

# Migration from Annotation-Based to Composition-Based Product Lines: Towards a Tool-Driven Process

Fabian Benduhn<sup>1,2</sup>, Reimar Schröter<sup>1</sup>, Andy Kenner<sup>2</sup>, Christopher Kruczek<sup>2</sup>,  
Thomas Leich<sup>2</sup>, and Gunter Saake<sup>1</sup>

University Magdeburg<sup>1</sup>, METOP GmbH<sup>2</sup>, Germany

<sup>1</sup>email:{fabian.benduhn, reimar.schroeter, gunter.saake}@ovgu.de,

<sup>2</sup>email:{andy.kenner, christopher.kruczek, thomas.leich}@metop.de

**Abstract**—Software product lines allow a developer to produce similar programs based on a common code base. Two main techniques exist: composition-based and annotation-based approaches. Although composition-based approaches offer potential advantages such as maintainability, in practice mostly annotation-based approaches are used. The main reason hindering the migration of existing projects is the difficulty of the transformation process that can take a lot of time in which maintenance and evolution of the system are put on hold. Thus, for a company, it is hard to estimate the transformation costs and the success is uncertain. As already stated in previous work, a hybrid solution using both approaches may be an adequate solution to overcome this problem. Therefore, we propose a migration concept focusing on technical requirements, such as tool- and language support to reduce the risk during the error-prone migration process. We exemplify the concept by considering the partial migration of a real-world system from preprocessor-based variability to an implementation based on feature-oriented programming. We identify conceptual and tool-based challenges that must be addressed for the practical application. We present technical considerations that must be taken into account for step-wise migration and specific challenges related to our case study.

**Keywords**—Software Product Lines; Step-wise Migration; Variability Mechanisms; Implementation Techniques.

## I. INTRODUCTION

Software product lines are a concept to create similar programs based on a common code base [1][2]. Several implementation techniques with different advantages and disadvantages exist [3]. We distinguish between annotation-based and composition-based approaches. In practice, variability is often implemented by annotating code with preprocessor directives to achieve conditional compilation. In detail, preprocessors are an easily accessible mechanism for fine-grained adaptation of source code to achieve similar programs with low effort. While the use of preprocessors provides an effective way to implement software variability, the technique comes with disadvantages related to code and feature traceability [4][5].

In contrast, composition-based approaches, such as feature-oriented programming (FOP), avoid these problems by providing specialized modularization mechanisms [6][7]. In FOP, each feature, which serves as a configuration option, is encapsulated in a dedicated module so that it can be combined with other features to generate variants. Therefore, compared to annotation-based approaches, the traceability of features is straight forward, which eases maintenance and extension of the source code.

Despite the potential advantages of composition-based over annotation-based approaches, their use has not been widely adopted in industry. There are several reasons for this situation. On the one hand, the usage of FOP in practice is difficult and error-prone and has high requirements regarding sufficient tool support. On the other hand, the process of a complete transformation to these techniques for legacy systems can be time consuming, and it is hard to estimate the transformation costs [8]. Thus, preprocessors remain the dominant approach in industry.

Kästner and Apel formulated the idea to use a combination of annotation-based and composition-based approaches to combine their advantages and to enable a step-wise migration process [9]. We build on this idea of such a hybrid approach and propose a process for its instantiation considering practical concerns, such as required tool support. In a case study, we investigate how Berkeley DB, a database management system using preprocessor directives to implement variability, could be migrated using our migration concept. We present details of the step-wise migration and identify technical and conceptual challenges.

In detail, we make the following contributions:

- We propose a concept for the instantiation of a step-wise migration process based on the combination of annotation-based and composition-based approaches that considers technical concerns.
- We exemplify our concept considering the migration of Berkeley DB, from an implementation in the programming language C with preprocessor annotations to a composition-based implementation using FeatureC, a newly developed extension of FeatureHouse [10].
- We identify technical and conceptual challenges that we have to consider during our migration concept.

In Section II, we introduce the necessary background for the rest of the paper. We propose our tool-driven migration concept in Section III and its practical application to a real-world project in Section IV. In Section V, we discuss several project-specific challenges. We give an overview of related work in Section VI and conclude in Section VII.

## II. IMPLEMENTATION OF VARIABILITY IN SOFTWARE PRODUCT LINES

In this section, we give a brief overview on implementation techniques for software product lines. We consider two basic approaches, annotation-based and composition-based [3].

```

1 static int __bam_getbothc(dbc, data)
2   DBC *dbc;
3   DBT *data;
4 {
5   ...
6   return (__bam_getboth_finddatum(dbc, data, ...));
7 }
8
9 #ifdef HAVE_COMPRESSION
10 static int __bam_getlte(dbc, key, data)
11   DBC *dbc;
12   DBT *key, *data;
13 {
14   int ret;
15   ...
16   return (ret);
17 }
18 #endif

```

Figure 1. Conditional compilation using preprocessor directives.

In detail, we focus on the representation of variability in source-code artifacts. However, the distinction between both approaches can be generally applied to many types of variable development artifacts, such as models, specifications, and documentations [11][12][13]. Now, we present advantages of both approaches and their combination.

#### A. Annotation-Based Implementation of Variability

The idea of annotation-based approaches to implement variability is that certain parts of the development artifacts are mapped to features by annotating them. Individual variants can be generated by removing parts representing undesired features.

As annotation-based technique, preprocessors are commonly used that exist for many languages and tools [14][15][16]. In practice, the usage of preprocessors is widespread, e.g., in open-source projects [17], operating systems, and databases. The mechanism also allows a developer to implement fine-grained variability, including changes to single characters within program statements. In Figure 1, we exemplify the usage of the C preprocessor for conditional compilation. The code in Line 9 is annotated using feature `HAVE_COMPRESSION`. In detail, the beginning of the variable code fragment is marked by the directive `#ifdef`, the end is marked by `#endif`. Thus, if feature `HAVE_COMPRESSION` is not activated, the preprocessor removes the specific code before it is given to the compiler.

Preprocessors support very fine-grained source code variability but this property negatively influences the code comprehension of the system. Thus, alternative annotation-based approaches with more sophisticated tool support and certain restrictions regarding the discipline of annotation usage have been developed [18][19][20]. Similarly, researchers have investigated the code comprehension of preprocessor programs and related problems and proposed several approaches to improve them, e.g., with different background colors [4][18]. Despite these efforts, the general problems with such annotation-based approaches, such as the C preprocessor, are not completely solved. However, preprocessors remain the dominant approach in practice.

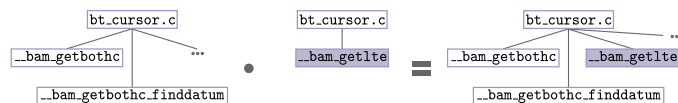


Figure 2. FOP - Structural combination of code artifacts.

#### B. Composition-Based Implementation of Variability

To overcome the problems of annotation-based approaches, several composition-based implementation techniques have been proposed [6][7][21][22][23][24][25]. In composition-based approaches, variable features of a system are mapped to dedicated modules. The main advantage is that this modularization improves traceability of features and separation of concerns [3].

In this paper, we focus on the composition-based approach FOP. The main idea of FOP is to modularize software into cohesive units — each module encapsulating a particular feature of the software. Individual program variants can be generated by superimposing the desired feature modules based on a hierarchical representation of their structure, called Feature Structure Trees (FST) [10]. In detail, two corresponding inner nodes (i.e., nonterminal) are merged when they have the same type and name. For terminals (i.e. leave nodes, e.g., functions), it depends on the implementation, i.e., whether the node is refined (i.e., extended) or completely overwritten. In Figure 2, we exemplify composition-based variability for FOP. We use the same example as given for the annotation-based approach of Figure 1. In the base feature, the file `bt_cursor.c` with the functions `__bam_getbothc` and `__bam_getbothc_finddatum` exists. However, the file `bt_cursor.c` also exists in feature `HAVE_COMPRESSION` with the function `__bam_getlte`. After the combination, the file includes all functions of both features.

#### C. Combination of Annotation-Based and Composition-Based Approaches

Despite the potential advantages of composition-based approaches, there are some problems regarding their practical application in industry. First, the migration from legacy systems, which often use preprocessors to implement variability, to composition-based approaches is a difficult and error-prone task. Long migration processes are generally problematic, because they must usually be performed in parallel to the regular development and maintenance of the software, and eventually merged - which is, again, a time-consuming task. Thus, the risk to switch to composition-based approaches is high. Second, composition-based approaches are not suitable for fine-grained variability as used in many preprocessor-based systems. Thus, it is especially difficult to ensure that a migration process results in a well-structured system, which may diminish the potential benefits of the composition-based approach.

As a possible way out of this dilemma, Kästner and Apel proposed the idea to combine annotation-based and composition-based approaches [9]. In detail, they formulated the idea to use this concept to enable a step-wise migration process from annotation-based to composition-based approaches in which intermediate steps use the combined approach and, thus, avoid the usually long, atomic migration process. Kästner and Apel discuss this migration process in a theoretical manner

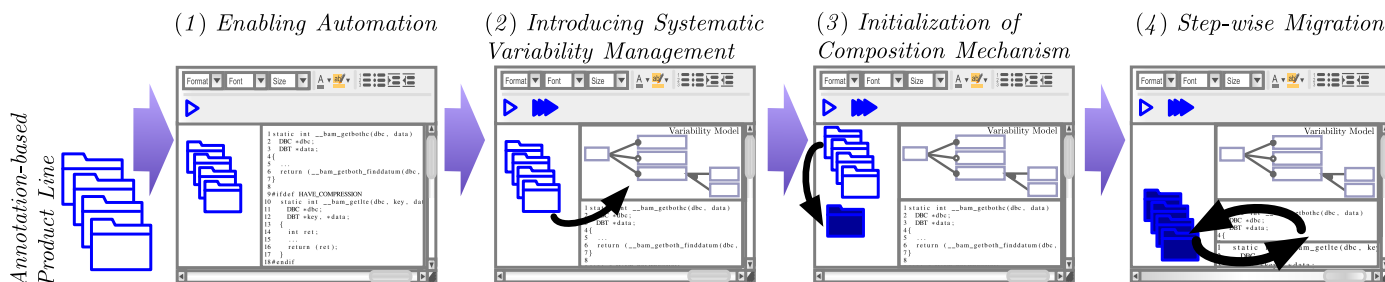


Figure 3. General migration concept for the instantiation of a step-wise migration of annotation-based to composition-based approaches.

and state the need for sufficient tool support as a necessary prerequisite for its practical application. We present a migration concept and discuss practical considerations for companies that are interested in the application of a system transformation.

### III. TOOL-DRIVEN MIGRATION CONCEPT

In this section, we introduce our concept for the tool-driven migration of software systems with annotation-based implementation of variability to a system using composition-based variability. Our concept is generally applicable in the sense that it is independent of specific annotation-based and composition-based implementation techniques. It describes a process consisting of four steps, as depicted in Figure 3. In particular, we have to (1) enable automation, (2) introduce support for systematic variability management, (3) initialize the desired composition mechanism, and (4) apply the step-wise migration.

One of the goals of our approach is to ensure a consistent state in each step. Thus, a long migration process that hinders regular development of the project is avoided and costs to merge the migration results can be reduced. When applying this to concrete projects, the details on how this consistency is ensured may vary. Ensuring a minimal interruption of the continuous development, requires that the application can be compiled, executed, and tested in each step. In the following, we describe each step of our concept in detail.

#### (1) Enabling Automation

The first step of our migration concept is to enable automation of important tasks in the software-development process for a given project. This will enable the developer to validate the correctness of each subsequent step. Typically, this includes the capability to automate the process of building, executing, and testing the program as provided by many modern integrated development environments (IDEs). The choice of the specific IDE depends on the later steps, e.g., the IDE’s support for variability management tools. Thus, even if the project already provides automation for relevant tasks, it might be still necessary to switch the used IDE in preparation of the next steps.

#### (2) Introducing Systematic Variability Management

Having ensured that we enabled automation, execution, and testing of our software, a systematic variability management should be introduced. An important aspect of variability management is to provide techniques for the modeling of the variability space and a mapping to code artifacts. Therefore, the variability artifacts must be identified and possibly re-engineered for integration. Furthermore, variability-related tool

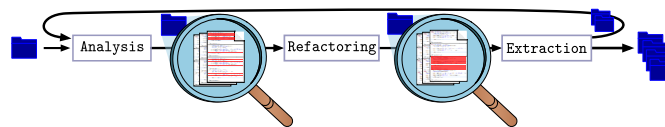


Figure 4. Step-wise migration in detail.

support must be provided to establish a tool-driven configuration process and the automation of variant generation. In addition, the possibility to execute variants or test them is a frequently desired aspect.

#### (3) Initialization of the Composition Mechanism

In this step, the technical prerequisites to employ a composition mechanism must be ensured. The result of this step is a trivial decomposition into a single module. However, it must be ensured that the used tools support the desired language, and that modules can be composed accordingly. For instance, if we take FOP as composition-based approach, we have to initialize the project using one base feature module in which we include all code fragments with all preprocessor directives. Afterwards, it must be possible to apply the composition mechanism, which, in this step, results in an output that is equal to the input (i.e., equal to the base feature). Furthermore, we have to ensure that the automated variant-generation mechanism using preprocessors of Step (2) can be used to remove the undesired features of this composed output.

#### (4) Step-wise Migration

After the execution of the previous steps, the main task of the step-wise migration can start. The step-wise migration consists of small refactoring steps in which we can extract a specific source-code artifact into a module (cf. Figure 4). Therefore, it is necessary to (4.1) analyze the code to identify the next fragment for the modularization, (4.2) refactor the code to improve the structure, so that it is easier to modularize, and (4.3) extract the corresponding code fragment into a module. Since we define each sub-step as a refactoring, the complete product line should be in a consistent state before and after each step. This can be ensured by taking advantage of the support for automated building, executing, and testing, which has been ensured in the first step.

### IV. PRACTICAL APPLICATION TO A REAL-WORLD PROJECT

In the previous section, we introduced a general process for the step-wise migration of annotation-based to composition-based approaches with the focus on necessary tool support to

enable a successful practical application. Now, we investigate the applicability of this process to initialize the migration of a real-world project. Our focus is to identify technical and conceptual concerns and possible challenges for the process that may be useful to guide companies in their own migration projects.

We consider the application of the proposed migration concept to Berkeley DB, a database management system, from an annotation-based implementation using the C preprocessor to a composition-based implementation based on FOP. In detail, Berkeley DB consists of 229,419 lines of code. Since Berkeley DB is written in C with preprocessor annotations, it is a practically relevant case study with high challenges of an annotation-based product line. Another reason for selecting Berkeley DB is that it was used in several case studies related to research on variability (for more details, see Section VI). According to the properties of Berkeley DB, we have to search for an IDE that allows us to apply our migration concept and that supports variability management for C with preprocessors and a solution for FOP in the programming language C.

Since it supports the required properties for our migration task, we have chosen the IDE Eclipse with the plugin FeatureIDE for variability support. FeatureIDE supports the complete development cycle of software product lines including the modeling step, as well as different implementation techniques, such as annotation-based and composition-based approaches (i.e., also FOP) [26]. Therefore, besides a hybrid solution of annotation- and composition-based approaches, it fulfills all theoretical requirements of our migration concept. In detail, FeatureIDE integrates a set of command line tools that support different implementation techniques. In our application, we extend the command line tool FeatureHouse which implements FOP based on superimposition as introduced in Section II-B [10]. In the following, we describe for each step of the migration process some of the relevant technical decisions and the tool-specific steps that have to be performed. We focus on the description of the applicability of the process in general, as well as on the practicability of the specific tools we have used.

### (1) Enabling Automation – Integration into Eclipse

As described before, to enable automation of the build and testing process for Berkeley DB, we use Eclipse because we plan to use FeatureIDE during the subsequent steps. In detail, we create a new Eclipse project for the programming language C and include our case study Berkeley DB with all files and additional material. For the automated generation and execution of a specific variant, we administrate further Eclipse settings so that we ensure a correct behavior of Berkeley DB (i.e., we do not consider variability management so far). In detail, we use the C and C++ Integrated Development Environment (CDT) of Eclipse for the configuration and apply the subsequent make process. In our case, this step took several hours. But the necessary time for this step strongly depends on the complexity of the application and on the intended test mechanism that should ensure the correctness of the system. Even if this step may take longer for other projects, it has to be done only once for the complete migration process.

### (2) Introducing Systematic Variability Management – Using FeatureIDE

The next step is to introduce systematic variability management for this Eclipse Berkeley DB project. Using FeatureIDE, we can automatically convert the project into a FeatureIDE project, by adding specific configuration properties. As result, it is possible to automate the configuration, generation and execution for each variant of the product line. A central part of this step is to make the variability explicit. Therefore, we need to create a feature model in which we define configuration options in terms of features and their dependencies. This task was easy for us, because Berkeley DB was studied several times and, thus, we reused information of a previous study and selected 10 out of 28 preprocessor variables as configurable features. As result, we use FeatureIDE to create a feature model of Berkeley DB that consists of one root feature and ten optional child features (i.e., they can be combined arbitrarily to variants). Afterwards, we can use FeatureIDE's configuration editor to specify the specific variants of Berkeley DB that we want to create. If the build process of a project is straight forward, FeatureIDE already supports direct compilation and execution. However, in the case of Berkeley DB, the build process involves a specific configuration process. Thus, FeatureIDE needs to start the configuration and make process for which we developed an extension to bridge the gap. Because of our experience in plugin development, it was easy to create a first prototype for our needs. Therefore, this step takes only a few days for us. By contrast, without our experience and without knowledge of the available variability options, this step can also take several weeks. Furthermore, the step can be extended arbitrarily, this depends on the tool support that a company needs. For instance, it is also possible to develop IDE views to give a code outline or editors with specialized visualization techniques to highlight variability. As explained above, we have decided to use FeatureIDE, but there are other tools for variability management that could be used, depending on project-specific requirements [27]. The time that is required for this step depends largely on project-specific infrastructure, but in general this step has to be performed only once per migration process

### (3) Initialization of the Composition Mechanism – Using FeatureC

The main advantage of FeatureIDE is that the plugin also supports other implementation techniques. Thus, in this step, we can change the implementation strategy that is used by FeatureIDE. However, no existing tool supports the required hybrid approach. Thus, we have developed FeatureC, a feature-oriented extension of C that additionally supports the use of preprocessors. Therefore, we have extended the existing composition tool FeatureHouse. As mentioned above, FeatureHouse is a command line tool for FOP in which different languages can be integrated using a specialized grammar [10]. On the one hand, this grammar is needed to parse a specific programming language. On the other hand, the grammar encodes the composition rules for this specific language. For a hybrid solution, the already existing C grammar of FeatureHouse has to additionally support annotations. Therefore, we developed the extension FeatureC. With FeatureC, we can create one feature module in which all the existing annotation-based source code is located. Afterwards, we can use the same procedure to configure, compile and execute a specific variant.

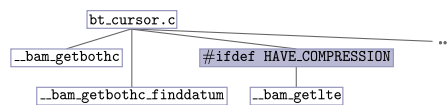


Figure 5. Required grammatical changes to support annotations in feature modules - illustrated by using the example of Figure 1.

For our case study Berkeley DB, several grammar changes were necessary. In detail, we adapted the grammar in a way that directives outside of a function are represented as nonterminal. This change allows us to parse the preprocessor annotations and to produce an output in which this annotation still exists. We depict this fundamental change in Figure 5. Here, we depict the structural representation of the file given in Figure 1 that can be combined with the corresponding nodes of another feature.

By contrast to the mentioned grammatical change, in some cases we were able to avoid grammatical changes through source-code disciplines (see next section for detailed information). However, in hindsight, we spent the most time on grammar changes and code disciplinination. All other parts of this step were partly straight forward. In sum, the complete step required several weeks; however it is largely dependent on the project-specific languages as well as tools, and is independent from the actual size of the project or the number of features to be extracted.

#### (4) Step-wise Migration – Application on Berkeley DB

In this step, the actual migration procedure can start. As stated in the previous section, each refactoring consists of three sub-steps. For our case study, we used a manual approach to extract two features (HAVE\_HASH and HAVE\_HEAP) with a focus on the identification of existing challenges. The identified project-specific challenges are discussed in the next section.

While we have used a manual approach, we identified the need for further tool support. We give a short overview on approaches that could aid the migration procedure based on our experience with the manual approach. In future work, we plan to investigate the integration of selected tools. For code analysis, several tools exist that consider variability [28]. On the one hand, it would be possible to use tools that aid the developer to identify potential code fragments representing features. On the other hand, code analysis can be used to ensure the correctness of each sub step. With Morpheus, Liebig et al. provide a promising tool for automated refactoring of C code that can cope with preprocessor directives [29]. Kästner et al. provide a concept for automatable refactorings that can be used to aid the migration from annotation-based to composition-based implementations [30]. The approach requires specific disciplined annotations and supports a subset of Java, for which it can be guaranteed that the refactorings can be performed correctly. It should be investigated to which extent this approach can be applied to real-world systems in other programming languages, such as C, for which preprocessor directives already exist. Furthermore, there are several approaches to transform preprocessor-based implementations into aspect-oriented implementations [31][32]. While we used feature-oriented programming for our case study, our general process can also be applied to a migration to an aspect-oriented

product line. Therefore, this line of research may be interesting for the practical application.

## V. PROJECT-SPECIFIC CHALLENGES

In this section, we discuss details of our experiences and challenges during the application of our migration concept to two features of Berkeley DB. Therefore, we present several insights regarding the application of concepts and discuss project-specific challenges regarding the tools and languages used for the migration of Berkeley DB.

### A. Interdependence of Process Steps

In our proposed tool-driven migration concept, we propose multiple steps to achieve an environment in which a step-wise and fine-grained migration can be applied to an annotation-based product line. As a result, in Step 4, it should be possible to refactor the code in a fine-grained manner so that we can introduce a composition-based approach in which each intermediate step presents a fully-functional hybrid solution. As a result, the transformation process is much more predictable and less risky, because it is not necessary to perform the complete migration process in one atomic step - avoiding expensive parallel development or delays. Concepts to cope with the interdependency of the individual process steps are required.

Considering the migration of Berkeley DB, we found out that a revision of a previous step can help to ease the current step. For instance, after we started to refactor Berkeley DB, we identified several tool-driven, as well as conceptual challenges that required changes to the initialization phase of Step 3. In detail, it could be beneficial to change the language support of FeatureC instead of adapting Berkeley DB in an awkward manner. This problem is strongly related to our next remarks.

### B. Undisciplined Use of Preprocessors

We started by using existing tools for the variability management of our case study. However, it turned out that the language support was not sufficient. In particular, the composition mechanism for FOP (i.e., FeatureHouse) did not support the hybrid strategy in which annotations are completely supported in compositions. Therefore, we introduced FeatureC with some grammatical changes so that it is possible to use FeatureHouse in a straight-forward manner.

Besides the discussed grammar change of Figure 5, we realized further adaptations of the underlying C grammar. However, it was possible to avoid some grammar changes because their necessity was the result of an undisciplined usage of preprocessor annotations. Therefore, we restricted the usage of annotations to certain useful patterns. In Figure 6, we show an example of such a problematic annotation that we have found in the source code of Berkeley DB. On the one hand, the restriction to disciplined preprocessor annotations is a promising approach to increase the understandability of the code. On the other hand, it would have been technically difficult to implement an approach flexible enough for all cases, while still taking advantage of the existing tool infrastructure. As a result, we had to refactor the code to introduce the desired preprocessor discipline. In general, it is advisable to consider the introduction of preprocessor discipline in concert with providing the necessary language support for the related tools.




---

```

1  __db_errx(env, DB_STR_A("0153",
2  "s(u): host lookup failed: s","s u s"),
3  nodename == NULL ? "" : nodename, port,
4#ifdef DB_WIN32
5  gai_strerrorA(ret));
6#else
7  gai_strerror(ret));
8#endif

```

---



```

9#ifdef DB_WIN32
10 __db_errx(env, DB_STR_A("0153",
11 "s(u): host lookup failed: s","s u s"),
12 nodename == NULL ? "" : nodename, port,
13 gai_strerrorA(ret));
14#else
15 __db_errx(env, DB_STR_A("0153",
16 "s(u): host lookup failed: s","s u s"),
17 nodename == NULL ? "" : nodename, port,
18 gai_strerror(ret));
19#endif

```

---

Figure 6. Disciplining of annotation-based approaches.

### C. Variability in the Presence of Scope-Sensitive Statements

In our practical application of our migration concept, we adapted existing tool support for FOP to cope with our specific language requirements, i.e., for C code with preprocessor annotations. The migration process to introduce FOP involves the creation of so-called hook functions, which are a mechanism to add code by a refinement in the middle of the function. The reason is that when only a small part of a function is variable, it is often useful to extract this part into a hook function, enabling more fine-grained refinements. While this process may often be straight forward, it may require modifications that may introduce errors. In particular, we experienced that statements that depend on the current scope are potentially problematic in Berkeley DB for instance `goto`, `switch`, and `return` statements. In Table I, we summarize how often these statements exist in the complete code, as well as how often they are part of variable code and compare these values with all cases in which their usage results in a problem for all identified features, selected features, and our two extracted features, `HAVE_HASH` and `HAVE_HEAP`.

#### *goto*-Statements

In Figure 7, we show an example of a `goto`-statement of Berkeley DB. Assume, we extract the code related to Feature `HAVE_HASH` (Line 6-12) into a hook function without further considerations. The result would be that the `goto`-statement from Line 9 operates in a new scope and, thus, the corresponding `goto`-label would be out-of-scope, and lead to an error. In the complete code of Berkeley DB we count 4642 occurrences of `goto` statements, whereas only 635 interact with variable code. 20% of these occurrences result in a potential problem considering all features. Several strategies exist to solve this situation. Depending on the exact occurrences of the problem, we have to select the most promising strategy that could lead to new required tool support.

The straight-forward solution is to extract the complete method into the `HAVE_HASH` feature module. This would lead to code clones and diminish the benefit of the modularization. In some cases, it is possible to create a hook method in which we insert the considered variable code part including the code artifacts behind the `goto`-label. This is often possible in case of ordinary error handling. However, this might require further

modifications, such as a huge parameter list, in which the current state of all variables in the scope of the `goto`-label must be preserved. A third solution considers a more conceptual mechanism of inlining, which allows us to extract only the variable code into a feature module also with the `goto`-statement. Through the generation process of the combined code, it is possible to achieve a code artifact in which the scope is again in the correct scope and works correctly. However, this solution separates the `goto`-statement from the label and, thus, breaks the convention regarding this mechanism.

#### *switch*-Environment

In Figure 7, we illustrate a `switch` construct that is problematic and cannot be treated in a straight-forward manner. In detail, it is not possible to extract the complete case into a hook method using a 1:1 extraction of the variable parts. Similar to the `goto`-statement, the scope will change and the case cannot be handled. In Table I, we can see that Berkeley DB includes 429 `switch` statements, whereas 16% are involved in variable code artifacts. Only half of them, exactly 36, result in a case that is not straight-forward manageable. As the problematic statements are relatively few, an expensive adaption of the tool-support may not be advisable in this case. Nevertheless, we discuss multiple solution strategies.

Similar to the `goto`-statement, multiple solution strategies exist for this example. First, it is possible to extract the complete `switch` into a feature module. However, this is not a viable option if the number of the used variables and, thus, needed parameters is too high. A second option could be a hook method inside of the case. For this, it must be ensured that the value `DB_HASH` (cf. Line 7) is actually defined and this could be error-prone. Third, we also can change the composer strategy so that, for instance, a new keyword allows us to refine the case on this specific line. However, special knowledge is needed to apply such grammar extensions.

#### *return*-Statements

Besides the previous problems, we also identified the extraction of `return` statements to hook functions as a challenge. Again, if we extract corresponding code artifacts, we change the scope of these statements. However, the problem occurs if a `return` statement is only defined for some cases and not for all, such as could appear in an `if-else` case in which only one case contains a `return`. In the code of Berkeley DB, there are 7779 `return` statements while only 1254 are part of variable code. Again, 20% of these variable cases lead to a potential problem that we can solve in multiple ways.

One solution is an additional parameter of the hook function that indicates whether we should utilize the `return` statement. Afterwards, the caller side needs to evaluate this additional parameter. By contrast to the first solution, in some cases, we can directly insert a new `return` statement for our purpose. In detail, if we know that, for instance, all original `return` statements result in a positive integer, we can also use a negative integer to indicate that a `return` is not valid. Furthermore, we can also use a grammar adaption as third solution. In this case, it is necessary to create a kind of an inlining. Nevertheless, the challenge is that a new function needs a `return` statement in all cases to be compatible to the signature. In the grammar-adaption solution, this can be solved with an additional keyword.

```

1 //db/partition.c
2 switch (new_dbc->dbtype) {
3     case DB_BTREE:
4         ...
5         break;
6 #ifdef HAVE_HASH
7     case DB_HASH:
8         if ((ret = __ham_stat(new_dbc, &hsp, flags)) != 0)
9             goto err;
10        ...
11        break;
12 #endif
13     default:
14         break;
15 }

```

Figure 7. Example of potential problems to extract hook methods.

## VI. RELATED WORK

The idea to combine annotation-based with composition-based implementation techniques to facilitate a step-wise migration process has been formulated by Kästner and Apel [9]. In our work, we consider a similar concept and investigate its application to a real-world project with a focus on technical concerns and challenges that must be considered to prepare and perform the actual migration process.

Researchers have investigated advantages and disadvantages of many different techniques to implement variability for product lines [3], and proposed several possible combinations. Aspectual Feature Modules have been proposed to combine the advantages of two different composition-based approaches: aspect-oriented programming and feature-oriented programming [33]. In contrast, we focus on a combination of annotation-based and composition-based approaches to enable the possibility of a step-wise migration of the prevalent preprocessor-based product lines in practice. Similarly, the choice calculus has been proposed to provide a formal basis for the combination of annotation-based and composition-based approaches to combine their benefits [34]. However, the potential application to migration has not been discussed. In order to improve the practicability of approaches that combine annotation-based with composition-based implementation, Behringer proposes a concept to improve the visualization of the underlying representations [35]. This complements our work, in the sense that it could be used to ease the practical application of the migration process.

Rosenmüller et al. also extracted feature modules from an implementation of Berkeley DB [36]. Complementary to our experiences, they have used a step-wise process in which they refactored the C code into C++, and transformed the resulting object-oriented version into a feature-oriented version of C++. In contrast, we have focused on a direct application of FOP to avoid long and costly refactorings and transformations with the goal to minimize the risk of the overall process. Furthermore, they identified the use of local variables as a potential problem that must be considered when extracting hook methods.

Alves et al. present a case study in which they migrated a real-world product line from an annotation-based to an aspect-oriented implementation [37]. They identify several language-specific strategies to transform certain variability patterns in the code into aspects. For the specific case of aspect-oriented programming this work provides complementary insights that can be used as a guide to perform the actual transformation, i.e., Step 4 in our proposed process.

TABLE I. OCCURRENCES OF POTENTIAL PROBLEMS IN BERKELEY DB.

	Complete Code	Variable Code	Number of Problems in		
			All Features	Selected Features	Extracted Features
#switch	429	71	36	19	11
#goto	4642	634	127	58	35
#return	7779	1254	248	34	9

## VII. CONCLUSION AND FUTURE WORK

Software product line engineering provides different development approaches. Whereas annotation-based approaches are mainly used in industry, compositional approaches promise advantages, such as eased maintainability. However, if a company plans to transform an existing annotation-based into a composition-based product line, a wide range of challenges exist. Whereas complete refactorings are error-prone and can take a lot of time, existing work proposed to use a hybrid solution that combines both approaches. We build on this idea and propose a detailed instantiation that allows us to transform the system in a step-wise manner. This tool-driven concept is based on intermediate results and ensures that the complete product line is in a consistent state at all times. To exemplify the application of our approach, we developed FeatureC, a hybrid implementation technique, and applied it to Berkeley DB as a case study. We identified challenges regarding the composition-based concept as well as the granularity of modularizations. In future work, we plan to apply our migration concept to further case studies. Furthermore, we aim to use the concept for projects of our industrial partners.

## ACKNOWLEDGMENTS

This work is partially funded by the BMBF project NaVaS (grant number 01IS14017A and 01IS14017B).

## REFERENCES

- [1] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Boston, MA, USA: Addison-Wesley, 2001.
- [2] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Berlin, Heidelberg: Springer, 2005.
- [3] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines: Concepts and Implementation*. Berlin, Heidelberg: Springer, 2013.
- [4] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, and G. Saake, "Do Background Colors Improve Program Comprehension in the #Ifdef Hell?" *Empirical Software Engineering (EMSE)*, vol. 18, no. 4, 2013, pp. 699–745.
- [5] D. Le, E. Walkingshaw, and M. Erwig, "#ifdef Confirmed Harmful: Promoting Understandable Software Variation," in *Proceedings of the International Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Washington, DC, USA: IEEE Computer Science, 2011, pp. 143–150.
- [6] C. Prehofer, "Feature-Oriented Programming: A Fresh Look at Objects," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Berlin, Heidelberg: Springer, 1997, pp. 419–443.
- [7] D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling Step-Wise Refinement," *IEEE Transactions on Software Engineering (TSE)*, vol. 30, no. 6, 2004, pp. 355–371.
- [8] P. Clements and C. Krueger, "Point/Counterpoint: Being Proactive Pays Off/Eliminating the Adoption Barrier," *IEEE Software*, vol. 19, no. 4, 2002, pp. 28–31.

- [9] C. Kästner and S. Apel, "Integrating Compositional and Annotative Approaches for Product Line Engineering," in Proceedings of the GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLe). Passau, Germany: Department of Informatics and Mathematics, University of Passau, 2008, pp. 35–40.
- [10] S. Apel, C. Kästner, and C. Lengauer, "Language-Independent and Automated Software Composition: The FeatureHouse Experience," IEEE Transactions on Software Engineering (TSE), vol. 39, no. 1, Jan. 2013, pp. 63–79.
- [11] I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, and K. Villela, "Software Diversity: State of the Art and Perspectives," International Journal on Software Tools for Technology Transfer (STTT), vol. 14, 2012, pp. 477–495.
- [12] F. Benduhn, T. Thüm, M. Lochau, T. Leich, and G. Saake, "A Survey on Modeling Techniques for Formal Behavioral Verification of Software Product Lines," in Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS). New York, NY, USA: ACM, 2015, pp. 80:80–80:87.
- [13] M. Alférez, R. Bonifácio, L. Teixeira, P. Accioly, U. Kulesza, A. Moreira, J. a. Araújo, and P. Borba, "Evaluating Scenario-Based SPL Requirements Approaches: The Case for Modularity, Stability and Expressiveness," Requirements Engineering, vol. 19, no. 4, 2014, pp. 1–22.
- [14] GCC Development Team, "The C Preprocessor," Website, 2015, available online at <http://gcc.gnu.org/onlinedocs/cpp/index.html>; visited on October 29th, 2015.
- [15] Munge Development Team, "Munge: A Purposely-Simple Java Preprocessor," Website, 2015, available online at <http://github.com/sonatype/munge-maven-plugin>; visited on October 29th, 2015.
- [16] J. Pleumann, O. Yadan, and E. Wetterberg, "Antenna: An Ant-to-End Solution For Wireless Java," Website, 2015, available online at <http://antenna.sourceforge.net/>; visited on October 29th, 2015.
- [17] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines," in Proceedings of the International Conference on Software Engineering (ICSE). Washington, DC, USA: IEEE Computer Science, 2010, pp. 105–114.
- [18] C. Kästner and S. Apel, "Virtual Separation of Concerns – A Second Chance for Preprocessors," Journal of Object Technology (JOT), vol. 8, no. 6, 2009, pp. 59–78.
- [19] Big Lever Software Inc., "Gears: A Software Product Line Engineering Tool," Website, 2015, available online at <http://www.biglever.com/solution/product.html>; visited on October 29th, 2015.
- [20] pure::systems, "pure::variants," Website, 2015, available online at [http://www.pure-systems.com/pure\\_variants.49.0.html](http://www.pure-systems.com/pure_variants.49.0.html); visited on October 29th, 2015.
- [21] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Longtier, and J. Irwin, "Aspect-Oriented Programming," in Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Berlin, Heidelberg: Springer, 1997, pp. 220–242.
- [22] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella, "Delta-Oriented Programming of Software Product Lines," in Proceedings of the International Software Product Line Conference (SPLC). Berlin, Heidelberg: Springer, 2010, pp. 77–91.
- [23] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr., "N Degrees of Separation: Multi-Dimensional Separation of Concerns," in Proceedings of the International Conference on Software Engineering (ICSE). New York, NY, USA: ACM, 1999, pp. 107–119.
- [24] Y. Smaragdakis and D. Batory, "Implementing Layered Designs with Mixin Layers," in Proceedings of the European Conference on Object-Oriented Programming (ECOOP). London, UK: Springer, 1998, pp. 550–570.
- [25] A. Bergel, S. Ducasse, and O. Nierstrasz, "Classbox/J: Controlling the Scope of Change in Java," in Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA). New York, NY, USA: ACM, 2005, pp. 177–189.
- [26] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich, "FeatureIDE: An Extensible Framework for Feature-Oriented Software Development," Science of Computer Programming (SCP), vol. 79, no. 0, 2014, pp. 70–85.
- [27] J. Pereira, K. Constantino, and E. Figueiredo, "A Systematic Literature Review of Software Product Line Management Tools," in Software Reuse for Dynamic Systems in the Cloud and Beyond, ser. Lecture Notes in Computer Science. Springer, 2014, vol. 8919, pp. 73–89.
- [28] J. Meinicke, T. Thüm, R. Schöter, F. Benduhn, and G. Saake, "An Overview on Analysis Tools for Software Product Lines." New York, NY, USA: ACM, 2014, pp. 94–101.
- [29] J. Liebig, A. Janker, F. Garbe, S. Apel, and C. Lengauer, "Morpheus: Variability-aware Refactoring in the Wild," in Proceedings of the International Conference on Software Engineering (ICSE). Piscataway, NJ, USA: IEEE Computer Science, 2015, pp. 380–391.
- [30] C. Kästner, S. Apel, and M. Kuhlemann, "A Model of Refactoring Physically and Virtually Separated Features," in Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE). New York, NY, USA: ACM, 2009, pp. 157–166.
- [31] B. Adams, W. De Meuter, H. Tromp, and A. E. Hassan, "Can We Refactor Conditional Compilation into Aspects?" in Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD). New York, NY, USA: ACM, 2009, pp. 243–254.
- [32] A. Reynolds, M. E. Fuczynski, and R. Grimm, "On the Feasibility of an AOSD Approach to Linux Kernel Extensions," in Proceedings of the AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software. New York, NY, USA: ACM, 2008, pp. 8:1–8:7.
- [33] S. Apel, T. Leich, and G. Saake, "Aspectual Feature Modules," IEEE Transactions on Software Engineering (TSE), vol. 34, no. 2, 2008, pp. 162–180.
- [34] E. Walkingshaw and M. Erwig, "A Calculus for Modeling and Implementing Variation," in Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE). New York, NY, USA: ACM, 2012, pp. 132–140.
- [35] B. Behringer, "Integrating Approaches for Feature Implementation," in Proceedings of the International Symposium Foundations of Software Engineering (FSE). New York, NY, USA: ACM, 2014, pp. 775–778.
- [36] M. Rosenmüller, S. Apel, T. Leich, and G. Saake, "Tailor-made data management for embedded systems: A case study on Berkeley DB," Data and Knowledge Engineering, vol. 68, no. 12, 2009, pp. 1493–1512.
- [37] V. Alves, A. Costa Neto, S. Soares, G. Santos, F. Calheiros, V. Nepomuceno, D. Pires, J. Leal, and P. Borba, "From Conditional Compilation to Aspects: A Case Study in Software Product Lines Migration," in Workshop on Aspect-Oriented Product Line Engineering (AOPL), 2006.