining source code, it has to map each selected feature to its annotations. Thereafter, undesired annotations are removed and, finally, the code is compiled. Hence, the customized variant is created. While this process might be straightforward [5], it still provides challenges, for instance mapping old make files. We discuss this topic in Section 8.3. It is conceptually possible to switch selection and composition of feature modules with the preprocessing step. However, it could be possible that switching the instantiation steps will result in different variants, due to the three sets of features in a hybrid product line (see Section 3).

To assess the impacts of changes within the instantiation process, several aspects require further research. First, *complexity and effort* is important. Both may heavily depend on the project implementation and the degree of migration. For instance, assuming that the composition is only

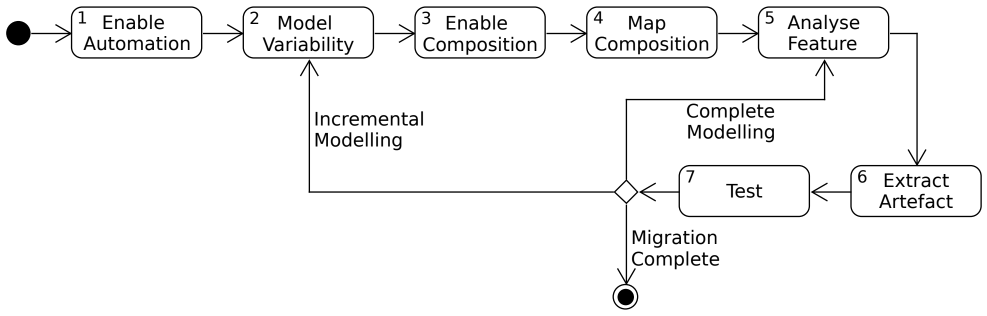


Figure 7. Activity diagram [47] of the step-wise migration process to integrate composition into an annotation-based product line.

applied on a coarse granularity and includes a lot of annotations, composing can exclude numerous preprocessor directives. Hence, preprocessing costs are reduced.

Second, *grammar changes*, as we applied them in FEATUREC, can be necessary. The corresponding adaptations depend on the used implementation techniques and also the order in which the instantiation is done. If we remove annotations first, we do not need to consider them in FEATUREC. However, changes on the preprocessor are necessary to consider feature modules.

Third, conceptually the order should not influence the *completeness* of the instantiated variants. Still, due to different implementation techniques and possibly required grammar changes, this cannot be ensured for all hybrid approaches. Further analyses and case studies on such approaches are required to assess adaptations.

Finally, these aspects can affect the main goal of refactorings not to change a system's behaviour [44, 45]. Thus, all migrations should be *minimal invasive*, causing as few changes in the source code as possible. To achieve this, we used the instantiation order depicted in Figure 6. Our BERKELEY DB version is implemented using the C preprocessor. The standard tool-chain for preprocessors combines processing and compilation of source code. Separating both is only done exceptionally for analyses. Also, the C preprocessor works correctly on composed code but may require changes to address feature modules. Hence, our instantiation process requires fewer changes and lowers the adoption barrier.

In this section, we introduced FEATUREC. It can compose variants of a hybrid software product line in the C programming language. We require FEATUREC's basic concept in our migration process and use it for our case study.

5. MIGRATION PROCESS

In this section, we introduce our migration process of annotated systems towards composition. As presented in previous papers, such processes are independent of concrete techniques [5, 17, 22], wherefore they can be adopted and applied for all of them. In this context and for the remaining article, we use the following terms:

- **Process** refers to our migration process, which follows the definition of Lonchamp [46] and, thus, is a set of steps that are used to maintain a software based on human and automated tasks.
- **Project** refers to an annotated legacy system and its corresponding migration process.
- **Feature** refers to a concern in a project, especially on conceptual level rather than in implementation.
- **Artefact** refers to a single variable code fragment of a feature and, thus, its implementation.

Hence, within a project one or more features are considered, which are implemented by one or more artifacts.

We illustrate the activity diagram [47] of our migration process, which consists of seven process steps [46], in Figure 7. As *input*, the process only requires an annotation-based legacy system but can be facilitated with additional information and artefacts, such as, documentation and tests. In detail, the provided legacy system is migrated with the following steps:

1. *Enable Automation*: Based on a project's characteristics, for example, the programming language and used composition mechanism, tool support is selected for each following step.
2. *Model Variability*: The variability of the system must be modelled to enable configuration for composition-based approaches. This can be done *incrementally* for each migrated feature or all at once (*complete*).
3. *Enable Composition:* The composition mechanism is introduced into the project.
4. *Map Composition*: Composition units (e.g., modules) are designed for features that are migrated and mapped to the model.
5. *Analyse Feature*: A feature in the legacy system is analysed to assess how it can be migrated.
6. *Extract Artefact*: An artefact of the analysed feature is migrated into a composition unit.
7. *Test*: The result of the migration is evaluated, which can be done after each or for a set of migrations.

The last three steps (5-7) are repeated to extract several artifacts and features of the product line. As described, it is also possible to repeat steps two to four if features are incrementally modelled and mapped as soon as they are extracted. This enables an incremental approach to focus on one feature at a time during the whole process.

As this indicates, the *output* of our process is not necessarily a fully migrated, composition-based product line (including a variability model). Instead, we enable a hybrid system, enabling annotations and composition at the same time, by following the idea of refactorings: improving source code without changing its behaviour [44, 45]. Hence, with respect to software product lines, we address *variability-preserving* migration and refactoring [4, 48]. Overall, we aim to facilitate maintenance and evolution of a company's systems.

The goal of our approach is to ensure a consistent system after each step. Thus, long development stops and additional costs are reduced. Both aspects are essential to integrate composition into existing software product lines [18]. While details on consistency unavoidably vary for concrete projects (e.g., depending on implementation techniques, languages, or tools) [49], we can ensure that the migration itself is consistent. In the following, we describe our migration process in detail.

5.1. Enable Automation

For minimal interruptions, it is necessary that the system can be compiled, executed, and tested after each step. Thus, the first step of our process is to select suitable tools, which enables automation for instantiating variants. An integrated development environment (IDE) helps during the whole process. It has to support annotation-based and composition-based software development at the same time. As in many IDEs, such as ECLIPSE, this includes building, executing, and testing programs that utilize both approaches. Most of these steps must be adapted for different implementation techniques and especially their combinations. Hence, support for the used implementation techniques is the most influential factor.

Other tasks in our migration concept can be further automated. Thus, we can categorize tools by the process steps they support and discuss them then. Still, every automation influences the IDE decision or may require adaptations. As a result, a company has to plan its migration to avoid later tool changes.

In contrast to further steps, selecting the right tools can hardly be automated. It requires manual analysis and selections based on the project and included tasks. This might be costly for some approaches that need further adaptations. Even though, it must only be done once for a project.

5.2. Model Variability

After tool support is selected, it is necessary to determine and model the variability within the system. In particular, systematic variability management must be introduced to customize the software. For an existing product line, it might be the case that such management already exists. Otherwise, more detailed analyses are necessary. Ideally, the variability of the program is modelled and mapped to corresponding code artefacts. Thus, developers can easier find and analyse variable code of the system in later steps. In contrast, it is also possible to reduce modelling and mapping to a minimum. We could only model features, which we will extract later, ad-hoc to reduce efforts, similar to step-wise refactorings. While this strategy may reduce the efforts during this step, others will be more challenging and modelling may not be consistent.

Variability management can be supported with several tools. Some IDEs, such as FEATUREIDE [26], GEARS [50], or PURE::VARIANTS [51], already include modelling and automated configuring. However, it must be ensured that both tasks are available for annotations and compositions in parallel. Analysing variability in a legacy system can also be supported [52], for instance with LEADT [53] or feature location tools [54, 55]. Especially in the presence of preprocessor statements or based on additional artifacts, automatically extracting parts of feature models is possible [56, 57, 58, 59]. Nonetheless, *variability mining* cannot be done fully automatic but requires manual work [60, 61]. Therefore, this analysis may require a lot of effort, time, and domain knowledge. Still, each feature must only be added once to the variability model for a project. Updating the model afterwards to add previously unrepresented dependencies is relatively simple and several refactorings and validity checks have been proposed [52].

5.3. *Enable Composition*

During this step, composition is integrated into the system. The result is a trivial decomposition of all code into a single module. For example, using feature-oriented programming we initialize the whole project with one base feature that includes all annotations. Thus, there is still no physical separation of variability but the required implementation technique is enabled. While it is the first change on the concrete implementation, variants can still be instantiated as before. Important is to ensure that the composer's grammar is able to compose the code correctly. For instance, this may require disciplining annotations for syntactical correctness and addressing correct language versions.

This whole step can be semi-automated. Therefore, a tool needs to provide an import functionality that supports developers. Still, even manual integration does not require much time or effort. The goal is to ensure that the composition is working without errors. This may require additional changes in the source code, such as disciplining annotations, that must be done manually. For instance, in our case study we had to implement FEATUREC. However, this step is only required once during a migration project.

5.4. *Map Composition*

While composition is available after the previous step, we still have to map implementation and variability management. Only this way, correct configuration and instantiation is ensured. Therefore, modules for features are defined on implementation level. With feature-oriented programming, each module is a file that will later contain a feature's code. These files have the same name as the file they refine to enable correct composition. Afterwards, each module is stored into a folder with the name of the corresponding feature (i.e., the previous preprocessor token). This maps the modules to a feature and the feature model. Extracted artefacts are later added to the corresponding modules. This way, a valid configuration can be selected in a tool and provided to the composer.

Identical to and depending on the variability modelling, this step can be done all at once or step-wise. Hence, in Figure 7 this step might be repeated for each feature. Mapping the composition-based feature modules requires manual effort. Developers have to create an according project structure and connect it to the configuration. However, as this step mainly requires to set up additional files and folders, it does not require much effort. Semi-automated support is possible by implementing a functionality that automatically links new feature modules to selected artefacts.

5.5. *Analyse Feature*

After the previous steps are done, the actual migration can start. The first task is to identify and analyse code artefacts that belong to a feature and shall be extracted. This is necessary to decide which artefacts to migrate. For instance, it is most likely not useful to extract a single line of variable

Table I. Overview of our step-wise migration process.

	Step	Iteration	Automation	Tool examples
1	Enable automation	P	Manual	None
2	Model variability	P/F	Semi	Modelling, feature location
3	Enable composition	P	Semi	Import functionality
4	Map composition	P/F	Semi	Variability management
5	Analyse feature	A	Semi	Highlighting, feature location
6	Extract artefact	A	Semi	Refactorings
7	Test	A	Semi	Unit test, configuration

P: Once per project, F: Once per feature, A: Once per artefact

code. At this point it seems more useful for a company to benefit from the combination of annotation and composition, and keep the code as it is. Afterwards, refactoring and restructuring the source code is planned. In particular, feature interactions must be addressed. These can require additional code changes and feature implementations.

It will most likely never be possible for a program to automatically identify all feature code [60, 61]. However, tools and IDEs to highlight and map annotations that belong to a specific feature exist [26, 62]. In addition, feature location [55] or variability mining [53, 61] techniques also support identification, analysis, and interpretation of artefacts. Despite such tools, the final migration plan must be designed by a developer. Analyses have to be done for each artefact of a feature that shall be extracted into a module.

5.6. Extract Artefact

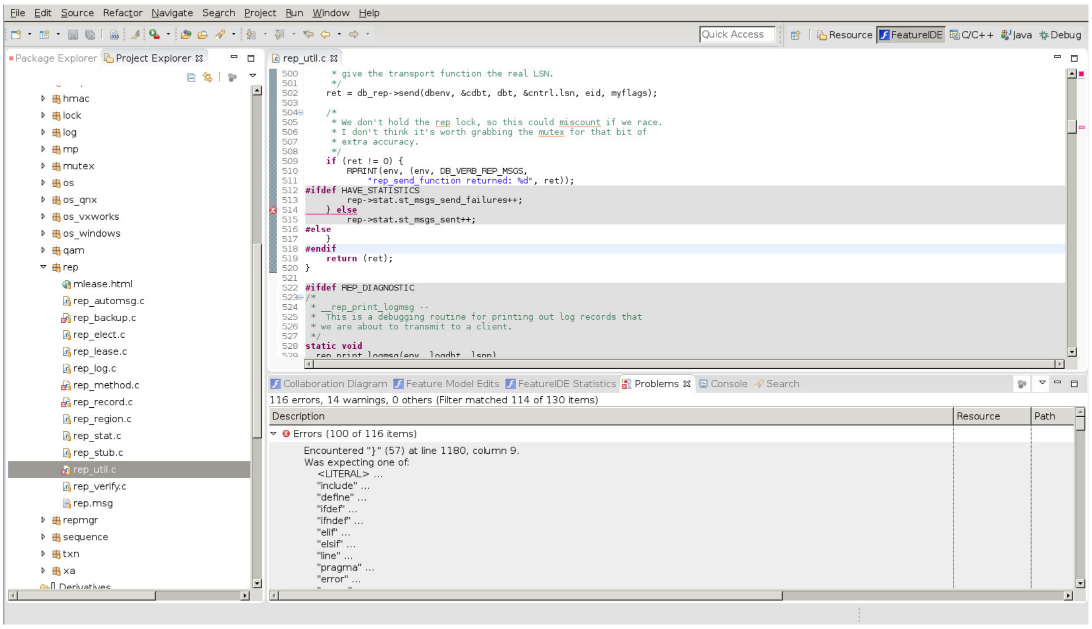
This step includes the actual migration of an annotated code fragment into a compositional module. Thus, the previously designed migration plan is executed. A first task is to refactor and improve the code to ease modularization. Removing design flaws, so called *code smells* [44, 63, 64], and disciplining the usage of preprocessors can be useful to reduce error-proness [65]. The second task is the migration of the source code artefact. Afterwards, a new compositional module exists and separates the feature from the base implementation. This module is linked to its original position based on the used composition mechanism.

It is possible to support the identification of error-prone feature code with metrics [66]. However, the actual migration must mainly be done manual. There are IDEs that can support these tasks with some simple mechanisms [67] or support refactorings, and some approaches for automatic modularization exist [22]. However, a developer has to use and assess the tools, and implement additional or adapted code. The whole step is repeated for each extracted artefact.

5.7. Test

With the previous step, the extraction of a single feature fragment is finalized. After each migration, the system shall be in a consistent and executable state. Still, the behaviour of the system must be tested and evaluated. Thus, developers can assess that the product line is working and can be configured correctly. This might be done after each migration, for instance if additional refactorings were needed, but for small and simple extractions several tests might be consolidated. Other approaches, to test product lines or software in general, can be used to further ensure the system's quality. Some examples are code inspections, reviews, or unit tests to assess migrations [68, 69].

For testing software product lines several approaches exist [70]. More and more of these can be executed automatically, test multiple aspects of a system, and are integrated into IDEs. However, testing the changed behaviour requires manual effort to identify and analyse suitable test cases. The effort of these tasks highly depends on the size and complexity of the extraction. Testing correct instantiation can be done with each IDE that supports the combined approach. Hence, it requires only few effort and might be automated. Still, developers have to manually assess the results and remove found errors.



```
	 * give the transport function the real LSN.
	 */
	ret = db_rep->send(dbenv, &cdbt, dbt, &cntrl.lsn, eid, myflags);

	/*
	 * We don't hold the rep lock, so this could miscount if we race.
	 * I don't think it's worth grabbing the mutex for that bit of
	 * extra accuracy.
	 */
	if (ret != 0) {
		RPRINT(env, (env, DB_VERB_REP_MSGS,
		    "rep_send_function returned: %d", ret));
#ifdef HAVE_STATISTICS
		rep->stat.st_msgs_send_failures++;
	} else
		rep->stat.st_msgs_sent++;
#else
	}
#endif
	return (ret);
}

#ifdef REP_DIAGNOSTIC
/*
 * __rep_print_logmsg --
 *  This is a debugging routine for printing out log records that
 * we are about to transmit to a client.
 */
static void
```

```
116 errors, 14 warnings, 0 others (Filter matched 114 of 130 items)
Errors (100 of 116 items)
    Encountered "}" (57) at line 1180, column 9.
    Was expecting one of:
        <LITERAL> ...
        "include" ...
        "define" ...
        "ifdef" ...
        "ifndef" ...
        "elif" ...
        "elsif" ...
        "line" ...
        "pragma" ...
        "error" ...
```

Figure 8. Snippet of BERKELEY DB after import into ECLIPSE CDT.

5.8. *Summary*

In this section, we introduced our migration process. We summarize each step in Table I, considering required iterations, automation degree, and examples for possible tool support. Modelling variability and mapping the resulting model to modules can be done for the whole project at once or only for features that are currently migrated. In contrast, the actual analysis and migration of variable code must be done for each artefact. While tool support exists for most of the steps to some extent, all of them still require some manual effort. During each task it can be necessary or helpful to go back and review gained results. We address this point in Section 8.1.

Overall, it is possible to implement composition without refactoring the whole product line at once. As the system is in a consistent state after each migration, companies do not need to stop production. This can significantly lower the adoption barrier [18]. Still, currently our process's manual effort limits its applicability to single features and smaller systems.

6. PRACTICAL APPLICATION

To this point, we introduced FEATUREC, a composer that can handle annotations, and our migration process. In this section, we illustrate our process on a real-world system. For each step, we describe detailed examples and challenges we found in our practical application. Our example is based on the industrial BERKELEY DB, an embedded database management system. The used version is implemented in C and includes preprocessor annotations to define variability. Overall, our BERKELEY DB system contains 229,419 lines of code. Hence, it is a relevant case study for practice, involving high challenges of annotations-based implementations. Also, BERKELEY DB is used and analysed in several other case studies in product-line research [22, 40, 41, 71, 72]. Our goal is to partly migrate the annotated system towards a compositional implementation, using feature-oriented programming with FEATUREC.

6.1. *Enable Automation*

The initial step for our migration is to select and adopt suitable tools. Mainly, we had to address two tasks:

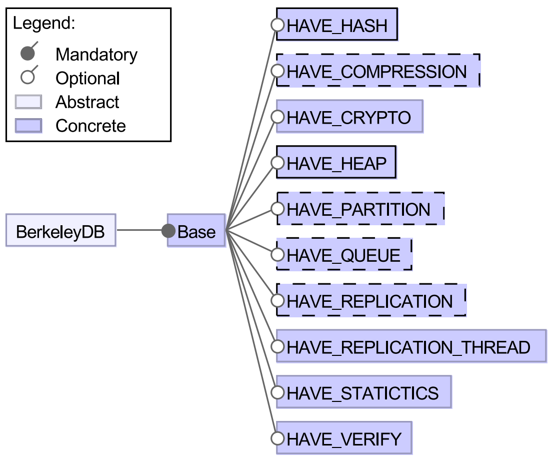


Figure 9. Feature model of BERKELEY DB. Features we fully migrated towards feature-oriented programming are framed with solid black. Dotted black frames indicate features that we partly migrated.

- Selecting an IDE that supports both composition-based and annotation-based approaches.
- Importing and integrating BERKELEY DB as project into the IDE.

There exist several IDEs that support product line development. For this case study, we build on our previous work, FEATUREIDE, an ECLIPSE plug-in for feature-oriented software development [17, 42, 43]. It includes all important tasks and techniques that we require for our migration. For example, FEATUREIDE supports modelling as well as annotation-based and composition-based approaches. Still, we needed to combine annotation-based and composition-based implementations, wherefore we integrated FEATUREC into FEATUREIDE.

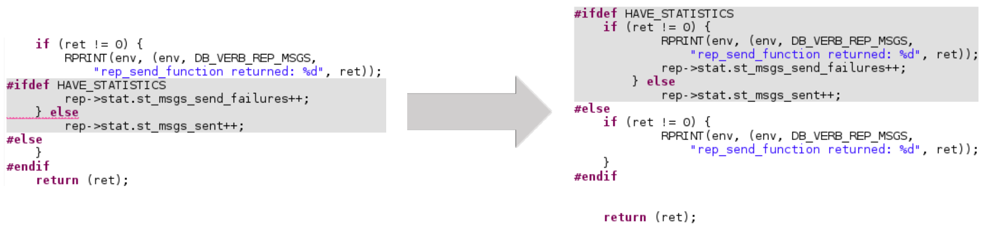
To import BERKELEY DB, we used the C/C++ DEVELOPMENT TOOLING (CDT). It is another plug-in that supports C and its preprocessor in ECLIPSE. In FEATUREIDE, we created a new C project and imported all source and additional files of the database system. Then, we adapted BERKELEY DB's settings and implementation to ensure correct behaviour. For our case, some errors occurred because of deviations from the C standard or deprecated statements. We show a screenshot of our imported BERKELEY DB project in Figure 8. The explorer and console illustrate 116 found errors. However, many of those are caused by syntax rules of the FEATUREC grammar, which we address in step three. Such problems hinder the usage of composition for now but we can still instantiate the system with the C preprocessor.

This step required several hours. Still, the effort strongly depends on the used IDE, its support for an integrated approach, and the complexity of the project. While we needed a lot of manual work, we only had to do it once for the project. Also, with the development of integrated approaches and tools, the effort of this step may decrease. Overall, it is essential to use an IDE that supports all implementation and instantiation tasks for a company's project.

6.2. *Model Variability*

The next step in our process is to introduce variability modelling. Therefore, we first created a feature model with a single base module. Afterwards, we expanded this model and introduced the features of BERKELEY DB. We illustrate the resulting variability in Figure 9. For BERKELEY DB, we were able to reuse analysis and information of previous research. Based on this, we identify the 10 preprocessor variables shown in Figure 9 to represent features. Each of these provides optional variability, which can be combined freely. During our case study, we completely migrated the features `Hash` and `Heap`. In addition, we analysed and extracted parts of other features, which was mainly did due to interactions. Thus, we partly migrated `Compression`, `Partition`, `Queue`, and `Replication`. However, in this article we exemplary focus on `Hash` and `Heap`.

For our example, the effort of modelling variability was minimal due to existing research. Other projects may require a lot of analysis before it can be described with a feature model. Due to the



```
    if (ret != 0) {
        RPRINT(env, (env, DB_VERB_REP_MSGS,
            "rep_send_function returned: %d", ret));
#ifdef HAVE_STATISTICS
        rep->stat.st_msgs_send_failures++;
    } else
        rep->stat.st_msgs_sent++;
#else
    }
#endif
    return (ret);
```

```
#ifdef HAVE_STATISTICS
    if (ret != 0) {
        RPRINT(env, (env, DB_VERB_REP_MSGS,
            "rep_send_function returned: %d", ret));
        rep->stat.st_msgs_send_failures++;
    } else
        rep->stat.st_msgs_sent++;
#else
    if (ret != 0) {
        RPRINT(env, (env, DB_VERB_REP_MSGS,
            "rep_send_function returned: %d", ret));
    }
#endif


    return (ret);
```

Figure 10. The undisciplined code (left) from Figure 8 is refactored according to the grammar provided by FEATUREC (right).

annotation-based implementation it seems feasible to use semi-automated and partial analysis to ease this step. For instance, feature location [54, 55] and variability mining [53, 61] are adoptable. Appropriate approaches may identify annotations and corresponding source code. This way, suitable features for migration could be identified with less effort.

6.3. *Enable Composition*

To this point, we imported BERKELEY DB into ECLIPSE using the FEATUREIDE and CDT plug-ins. In this step, we migrated its base code into a module to enable composition. Afterwards, the whole implementation is encapsulated within a single base feature. With our adopted FEATUREC grammar, we are able to instantiate the implementation by selecting this base module. Then, we use the C preprocessor to customize the annotations.

However, to use FEATUREC we first had to refactor the BERKELEY DB implementation. As described in Section 6.1, FEATUREIDE reported many errors that are caused by conflicts with our defined grammar. Mainly, this was due to undisciplined use of annotations. In Figure 10 we illustrate an example in which we refactored such an error. The original implementation on the left side has two closing brackets that are part of different annotations. Currently, FEATUREC does not support such constructs. Instead, we migrate the implementation to an equivalent but disciplined form. On the right side of Figure 10, opening and closing brackets are equal within each annotated code fragment. This disciplined style is supported by FEATUREC.

During our case study we found that such refactorings can be challenging tasks. They are error-prone, we deviate from the original implementation, and need code clones to address feature interactions. For instance, in Figure 10 some code is duplicated after disciplining. Most of the time we spent to manually adapt the implementation to suit the compositional grammar. However, these tasks are mostly required because no suitable tooling is available. Hence, semi-automatic tool support can ease such refactorings. Alternatively, we could further adopt FEATUREC to accept fine-grained adaptations. Still, disciplined annotations are often considered to improve source code understanding and analysis tasks [27, 28]. Thus, we decided to restructure the implementation rather than adopting our grammar furthermore, which is an error-prone task.

An interesting fact we found is that the overall effort for this step is less dependent on the size or number of features. More impact have the language and tools that are used to integrate composition. In our case, we had to refine the FEATUREHOUSE grammar. Nevertheless, we could not keep all annotations as they were but had to discipline them. Additionally, differences in supported language versions must be resolved. These two aspects caused the most effort for us while enabling composition. Using a grammar that supports undisciplined annotations and the identical programming language can ease this step.

6.4. *Map Composition*

Until now, we have enabled composition within a single base module. However, variable code is not physically separated, yet. During this step, we introduce further modules for each feature we aim to extract. Therefore, we only have to create according folders and map them to the feature model and, thus, variability management. As a result, FEATUREIDE is able to automate configuration,

```
#ifdef HAVE_HASH
 if (dbp->type == DB_HASH)
   __ham_copy_config(dbp, part_db, part->nparts);
#endif
```

Figure 11. Snippet of annotated BERKELEY DB code.

```
// Call hook method
__hook_HAVE_HASH_3(dbp,part_db,part);
// #ifdef HAVE_HASH [...]

// Definition of hook-method in feature module
void __hook_HAVE_HASH_3(DB *dbp,DB *part_db,DB_PARTITION *part) {
 if (dbp->type == DB_HASH)
   __ham_copy_config(dbp, part_db, part->nparts);
}
```

Figure 12. Migrated feature module of the feature `Hash` from Figure 11.

generation, and execution, for each migrated feature. A challenging aspect of this task can be the mapping of existing build or make processes. Companies still want to customize their legacy products and, thus, migration of these processes is necessary. We further address this challenge in Section 8.3.

While we did this step manually, the mapping of modules to variability is a simple task. More challenging is the migration of existing build processes. For our case study, we manually migrated predefined configurations and generations into FEATUREIDE. However, the whole step took us only few time. We could easily map the feature model, which we defined in step two, towards the corresponding folders. Also, configurations can easily be defined in FEATUREIDE and, thus, are well supported.

6.5. Analyse Feature

At this point, we started to identify and analyse feature artefacts in BERKELEY DB. Therefore, we searched for annotations that belong to the features `Hash` and `Heap`. In Figure 11, we show an example of the original implementation. There, `Hash` adds a conditional with a single line of executed code. While it is only a small part, we identified it to be easy to extract and modularize into composition. We planned to define a hook method, which encapsulates the previously annotated code. In the base code, we completely replace the variable part with a method call to the newly defined hook. Basically, hook methods are empty methods within the base code that are later refined [72, 73]. Although they may not be best practice [72, 74], they provide a common *extract-method refactoring* to physically separate variability [4, 73]. With hooks, we can modularize parts of features at any point of a method and, thus, apply and ease a step-wise migration. Still, we used hooks only sparsely. In most cases, the annotation discipline of FEATUREC encapsulated whole methods. Hence, we can migrate these artefacts into modules defined by FEATUREHOUSE to refine whole methods.

The effort for such analyses varies significantly. Simple cases as the one in Figure 11 are straightforward. However, there are more complex situations and feature interactions that require more detailed assessments. This can hardly be automated and is a challenging task, especially in the context of scope-sensitive statements, which we discuss in Section 8.4. Besides the analysis, locating all relevant feature artefacts and interactions is challenging. This cannot be fully automated [60, 61]. However, there are some methods that can provide assistance, for example colourization [9, 62] or searching for annotations. Also, we can consider only parts of a feature to ease this step and reduce risks and costs.

6.6. *Extract Artefact*

After a suitable code artefact is identified, we extract it towards composition. Therefore, we either refine methods, as intended by FEATUREHOUSE, or migrate variable code into a hook method. Such a method encapsulates the previously annotated implementation. To enable feature-oriented programming, we add a corresponding call at the position of the extracted variability. We illustrate the migrated code of Figure 11 in Figure 12. Line 2 implements the call of the hook method. It is placed at the same position as the originally annotated code, which are commented in lines 3 to 6. The physically extracted module is implemented from line 9 to 12 and is stored in the corresponding *Hash* directory defined in step 4 (see Section 6.4).

We migrated all features manually, wherefore all extractions were time consuming and also error-prone. However, migrating a single artefact is unproblematic in most cases. There are some approaches that address such refactorings [48] and aim to ease or partly automate this task. For instance, Liebig et al. [75] introduce MORPHEUS, a tool for automated refactorings of C code that can handle annotations. Kästner et al. [22] describe concepts to automate extractions. They also focus on the migration of annotation-based towards composition-based approaches. Similar to our process, they require disciplined annotations. Their concepts may provide help for our case study but must be adapted from LIGHTWEIGHT JAVA to C. Other approaches investigate the transformation of preprocessor annotations towards aspect-oriented programming [76, 77, 78]. While our case study uses feature-oriented programming, our general process can still be applied. All these approaches require further analyses but provide potential to integrate them into our concept. This way, a tool chain with defined tasks could be established.

6.7. *Test*

We tested the migrated BERKELEY DB on several occasions. To do this, we summarized simple migrations to test cases, reducing necessary time and effort. For instance, we did multiple extractions like the one shown in Figure 12. These are straight-forward and require only small changes. After we did some of such migrations for a specific feature, we automatically tested whether the configuration worked. In addition, we used unit tests [68, 69] to assess the behaviour of our adaptations. Still, there are more challenging migrations, which we describe in Section 8.4. They often required several adaptations or the introduction of new modules with cloned code. Thus, errors are more likely to happen. We evaluated such changes immediately after migration to ensure consistency.

There exist several approaches that can help to automate the testing of a migration [43, 68, 70, 79]. We used automated instantiations of FEATUREIDE and unit tests. Still, we needed some time and effort, mainly to fix bugs and assess code manually. Overall, the number of errors was small and mostly due to wrong syntax or missing parameters in method calls. This might be an indicator that our step-wise migration provides a good way to avoid larger bugs during migrations. We focused on a single code artefact at a time and did not change the behaviour of the system. Hence, if carefully analysed, the potential for errors is small. Instead, if we would refactor a whole and scattered feature all at once, it is more challenging, increasing the threat of introducing errors.

6.8. *Summary*

In this section, we described the application of our step-wise migration concept on a real-world system. We illustrated all steps of our process and described possible tool support and efforts. Overall, we found our approach to ease the integration of composition into an annotated system. Within the next section, we evaluate the resulting system of this case study. Afterwards, we discuss experiences we gained during the migration in Section 8.

7. RESULTS

BERKELEY DB is a productive embedded database written in C and uses preprocessor directives to enable variability. We fully migrated two of its features from annotation towards composition

Table II. Performance and footprints of BERKELEYDB variants.

Configuration	**Original BERKELEY DB**		**Migrated BERKELEY DB**		**Differences**	
	Time in sec.	Footprint in MB	Time in sec.	Footprint in MB	Δ Time	Δ Footprint
Base	16.4	7.37	16.9	7.35	+0.5	-0.02
`Partition`	17.2	7.37	16.9	7.35	-0.3	-0.02
`Hash`	16.7	7.41	16.8	7.39	+0.1	-0.02
`Hash, Heap`	16.6	7.42	16.1	7.43	-0.5	+0.01
`Hash, Partition`	16.5	7.41	16.3	7.39	-0.2	-0.02
`Heap`	17.1	7.38	16.6	7.39	-0.5	+0.01
`Heap, Partition`	16.3	7.38	16.6	7.39	+0.3	+0.01

without changing their behaviour and partly migrated four other features. Summarized, we extracted 7,146 out of 16,680 lines of code (42.8 %) from the ten defined features. This way, we developed a hybrid product line containing all three sets of features that are possible (see Section 3). While the number of migrated lines of code is relatively small in comparison to the full size of BERKELEY DB, we remark that we especially focused on this hybrid approach and already gained detailed insights. Overall, we found some points that reason for integrated approaches and the usability of our process.

Firstly, we reduced the overall size by 396 lines, which is approximately 6% of the extracted code. The code base can be further reduced by improved extraction concepts. We often had to extract methods multiple times to address feature interactions, as we discuss in Section 8.4. Still, due to reusing migrated parts, the code sized decreased.

Secondly, we modularize and gather scattered source code. For instance, `Hash` affects 12 and `Heap` 9 files within BERKELEY DB. In contrast to sole annotations, their variability can now be assessed within a single file. Hence, we can achieve physical separation of concerns if a feature is suitable for it. Otherwise, developers can still use annotations within the base code.

Finally, we rarely had to use hook methods. Within `Hash`, only three situations required to use them. Due to disciplined annotations, the designated composition mechanism of FEATUREHOUSE could be used most of the time. Hence, whole methods refine the base implementation and only few further adaptations are necessary.

To assess our second research question, we used a test suite that is provided together with BERKELEY DB and contains several test cases. We generated different variants of BERKELEY DB for the original and migrated system. For each variant, the test suite contained 19 test cases if `Partition` was deselected and 25 otherwise. No test that we could successfully run on the original BERKELEY DB failed in the migrated system. Hence, our migration is correct as far as the test cases cover the features' behaviour.

In Table II, we present the times a performance suite, which is provided by ORACLE, needed to run for 7 different variants as well as the variants' footprints. We remark that this suite is implemented for an older version and, thus, we first migrated the seven performance tests to the version of our BERKELEY DB system. These tests cover, for instance, read, write, and bulk operations, each using 1,000,000 key-value entries. Considering the performance, we repeated each run 10 times for each variant and present the average time reported by the suite. While the tests require more time for some migrated variants, they are faster for others.

The same accounts for the binary size of the migrated BERKELEY DB. There are only minor differences compared to the original system, which might be due to different optimizations of the compiler. Overall, we find no significant changes in the execution times and binary footprints. Thus, for our second research question, we argue that variants in a hybrid product line can perform equally to annotated systems.

8. EXPERIENCES AND DISCUSSION

In this article, we proposed a process to integrate composition into an annotation-based product line. We illustrated our step-wise and consistent migration on a real world system. During this case study, we found some additional challenges that we discuss in the following. More detailed, we address, the *interdependence of process steps*, *undisciplined annotations*, *preservation of legacy configurations*, *scope-sensitive statements*, and our process's *practical implementation*.

8.1. Interdependence of Process Steps

Our migration process contains several steps that build on each other. However, a company is not forced to perform them all as atomic tasks. Thus, it is not necessary to fully migrate a whole feature towards composition at once. This avoids parallelism, for example keeping a productive and a development system, as well as delays, costs, and risks. Overall, our proposed approach can lower the adoption barrier [18]. Still, there exist interdependencies between the steps that require further analyses and support.

During the migration of BERKELEY DB we found that revisions of previous steps might be necessary. For example, after the first refactoring, we identified some tool-driven and conceptual challenges that resulted in changes in previous steps. This led to further adaptations on FEATUREC to better integrate composition and enabling physical separation and mapping. Mostly, these revisions resulted in slight changes in the process. During further reviews and practical applications additional improvements and details might be identified.

8.2. Undisciplined Annotations

We did our first migrations using only existing tools. However, we found that language support was not sufficient. More precise, composition-based mechanisms for feature-oriented programming (e.g., FEATUREHOUSE) do not support all kinds of annotations. To overcome this problem, we introduced FEATUREC to provide a suitable grammar. Additionally, we were able to avoid several changes by disciplining preprocessor annotations.

We partly addressed discipline during our case study while enabling composition. This is not the only possible solution to suit a compositional grammar. New concepts or adaptations can improve FEATUREC. For instance, we may enable support for fine-grained annotations. Still, disciplining the usage of preprocessors promises to improve the development process [27, 28]. Hence, this may still be a suitable approach. The alternative of improving concepts for parallel composition-based and annotation-based implementations is challenging. We would not be able to use existing tools without significant adaptations. In detail, introducing an approach that can handle all cases of preprocessor statements and also composition is difficult. For this reason, we choose to refactor the source code with the desired discipline of annotations.

8.3. Preservation of Legacy Configurations

Building a system from its sources is a non-uniform process and, thus, includes adapted concepts and tasks. Developing variable software systems uses certain configurations before compilation and build processes. These steps vary heavily in terms of shape, language support, tools, and degree of automation. To utilize a hybrid annotation-based and composition-based application, we must ensure that it is possible to built legacy products as before, although composition is introduced. The challenge is to cope the changed configuration space, adapted by composition-based techniques, to existing configurations that only target annotated code.

Configuring a code base defines how and which source parts are modified before compilation. In Section 2.1 we introduced annotation-based approaches and their major techniques to enable variability. The question is how to cope the configuration space of used symbols in order to express valid and desired products. Therefore, the build process is used. Building a code base describes how to construct a desired artefact. It is influenced by several aspects of software development, such as:

- *Programming language*: Different languages require adopted build processes. For instance, compiling a C file is different from the interpretation of PYTHON code.
- *Runtime environment*: Depending on the runtime environment, the support for build processes changes. For example, conditional compilation for native code of LINUX and managed code of .NET are significantly different.

```
1  switch (new_dbc -> dbtype) {
2   case DB_BTREE:
3    // [...]
4    break;
5  #ifdef HAVE_HASH
6   case DB_HASH:
7    if ((ret = __ham_stat (new_dbc, &hsp, flags)) != 0)
8     goto err;
9    // [...]
10   break;
11 #endif
12  default:
13   break;
14 }
```

Figure 13. Example of a problematic `switch` environment in BERKELEY DB.

- *Code homogeneity*: While our case study uses the pure C implementation of BERKELEY DB, there are also projects that combine several programming languages. Thus, they may also use different annotation styles and build processes that must be considered.
- *Company constraints*: A company can enforce guidelines or constraints. For example, it may use tailor-made build tools that are not uniform with the standard and, hence, require further adaptations.

For such reasons, build processes range from direct compiler invoking, over multi-step processes, to cross-language compilation. These processes must be adapted or mapped when a composition-based approach is integrated on top of an existing implementation. This can be challenging, depending on existing tooling and used techniques. In our case study, we manually migrated existing configurations of BERKELEY DB towards the introduced composition. As a result, we could use FEATUREC to invoke the same variants as before. However, we thereby made existing configuration and build processes useless for the migrated product line. For a company it may be problematic that such legacy builds cannot be used. However, we argue that a suitable tooling to support the mapping towards composition can facilitate this task.

8.4. Scope-Sensitive Statements

During our practical application, we often tried to use hook methods to enable fine-grained feature-oriented programming in BERKELEY DB. These would allow us to extract small parts of variability and later include them again at defined positions. However, we faced several problems that forced us to extract whole methods as desired by FEATUREHOUSE. While this might be a more disciplined and understandable solution, it also results in code clones. We experienced that statements that are depending on their current *scope* can be problematic. Such statements are part of specific implementation constructs that cannot work on their own. For instance, we cannot separate a single `case` within a `switch` despite the fact that only this one is variable. We display this example in Figure 13. This code snippet is found in BERKELEY DB and includes several problems with scope-sensitive statements, due to the variable `case` (line 5 to 11) and the included `goto` (line 8).

In BERKELEY DB, we found that also `goto` and `return` statements can cause problems. We summarize the occurrences of such situations, we identified during our practical implementation, in Table III. As we can see, potentially problematic constructions are rare within BERKELEY DB, compared to the overall occurrences of scope-sensitive statements. Still, they occur, are challenging to extract, and require adapted refactorings.

In the following, we briefly analyse these statements. We do not provide best practices or patterns, but describe how we migrated such scope-sensitive statement into feature-oriented programming.

Table III. Occurrences of scope-sensitive statements within BERKELEY DB.

	Berkeley DB	Variable Code	Problematic
`switch`	429	71	36
`goto`	4642	634	127
`return`	7779	1254	248

goto Statements In Figure 13, we illustrate a problematic `goto` statement. Assume, we just modularize this statement into a module. This would result in a loss of scope for the `goto` and, thus, an error. In its new scope within the features hook method, the referenced label (`err`) is undefined. Thus, we cannot simply migrate such constructs without further refactorings. Within BERKELEY DB, we identified 635 `goto` statement in variable code. Of these, 20% may cause problems. To resolve such constructs, we found three solution strategies. We applied the first two depending on which was the most promising one, while the third requires a different concept during composition than we applied.

Firstly, the straightforward solution is to migrate not only the `goto` statement but its whole encapsulating method into a module. This solution can be used in all situations but results in code clones and, thus, diminishes the benefits of modularization. In addition, clones can cause new problems and are often considered as bad design [44, 80].

Secondly, in some cases it is possible to extract both, the `goto` statement and its label together. This is often possible for simple error-handling. However, it requires that the variable code by itself does not include scope-sensitive statements. Also, we require further modifications. For instance, we have to preserve the state of all variables in the scope of the `goto` [41]. This results in huge parameter lists, which are considered as anti-pattern in software engineering and, hence, not desirable [44, 63, 64].

Finally, we could apply the concept of *inlining* [81], with which the hook method's call would be completely replaced by its implementation. This would allow us to only extract the `goto` statement as it would be inlined during composition, thus, regaining its correct scope. However, with this approach we separate the `goto` and its label, breaking the convention for this mechanism. Also, we would have to further adapt FEATUREC, or any other composition-based approach used, in an invasive style.

switch Environment As we described, the example in Figure 13 cannot be treated in a straightforward manner. Extracting any `case` without the corresponding `switch` results in errors. The `case` but also the according `break` are unaware of the environment they are used in. We can see in Table III that `switch` environments are rare within BERKELEY DB. Still, while they only occur 71 times in variable code, approximately 50% of them are problematic. Because the number of such situations is relatively low, expensive adoption of tools may not be advisable. Instead, we used two refactorings to resolve `switch` environments and identified a conceptual solution.

The first option is to migrate the `switch` as a whole into a module. Within the code base, only all non-variable artefacts remain. During composition, the feature refines the `switch`. However, this strategy is problematic if too many variables are necessary to preserve the status of all values in scope. A second option is to keep the variable `case` within the base code and use a hook in it. This can require additional adaptations to ensure that the control variable is defined, which can be error-prone. For instance, in Figure 13 we would have to ensure that `DB_HASH` (line 6) is always set even if the feature *`Hash`* is not selected. Finally, we could change the composition mechanism with, for example, a new keyword to refine `case` statements. However, this requires special knowledge and is an error-prone adaptation of existing composers.

return Statement A third problem we encountered within BERKELEY DB is the migration of `return` statements. However, extracting a `return` into a hook is only problematic if it is defined for some but not all possible cases, which may appear in `switch` environments. In those situations, the hook may or may not return a parameter. Thus, it is problematic to simply use a hook method. BERKELEY DB contains 7779 `return` statements of which 1254 are variable and 248 are problematic. We considered two possible solutions during our practical application.

On the one hand, we can use an additional parameter to indicate whether a value is returned. This parameter has to be evaluated on the caller side, requiring some smaller adaptations. On the other hand, we can also directly utilize `return` statements. More precise, if we know, for example, that all originally returned values are positive integers, we can use a negative one to indicate that the return is not valid.

Overall, we found some challenges in extracting scope-sensitive statements that are part of variable code. We propose several solutions to still migrate annotations towards feature-oriented programming. However, further analysis and concepts are necessary to ease such transformations.

8.5. Practical Implementation

Our step-wise migration process can be applied in any company that aims to introduce composition into an annotation-based product line. During our practical application we gained some experiences that a company has to consider:

- *Annotation and composition in concert*: We experienced that we can achieve advantages by combining annotation-based and composition-based implementations. More precise, we could decide for each artefact, which approach is more useful. Especially, we do not have to migrate fine-grained variability. While this is often possible after disciplining annotations, we argue that it is not always beneficial, for instance when considering a single line of code.

  A point that may complicate the integration of both approaches is uniformity [4, 14]. Developers have to assess several implementation techniques and styles, which decreases their understanding. For instance, after migration BERKELEY DB uses C, its preprocessor, and FEATUREC. Hence, we added an additional and non-uniform implementation layer.
- *Step-wise migration*: Our process allows us to only migrate parts of a product line towards composition. Thus, a company does not have to extract complete features. Instead, it can analyse critical parts of the code base and only migrate those for which it is beneficial. This mechanism can significantly lower the adoption barrier.
- *Refactoring legacy systems*: The introduction of composition-based implementation into an annotation-based system is mainly driven by benefits of modularization. However, we found that a step-wise migration is also an opportunity to analyse and refactor legacy systems. On the one hand, this leads to additional costs, for instance disciplining annotations or preserve configurations. On the other hand, a company can improve its source code, documentations, or models. For example, while we had feature models of BERKELEY DB due to previous work, during a step-wise migration an according model can also be extracted.
- *Tool support*: Tooling for combined approaches is insufficient yet. While Kästner and Apel [5] argue that tool integration is only an engineering task, we found that it is quite challenging. A company cannot just use existing tooling and simply put it on top of their development. Current processes must be carefully analysed and suitable implementation approaches assessed. Also, even if suitable tools exist, further adaptations might be necessary to address combined implementation techniques. For instance, we had to change the grammar of FEATUREHOUSE to support annotations.
- *Composition*: Composition provides several benefits, mainly modularization [4, 5, 13, 14]. We found that it is possible to migrate BERKELEY DB's features into separated modules. Hence, it is possible to access and overview all corresponding variable code at once. Still, composition is rarely used in practice [4, 5]. For companies this is a problematic situation. Without suitable experiences in composition-based implementation, migrating a variable system can provide several challenges. For instance, we required profound knowledge to implement FEATUREC or refactor scope-sensitive statements.

To gain new and assess these experiences, further studies especially with an industrial background are required. While we argue that our migration process eases the migration towards composition, we also see challenges.

8.6. Summary

Combining annotation and composition into a system provides several challenges. In this section, we discussed problems we faced during our practical application. Additionally, we summarized and described experiences that are important for companies. Overall, we found that a step-wise migration is a helpful concept to ease the introduction of composition into existing applications. However, there are some aspects that must be addressed in further research. Especially, suitable tooling is an important factor currently hindering composition-based implementations and the application of our process in practice.

9. RELATED WORK

Kästner and Apel [5] formulate the idea of combining annotation-based and composition-based implementation approaches to facilitate step-wise migration. We base on their idea, provide a corresponding process, and assess its application on a real world system. Following, we describe related work categorized by those *focusing on combining implementation techniques* and *studies on refactorings of software product lines*.

Combining implementation techniques There exist several implementation approaches for software product lines with different advantages and disadvantages [3, 4]. The following approaches address the combination of different techniques.

An idea proposed by Apel et al. [82, 83] is to combine feature- and aspect-oriented programming. In particular, shortcomings of feature-oriented programming for incremental development are solved with aspects. Apel et al. [82] also present FEATUREC++, a language extension for C++. However, the used approaches are both composition-based. While we also introduce a language extension, our goal is to integrate feature-oriented programming into an annotated product line.

Kästner and Apel [5] analyse and compare annotation-based and composition-based approaches in detail. They also focus on a combination, which can utilize benefits of both techniques. While our work is based on this article, we propose a migration process for such combinations and investigate its practical applicability. This is complementary to their work as they solely focus on the characteristics, for example, granularity and traceability, but not actual migration processes. Similar to us, Kästner and Apel [5] also provide some code examples on a simple stack implementation. In contrast, we use the real-world system BERKELEY DB to show the potential of our approach.

Kästner et al. [22] describe several formal refactorings from annotations towards composition and vice-versa. They showed that such code transformations are complete and, hence, that both approaches can replace each other. In addition, they provide several case studies to illustrate their refactorings. To do this, they rely on a simplified JAVA version, LIGHTWEIGHT JAVA [84]. This complements our work, as it could be possible to use their proposed migrations during our process. Still, the focus of this article is on the practical implementation and combination of composition and annotation rather than just refactorings.

Walkingshaw and Erwig [85] introduce a formal calculus for the implementation of variability. The proposed model unifies composition and annotation on a formal basis. In contrast to our work, no practical application or technical discussion is provided.

Behringer [86] proposes to unify annotation-based and composition-based approaches. The goal is to improve the integration of both techniques, for instance with adapted tools. In another article, Behringer and Rothkugel [87] build on this idea and introduce the usage of structured document graphs. This model enables developers to switch between composition-based, annotation-based, and mixed implementations of the same product line. Overall, these works complement ours and can provide assistance during our migration concept. Especially the need for suitable tool support will be addressed in their further work.

Finally, Krüger et al. [88] proposed compositional annotations to integrate the idea of physically separating features into annotation-based approaches. They discussed the advantages and shortcomings of their idea but did not implement a tool or evaluate it. The goal is to facilitate later migrations towards composition-based implementations, such as feature-oriented programming. Hence, this approach might be able to further prepare our process and lower the adoption barrier.

Refactoring studies Several case studies address the extraction or migration of existing systems and product lines towards composition. Following, we briefly summarize some of those.

A common case study is the refactoring of annotated database systems towards aspect-oriented product lines. Tesanovic et al. [71] show the possibilities of tailoring and managing BERKELEY DB with aspects. They focus on the benefits and shortcomings of aspect-oriented development for databases. Similar to this, Kästner et al. [72] also refactored BERKELEY DB using ASPECTJ [89]. They conclude, that legacy applications are rarely designed for feature-extensibility. In their opinion, the migration towards composition is not only a difficult task but ASPECTJ is also not appropriate to do this. Contrary to these works, we integrated feature-oriented programming into BERKELEY DB instead of fully migrating towards aspects. Also, we based our study on a step-wise migration process that provides guidance during the process.

Alves et al. [76] present another case study on a real-world product line. They also use aspect-oriented programming to refactor the system towards composition. In addition, they propose several language-specific strategies to transform certain patterns. Hence, their work provides additional guidance for the actual migration in our process and partly complements the insights we gained.

Reynolds et al. [77] analysed the code of the LINUX kernel to analyse the potential to extend it by using aspect-oriented programming. They conclude, that most migrations of existing preprocessor annotations could be straightforward. While we are not aware of an according case study in which the kernel is migrated, this work provides details on analysing annotated code. Hence, it can provide help in assessing the potential of migrating a product line towards composition.

Finally, Rosenmüller et al. [41] extracted a feature-oriented implementation of BERKELEY DB. In contrast to us, they first refactored the C code towards C++. Only afterwards, the object-oriented code is migrated into modules. We focused on the direct application of feature-oriented programming on the original code. Hence, we avoid additional refactorings, lowering costs and risks. Also, the focus of Rosenmüller et al. [41] is to show the ability to customize database systems without performance loss, while our scope is the migration process itself, without previously moving towards object-oriented programming.

10. CONCLUSION

Integrating annotation-based and composition-based implementation techniques promises to raise benefits of both [5]. In practice, annotation is the dominant strategy [4, 5, 16]. Hence, most companies would have to introduce composition into an existing product line. This is an error-prone and expensive task.

To facilitate this migration from annotation-based to composition-based implementations and lower its adoption barrier [18], we propose a step-wise migration process. We focus on preprocessor annotations, which are widely used [7], and feature-oriented programming [4, 13, 14]. For each step in our process, we analyse possible tool support and required effort. In addition, we introduce FEATUREC, a FEATUREHOUSE [21] adaptation, to support annotations within feature modules.

The focus of this article is the practical application of the migration process. Therefore, we have used BERKELEY DB, an embedded database system implemented in C, as an example. We applied our step-wise migration to refactor BERKELEY DB partly towards composition. Our migration illustrated the practical usability of our process. It provides guidance and allows to migrate a product line in a consistent way. Additionally, we are able to migrate only selected artefacts while keeping annotations where beneficial. Still, we experienced some challenges during this study. For example, we discussed the preservation of legacy configurations and scope-sensitive statements.

Our approach suggests to facilitate the introduction of composition-based implementation techniques in practice. Nevertheless, there are open issues to address in future work. Firstly, additional case studies and industrial applications are needed. By this means, we will continue to refine our process and apply it to other implementation techniques. In cooperation with industrial partners, further and more practical insights could be gained.

Secondly, a detailed and adapted tool-chain for the migration process is a prerequisite. Several other approaches start to develop combinations of annotation-based and composition-based approaches [22, 86, 87]. However, these are only under progress and require further analysis. Also, it would be advantageous to integrate them into a single framework.

Finally, we plan to analyse scope-sensitive statements. In existing refactorings from annotation towards composition and the other way around [22], we found no detailed analysis of such constructs, yet. Still, our practical application showed that they are quite common in annotated product lines and are difficult to migrate. In this regard, our goal is to provide a detailed catalogue with solution strategies and further analyses of existing systems.

ACKNOWLEDGEMENTS

This research was partly supported by DFG grants LE 3382/2-1 and SA 465/49-1, BMBF grants 01IS14017A and 01IS14017B, and Volkswagen Financial Services AG.

REFERENCES

[1] Clements PC, Northrop LM. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.
[2] Pohl K, Böckle G, van der Linden FJ. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.
[3] Gacek C, Anastasopoules M. Implementing Product Line Variabilities. *Symposium on Software Reusability: Putting Software Reuse in Context*, SSR, ACM, 2001; 109–117, doi:10.1145/375212.375269.
[4] Apel S, Batory D, Kästner C, Saake G. *Feature-Oriented Software Product Lines*. Springer, 2013, doi:10.1007/978-3-642-37521-7.
[5] Kästner C, Apel S. Integrating Compositional and Annotative Approaches for Product Line Engineering. *GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering*. MCGPLE, University of Passau, 2008; 35–40.
[6] Medeiros F, Kästner C, Ribeiro M, Nadi S, Gheyi R. The Love/Hate Relationship with the C Preprocessor: An Interview Study. *European Conference on Object-Oriented Programming*. ECOOP, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015; 495–518, doi:10.4230/LIPIcs.ECOOP.2015.495.
[7] Hunsen C, Zhang B, Siegmund J, Kästner C, Leßenich O, Becker M, Apel S. Preprocessor-Based Variability in Open-Source and Industrial Software Systems: An Empirical Study. *Empirical Software Engineering* 2016; **21**(2):449–482, doi:10.1007/s10664-015-9360-1.
[8] Le D, Walkingshaw E, Erwig M. #ifdef Confirmed Harmful: Promoting Understandable Software Variation. *Symposium on Visual Languages and Human-Centric Computing*. VL/HCC, IEEE, 2011; 143–150, doi:10.1109/VLHCC.2011.6070391.
[9] Feigenspan J, Kästner C, Apel S, Liebig J, Schulze M, Dachselt R, Papendieck M, Leich T, Saake G. Do Background Colors Improve Program Comprehension in the #ifdef Hell? *Empirical Software Engineering* 2013; **18**(4):699–745, doi:10.1007/s10664-012-9208-x.
[10] Thüm T. Product-Line Specification and Verification with Feature-Oriented Contracts. PhD Thesis, University of Magdeburg 2015.
[11] Kenner A, Kästner C, Haase S, Leich T. TypeChef: Toward Type Checking #Ifdef Variability in C. *International Workshop on Feature-Oriented Software Development*, FOSD, ACM, 2010; 25–32, doi:10.1145/1868688.1868693.
[12] Kästner C, Apel S, Thüm T, Saake G. Type Checking Annotation-Based Product Lines. *ACM Transactions on Software Engineering and Methodology* 2012; **21**(3):14:1–14:39, doi:10.1145/2211616.2211617.
[13] Prehofer C. Feature-Oriented Programming: A Fresh Look at Objects. *European Conference on Object-Oriented Programming*. ECOOP, Springer, 1997; 419–443, doi:10.1007/BFb0053389.
[14] Batory D, Sarvela JN, Rauschmayer A. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering* 2004; **30**(6):355–371, doi:10.1109/TSE.2004.23.
[15] Krüger J. Lost in Source Code: Physically Separating Features in Legacy Systems. *International Conference on Software Engineering Companion*. ICSE-C, IEEE, 2017; 461–462, doi:10.1109/ICSE-C.2017.46.

[16] Kästner C, Apel S, Kuhlemann M. Granularity in Software Product Lines. *International Conference on Software Engineering*. ICSE, ACM, 2008; 311–320, doi:10.1145/1368088.1368131.

[17] Benduhn F, Schröter R, Kenner A, Kruczek C, Leich T, Saake G. Migration from Annotation-Based to Composition-Based Product Lines: Towards a Tool-Driven Process. *International Conference on Advances and Trends in Software Engineering*. SOFTENG, IARIA, 2016; 102–109.

[18] Clements PC, Krueger CW. Point / Counterpoint: Being Proactive Pays Off / Eliminating the Adoption Barrier. *IEEE Software* 2002; **19**(4):28–31, doi:10.1109/MS.2002.1020283.

[19] Krüger J, Fenske W, Meinicke J, Leich T, Saake G. Extracting Software Product Lines: A Cost Estimation Perspective. *International Systems and Software Product Line Conference*. SPLC, ACM, 2016; 354–361, doi:10.1145/2934466.2962731.

[20] Apel S, Kästner C, Lengauer C. FeatureHouse: Language-Independent, Automated Software Composition. *International Conference on Software Engineering*. ICSE, IEEE, 2009; 221–231, doi:10.1109/ICSE.2009.5070523.

[21] Apel S, Kästner C, Lengauer C. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Transactions on Software Engineering* 2013; **39**(1):63–79, doi:10.1109/TSE.2011.120.

[22] Kästner C, Apel S, Kuhlemann M. A Model of Refactoring Physically and Virtually Separated Features. *International Conference on Generative Programming and Component Engineering*. GPCE, ACM, 2009; 157–166, doi:10.1145/1621607.1621632.

[23] Jarzabek S, Bassett P, Zhang H, Zhang W. XVCL: XML-Based Variant Configuration Language. *International Conference on Software Engineering*. ICSE, IEEE, 2003; 810–811, doi:10.1109/ICSE.2003.1201298.

[24] Pawlak R. Spoon: Annotation-Driven Program Transformation - the AOP Case. *Workshop on Aspect Oriented Middleware Development*. AOMD, ACM, 2005; 1–6, doi:10.1145/1101560.1101566.

[25] Kernighan BW, Ritchie DM. *The C Programming Language*. Prentice Hall, 1988.

[26] Leich T, Apel S, Marnitz L, Saake G. Tool Support for Feature-Oriented Software Development - FeatureIDE: An Eclipse-Based Approach. *OOPSLA Workshop on Eclipse Technology eXchange*. eclipse, ACM, 2005; 55–59, doi:10.1145/1117696.1117708.

[27] Liebig J, Kästner C, Apel S. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. *International Conference on Aspect-Oriented Software Development*. AOSD, ACM, 2011; 191–202, doi:10.1145/1960275.1960299.

[28] Schulze S, Liebig J, Siegmund J, Apel S. Does the Discipline of Preprocessor Annotations Matter?: A Controlled Experiment. *SIGPLAN Notices* 2013; **49**(3):65–74, doi:10.1145/2637365.2517215.

[29] Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier JM, Irwin J. Aspect-Oriented Programming. *European Conference on Object-Oriented Programming*. ECOOP, Springer, 1997; 220–242, doi:10.1007/BFb0053381.

[30] Schaefer I, Bettini L, Bono V, Damiani F, Tanzarella N. Delta-Oriented Programming of Software Product Lines. *International Systems and Software Product Line Conference*. SPLC, Springer, 2010; 77–91, doi:10.1007/978-3-642-15579-6_6.

[31] Lee J, Muthig D. Feature-Oriented Variability Management in Product Line Engineering. *Communications of the ACM* 2006; **49**(12):55–59, doi:10.1145/1183236.1183266.

[32] Kästner C, Thüm T, Saake G, Feigenspan J, Leich T, Wielgorz F, Apel S. FeatureIDE: A Tool Framework for Feature-oriented Software Development. *International Conference on Software Engineering*, ICSE, IEEE, 2009; 611–614, doi:10.1109/ICSE.2009.5070568.

[33] Czarnecki K, Eisenecker UW. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[34] Schobbens PY, Heymans P, Trigaux JC. Feature Diagrams: A Survey and a Formal Semantics. *International Conference Requirements Engineering*. RE, IEEE, 2006; 139–148, doi:10.1109/RE.2006.23.

[35] Chen L, Babar MA. A Systematic Review of Evaluation of Variability Management Approaches in Software Product Lines. *Information and Software Technology* 2011; **53**(4):344–362, doi:10.1016/j.infsof.2010.12.006.

[36] Schaefer I, Rabiser R, Clarke D, Bettini L, Benavides D, Botterweck G, Pathak A, Trujillo S, Villela K. Software Diversity: State of the Art and Perspectives. *International Journal on Software Tools for Technology Transfer* 2012; **14**(5):477–495, doi:10.1007/s10009-012-0253-y.

[37] Czarnecki K, Grünbacher P, Rabiser R, Schmid K, Wąsowski A. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. *International Workshop on Variability Modeling of Software-Intensive Systems*. VaMoS, ACM, 2012; 173–182, doi:10.1145/2110147.2110167.

[38] Mohabbati B, Asadi M, Gašević D, Hatala M, Müller HA. Combining Service-Orientation and Software Product Line Engineering: A Systematic Mapping Study. *Information and Software Technology* 2013; **55**(11):1845–1859, doi:http://dx.doi.org/10.1016/j.infsof.2013.05.006.

[39] Berger T, Rublack R, Nair D, Atlee JM, Becker M, Czarnecki K, Wąsowski A. A Survey of Variability Modeling in Industrial Practice. *International Workshop on Variability Modelling of Software-Intensive Systems*. VaMoS, ACM, 2013; 7:1–7:8, doi:10.1145/2430502.2430513.

[40] Liebig J, Apel S, Lengauer C, Kästner C, Schulze M. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. *International Conference on Software Engineering*. ICSE, ACM, 2010; 105–114, doi:10.1145/1806799.1806819.

[41] Rosenmüller M, Apel S, Leich T, Saake G. Tailor-Made Data Management for Embedded Systems: A Case Study on Berkeley DB. *Data & Knowledge Engineering* 2009; **68**(12):1493–1512, doi: 10.1016/j.datak.2009.07.013.

[42] Thüm T, Kästner C, Benduhn F, Meinicke J, Saake G, Leich T. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming* 2014; **79**:70–85, doi:http://dx.doi.org/10.1016/j.scico.2012.06.002.

[43] Meinicke J, Thüm T, Schröter R, Krieter S, Benduhn F, Saake G, Leich T. FeatureIDE: Taming the Preprocessor Wilderness. *International Conference on Software Engineering*, ICSE, ACM, 2016; 629–632, doi:10.1145/2889160.2889175.

[44] Fowler M, Beck K, Brant J, Opdyke W, Roberts D. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[45] Mens T, Tourwé T. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering* 2004; **30**(2):126–139.

[46] Lonchamp J. A Structured Conceptual and Terminological Framework for Software Process Engineering. *International Conference on the Software Process*. ICSP, IEEE, 1993; 41–53, doi:10.1109/SPCON.1993.236823.

[47] Rumbaugh J, Jacobson I, Booch G. *The Unified Modeling Language Reference Manual*. Pearson, 2004.

[48] Fenske W, Thüm T, Saake G. A Taxonomy of Software Product Line Reengineering. *International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS, ACM, 2014; 4:1–4:8, doi: 10.1145/2556624.2556643.

[49] Humphrey WS. *Managing the Software Process*. Addison-Wesley, 1989.

[50] Krueger CW. BigLever Software Gears and the 3-tiered SPL Methodology. *ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*. OOPSLA, ACM, 2007; 844–845, doi: 10.1145/1297846.1297918.

[51] Beuche D. Modeling and Building Software Product Lines with Pure::Variants. *International Systems and Software Product Line Conference*. SPLC, ACM, 2012; 255, doi:10.1145/2364412.2364457.

[52] Laguna MA, Crespo Y. A Systematic Mapping Study on Software Product Line Evolution: From Legacy System Reengineering to Product Line Refactoring. *Science of Computer Programming* 2013; **78**(8):1010–1034, doi:10.1016/j.scico.2012.05.003.

[53] Kästner C, Dreiling A, Ostermann K. Variability Mining with LEADT. *Technical Report*, Philipps University Marburg 2011.

[54] Dit B, Revelle M, Gethers M, Poshyvanyk D. Feature Location in Source Code: A Taxonomy and Survey. *Journal of Software: Evolution and Process* 2013; **25**(1):53–95, doi:10.1002/smr.567.

[55] Assunção WKG, Vergilio SR. Feature Location for Software Product Line Migration: A Mapping Study. *International Systems and Software Product Line Conference*. SPLC, ACM, 2014; 52–59, doi:10.1145/2647908.2655967.

[56] Nadi S, Berger T, Kästner C, Czarnecki K. Mining Configuration Constraints: Static Analyses and Empirical Results. *International Conference on Software Engineering*. ICSE, ACM, 2014; 140–151, doi:10.1145/2568225.2568283.

[57] Nadi S, Berger T, Kästner C, Czarnecki K. Where do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. *IEEE Transactions on Software Engineering* 2015; **41**(8):820–841, doi:10.1109/TSE.2015.2415793.

[58] Yu D, Geng P, Wu W. Constructing Traceability between Features and Requirements for Software Product Line Engineering. *Asia-Pacific Software Engineering Conference*. APSEC, IEEE, 2012; 27–34, doi:10.1109/APSEC.2012.135.

[59] Yu D, Chen Z, Zhang Y. From Goal Models to Feature Models: A Rule-Based Approach for Software Product Lines. *Asia-Pacific Software Engineering Conference*. APSEC, IEEE, 2015; 277–284, doi: 10.1109/APSEC.2015.22.

[60] Biggerstaff TJ, Mitbander BG, Webster D. The Concept Assignment Problem in Program Understanding. *International Conference on Software Engineering*. ICSE, IEEE, 1993; 482–498.

[61] Kästner C, Dreiling A, Ostermann K. Variability Mining: Consistent Semi-Automatic Detection of Product-Line Features. *IEEE Transactions on Software Engineering* 2014; **40**(1):67–82, doi: 10.1109/TSE.2013.45.

[62] Kästner C. CIDE: Decomposing Legacy Applications into Features. *International Systems and Software Product Line Conference*. SPLC, 2007; 149–150.

[63] Mäntylä M. Bad Smells in Software - A Taxonomy and an Empirical Study. PhD Thesis, Helsinki University of Technology 2003.

[64] Moha N, Gueheneuc YG, Duchien L, Le Meur AF. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering* 2010; **36**(1):20–36, doi:10.1109/TSE.2009.50.

[65] Kästner C, Apel S. Virtual Separation of Concerns - A Second Chance for Preprocessors. *Journal of Object Technology* 2009; **8**(6):59–78.

[66] Fenske W, Schulze S, Meyer D, Saake G. When Code Smells Twice as Much: Metric-Based Detection of Variability-Aware Code Smells. *International Working Conference on Source Code Analysis and Manipulation*. SCAM, IEEE, 2015; 171–180, doi:10.1109/SCAM.2015.7335413.

[67] Murphy GC, Kersten M, Findlater L. How are Java Software Developers Using the Elipse IDE? *IEEE Software* 2006; **23**(4):76–83, doi:10.1109/MS.2006.105.

[68] Runeson P. A Survey of Unit Testing Practices. *IEEE Software* 2006; **23**(4):22–29, doi:10.1109/MS.2006.91.

[69] Myers GJ, Sandler C, Badgett T. *The Art of Software Testing*. Wiley, 2011.

[70] Engström E, Runeson P. Software Product Line Testing - A Systematic Mapping Study. *Information and Software Technology* 2011; **53**(1):2–13, doi:10.1016/j.infsof.2010.05.011.

[71] Tesanovic A, Sheng K, Hansson J. Application-Tailored Database Systems: A Case of Aspects in an Embedded Database. *International Database Engineering and Applications Symposium*, IDEAS, IEEE, 2004; 291–301, doi:10.1109/IDEAS.2004.1319803.

[72] Kästner C, Apel S, Batory D. A Case Study Implementing Features Using AspectJ. *International Systems and Software Product Line Conference*. SPLC, IEEE, 2007; 223–232, doi:10.1109/SPLINE.2007.12.

[73] Lopez-Herrejon RE, Montalvillo-Mendizabal L, Egyed A. From Requirements to Features: An Exploratory Study of Feature-Oriented Refactoring. *International Systems and Software Product Line Conference*. SPLC, IEEE, 2011; 181–190, doi:10.1109/SPLC.2011.52.

[74] Murphy GC, Lai A, Walker RJ, Robillard MP. Separating Features in Source Code: An Exploratory Study. *International Conference on Software Engineering*, ICSE, IEEE, 2001; 275–284, doi:10.1109/ICSE.2001.919101.

[75] Liebig J, Janker A, Garbe F, Apel S, Lengauer C. Morpheus: Variability-Aware Refactoring in the Wild. *International Conference on Software Engineering*. ICSE, IEEE, 2015; 380–391.

[76] Alves V, Costa Neto A, Soares S, Santos G, Calheiros F, Nepomuceno V, Pires D, Leal J, Borba P. From Conditional Compilation to Aspects: A Case Study in Software Product Lines Migration. *Workshop on Aspect-Oriented Product Line Engineering*. AOPLE, 2006.

[77] Reynolds A, Fiuczynski ME, Grimm R. On the Feasibility of an AOSD Approach to Linux Kernel Extensions. *AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, ACP4IS, ACM, 2008; 8:1–8:7, doi:10.1145/1404891.1404899.

[78] Adams B, De Meuter W, Tromp H, Hassan AE. Can We Refactor Conditional Compilation into Aspects? *International Conference on Aspect-Oriented Software Development*, AOSD, ACM, 2009; 243–254, doi:10.1145/1509239.1509274.

[79] Al-Hajjaji M, Thüm T, Meinicke J, Lochau M, Saake G. Similarity-Based Prioritization in Software Product-line Testing. *International Systems and Software Product Line Conference*, SPLC, ACM, 2014; 197–206, doi:10.1145/2648511.2648532.

[80] Fenske W, Schulze S. Code Smells Revisited: A Variability Perspective. *International Workshop on Variability Modelling of Software-Intensive Systems*. VaMoS, ACM, 2015; 3–10, doi:10.1145/2701319.2701321.

[81] Thüm T, Apel S, Zelend A, Schröter R, Möller B. Subclack: Feature-oriented Programming with Behavioral Feature Interfaces. *Workshop on MechAnisms for SPEcialization, Generalization and inHerItance*, MASPEGHI, ACM, 2013; 1–8, doi:10.1145/2489828.2489829.

[82] Apel S, Leich T, Rosenmüller M, Saake G. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. *International Conference on Generative Programming and Component Engineering*, GPCE, Springer, 2005; 125–140, doi:10.1007/11561347_10.

[83] Apel S, Leich T, Saake G. Aspectual Feature Modules. *IEEE Transactions on Software Engineering* 2008; **34**(2):162–180, doi:10.1109/TSE.2007.70770.

[84] Strniša R, Sewell P, Parkinson M. The Java Module System: Core Design and Semantic Definition. *Conference on Object-Oriented Programming Systems and Applications*, OOPSLA, ACM, 2007; 499–514, doi:10.1145/1297027.1297064.

[85] Walkingshaw E, Erwig M. A Calculus for Modeling and Implementing Variation. *ACM SIGPLAN Notices* 2012; **48**(3):132–140, doi:10.1145/2480361.2371421.

[86] Behringer B. Integrating Approaches for Feature Implementation. *International Symposium on Foundations of Software Engineering*, FSE, ACM, 2014; 775–778, doi:10.1145/2635868.2666605.

[87] Behringer B, Rothkugel S. Integrating Feature-Based Implementation Approaches Using a Common Graph-based Representation. *Symposium on Applied Computing*, SAC, ACM, 2016; 1504–1511, doi:10.1145/2851613.2851791.

[88] Krüger J, Schröter I, Kenner A, Kruczek C, Leich T. FeatureCoPP: Compositional Annotations. *International Workshop on Feature-Oriented Software Development*. FOSD, ACM, 2016; 74–84, doi:10.1145/3001867.3001876.

[89] Laddad R. *AspectJ in Action: Practical Aspect-Oriented Programming*. Dreamtech Press, 2003.