On the Impact of the Optional Feature Problem: Analysis and Case Studies

Christian Kästner[1], Sven Apel[2], Syed Saif ur Rahman[1], Marko Rosenmüller[1], Don Batory[3], and Gunter Saake[1]

[1] *School of Computer Science, University of Magdeburg, Germany, {ckaestne,srahman,rosenmue,saake}@ovgu.de*
[2] *Department of Informatics and Mathematics, University of Passau, Germany, apel@uni-passau.de*
[3] *Department of Computer Sciences, University of Texas at Austin, USA, batory@cs.utexas.edu*

Abstract

A software product line *is a family of related programs that are distinguished in terms of features. A* feature *implements a stakeholders' requirement. Different program variants specified by distinct feature selections are produced from a common code base. The* optional feature problem *describes a common mismatch between variability intended in the domain and dependencies in the implementation. When this situation occurs, some variants that are valid in the domain cannot be produced due to implementation issues. There are many different solutions to the optional feature problem, but they all suffer from drawbacks such as reduced variability, increased development effort, reduced efficiency, or reduced source code quality. We examine the impact of the optional feature problem in two case studies from the domain of embedded database systems, and we survey different state-of-the-art solutions and their trade-offs. Our intension is to raise awareness of the problem, to guide developers in selecting an appropriate solution for their product line, and to identify opportunities for future research.*

1. Introduction

Software product line (SPL) engineering is an efficient and cost-effective approach to produce a family of related program variants for a domain [1], [2], [3]. SPL development starts with an analysis of the domain to identify commonalities and differences between the programs of the product line, which are described as features in a feature model [4]. There are many different ways to implement SPLs; we focus on those in which common implementation artifacts are produced and a *variant* (a.k.a. product line member) can be automatically *generated* from those artifacts without further implementation effort based on the selected features for this product. This includes implementation approaches based on framework and plug-in architectures [5], generative programming [3], [6], preprocessors [2], [7], and various forms of aspects and feature modules [8], [9], [10], [11].

In this paper, we address one particular problem of SPL development: the *optional feature problem*. It is fundamental to SPL development and occurs when there are two features in a domain that are both optional, but the implementations of both features are not independent. For example, consider an SPL of embedded *database management systems (DBMS)*.[1] In this SPL, a feature STATISTICS is responsible for collecting statistics about the database such as buffer hit ratio, table size, cardinality, or committed transactions per second. Not every embedded DBMS will need statistics – to reduce runtime overhead and binary size, or statistics are simply not needed – therefore this feature is optional in this domain. In the same SPL, a feature TRANSACTIONS, that supports safe concurrent access (ACID properties), is also optional because it is not needed in all systems of the domain – e.g., in single user or read-only variants. Nevertheless, even though both features are optional, they cannot be implemented independently as long as feature STATISTICS collects values like "committed transactions per second". That is, the implementation of one optional feature affects or relies on the implementation of another; features that appear to be independent in the domain are not in their implementation.

There are different solutions to the optional feature problem, but all have one or the other drawback, leading to a fundamental trade-off in SPL development: Solutions either reduce variability (i.e., they impose limitations on possible feature combinations such that certain variants like "a database with statistics but without transactions" can no longer be generated), increase development effort, deteriorate program performance and binary size, or decrease code quality. For example, an intuitive solution would be to use multiple implementations of the statistics feature, one that is used when transactions are selected and another that is used without transactions. However, this solution increases development effort and leads to code replication. Another solution is to use additional modules to untangle the conflicting implementation as used in [15], [16], [8], [17]. This approach has been formalized with an underlying theory by Liu et al. [16] as *derivatives*, but concerns have been raised in discussions at ICSE'06 whether derivatives scale in practice. However, so far there is only little empirical insight into the optional feature problem regarding (a) its impact (how often does it occur) and (b) the suitability of different solutions in practice.

1. Throughout this paper, we use the domain of embedded DBMS because it is well-understood and applying SPL technology promises benefits for embedded DBMS since functionality must be carefully tailored to specific tasks and scarce resources on embedded devices [12], [13], [14].

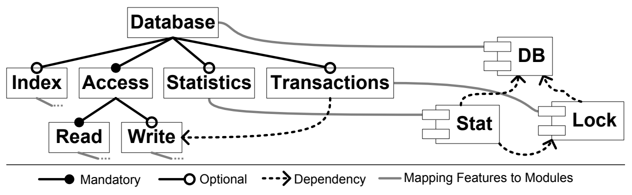

Figure 1. Excerpt of the feature model (left) and implementation model (right) of a DBMS SPL.

We survey the state-of-the-art techniques and trade-offs in solving the optional feature problem. We deduce recommendations for their application and report experience from two exploratory case studies in the domain of embedded DMBS. In both studies, the impact of the optional feature problem was severe: We found more pairs of implementation modules that exhibit the optional feature problem than there are features in the SPL. Depending on the solution used, this drastically reduced (almost entirely eliminated) variability, increased binary size (by up to 15 %), or increased development effort (days instead of hours). We found that there is no optimal solution; therefore we discuss a trade-off between solutions to assist developers with choosing the best solution in the context of their SPL.

2. The Optional Feature Problem

The optional feature problem occurs if two (or more) optional features are independent in a domain, but are not independent in their implementation. For a closer analysis, we first distinguish between feature model and implementation model in SPL development.

A *feature model* (a.k.a. domain model or product line variability model) describes the features of a domain or SPL and their relationships. That is, it defines the scope of the SPL and the domain's variability: it specifies which products of the SPL are conceptually meaningful and desired [4], [3]. A common way to describe a feature model is with a feature diagram, as shown for our database in Figure 1 (left), but other representations are possible. Dependencies between features in a feature model are called *domain dependencies* and are independent of the features' implementations. In our database example, TRANSACTIONS requires WRITE because there is no use for transactions in a read-only database. However, there is no domain dependency between STATISTICS and TRANSACTIONS, both features make sense in isolation and in combination.

While the feature model describes domain concepts, implementation modules (such as components, plug-ins, aspects, or feature modules) and their relationships are described in an *implementation model* (a.k.a. software variability model or family model) [2]. Domain model and implementation model are linked, so that for a given feature selection the according implementation modules can be composed. There are many different notations for implementation models, in Figure 1 (right) we use a simple component diagram for illustration. In this example, the implementation of statistics references or extends code from the implementation of transactions to collect values as "committed transactions per second", as explained above. We call a dependency between implementation modules an *implementation dependency*.[2]

The optional feature problem arises from a mismatch between feature model and implementation model. In our example, features STATISTICS and TRANSACTIONS are optional and independent, but there is a dependency in their implementation. Although, considering the feature model, it appears that we can create a variant with STATISTICS but without TRANSACTIONS, an implementation dependency prevents this.

The optional feature problem has interesting implications. First, if the feature model describes that both features can be selected independently (assuming that the feature model correctly describes the domain), we should be able to find an implementation for each variant; the difficulty lies in generating those variants from reusable implementation modules. Second, an implementation dependency between two features is usually not accidental. If both features are selected, there should be some additional behavior. In our example, it is intended that STATISTICS collects statistics about TRANSACTIONS if both features are selected, but both features should also work in isolation. This fine-grained adjustment in behavior does not justify adding another feature like 'STATISTICS ON TRANSACTIONS' to the feature model – it is an implementation concern.

For further illustration, consider a second example: an optional feature WRITE in the database allows us to generate read-only variants (by removing feature WRITE). However, the implementation of WRITE cuts across the entire database implementation and many features. For example, indexes require a different implementation, so do the buffer manager, statistics, and many other features. Once feature WRITE is selected, it has an impact on numerous features and causes many implementation dependencies. Finding the right implementation mechanism so that read-write and read-only variants with and without indexes or statistics can all be generated efficiently remains the challenge for the remainder of this paper.

3. An Analysis of Standard Solutions

When developers find the optional feature problem in their code (manually or with tool support as in [18]), i.e., when they find an implementation dependency that is not covered

2. We assume that the implementation model correctly describes implementation modules and includes all implementation dependencies. Detecting a mismatch between implementation model and implementation modules is a separate research problem outside the scope of this paper (see related work in Sec. 6).

by a domain dependency, they need to decide how to handle it. They can either (1) keep the implementation dependency or (2) eliminate it by changing the implementation. In the following, we discuss both approaches and their implications.

3.1. Keeping the Implementation Dependency

The first solution is simple, because we do not need to change anything. We can simply acknowledge the existence of the implementation dependency and accept that it prevents us from generating certain variants like a database with STATISTICS but without TRANSACTIONS.[3]

While this solution has the advantage that it is not necessary to take any action, it reduces the variability of the SPL (or at least acknowledges the reduced variability) compared to what *should* be possible in the domain. Depending on the importance of the excluded variants – that are possible in the domain, but not with the current implementation – the reduced variability can have a serious impact on the strategic value of the SPL.

3.2. Changing Feature Implementations

Instead of reducing variability by acknowledging an implementation dependency, we can eliminate the dependency by changing the implementations of the features involved. There are different state-of-the-art approaches which we discuss separately.

Changing Behavior to Enforce Orthogonality. To eliminate an implementation dependency, we can change the behavior of one (or more) variants such that the involved features can be implemented without dependency. In our example, it may be acceptable to never collect statistics about transactions. Although this alters the behavior of a variant with STATISTICS and TRANSACTIONS against the original intensions, it eliminates the implementation dependency.

Unfortunately, this solution is only possible in rare cases. As discussed above, the additional behavior when both features are selected is usually intended. Omitting this behavior can lead to variants with unsatisfactory, surprising, or even incorrect results. In many cases, it is difficult to think of reasonable changes that would lead to orthogonal implementations in the first place.

Moving Code. In some cases it is possible to move some code of the implementation of one feature to another feature. For example, we could move the code for collecting and querying statistics on transactions to the implementation of feature TRANSACTIONS. This way, a piece of the functionality of statistics is included in every variant with TRANSACTIONS, independently of whether STATISTICS is selected. When TRANSACTIONS is selected without STATISTICS some of the statistics code is included unnecessarily but simply never called by other implementation modules.

While this approach is often feasible, it has two drawbacks. First, this violates the principle of *Separation of Concerns (SoC)* [19], [8], since we move code into implementation modules where it does not belong, only for technical reasons (see also related work in Sec. 6). Second, some variants contain unnecessary code, which can lead to inefficient programs regarding both binary size and performance (overhead for collecting statistics that are never queried). Whether this is acceptable depends on the particular SPL.

Multiple Implementations per Feature. A third solution, without the drawbacks of the previous two, is to provide different implementations of a feature, depending on the selection of other features. In our example, we could have two implementations of feature STATISTICS: one that collects statistics on transactions and one that does not. Which of these implementations is select during generation depends on the presence of feature TRANSACTIONS.

While this approach allows us to generate all variants, there are two main problems: code replication and scaling. The alternative implementations usually share a significant amount of code. Furthermore, a feature with implementation dependencies to n optional features would require 2^n different implementations to cover all variants.

Conditional Compilation. A very different solution is not to modularize the feature implementation, but use preprocessors or similar tools for conditional compilation. This way, it is possible to surround the code that collects statistics about transactions with directives like '*#ifdef TXN*' and '*#endif*'. We call such preprocessor statements regarding features 'annotations'. During generation, annotations are evaluated and – in our example – if feature TRANSACTION is not selected, the according code is removed before compilation. Nested *#ifdef* directives can be used to annotate code that is included only when two or more features are selected. Conditional compilation is a general implementation mechanism for SPLs [2], [7] or can be used inside modular implementation mechanisms as plug-ins, components, feature modules, or aspects [20].

Although this solution can implement all variants without code replication or unnecessary code, conditional compilation is heavily criticized in literature as undisciplined approach that violates separation of concerns and modularity and that can easily introduce hard-to-find errors [21], [22], [2].

Refactoring Derivatives. Finally, there is a general refactoring to remove implementation dependencies

3. Depending on the technique used for modeling and reasoning, this may require to change the feature model. Changing the feature model can blur the separation of domain and implementation concerns, therefore it is recommendable to at least document which dependencies in the feature model represent implementation issues instead of domain issues.

Solution	variability	additional effort	binary size & performance	code quality
Ignore Implementation Dep.	less variants (– –)	no effort (++)	efficient (++)	no impact (++)
Changing Behavior	all variants, when possible (+)	little to no effort (+)	potentially inefficient (–)	no impact (++)
Moving Code	all variants (++)	little effort (+)	inefficient (– –)	reduced SoC (–)
Multiple Implementations	all variants (++)	very high effort (– –)	efficient (++)	high replication (– –)
Conditional Compilation	all variants (++)	little to medium effort (+)	efficient (++)	no modularity & SoC (– –)
Derivatives	all variants (++)	high effort (–)	efficient (++)	modular but many modules (+)

Table 1. Comparison of solutions to the optional feature problem

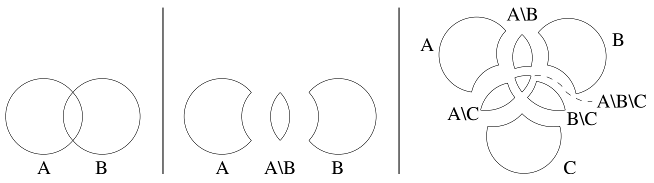

Figure 2. Illustration of derivative modules.

while keeping modular implementations, described by Prehofer [15] and formalized by Liu et al. [16]. The idea is to extract the code responsible for the dependency into a separate module. This solution consists of two steps: first, the responsible code is removed from both features' implementation so that both are orthogonal; second, this code is reintroduced as a new implementation module called *derivative module*. The derivative module is included in the generation process to restore the original behavior if and only if both original implementation modules are selected. In our example, the derivative would contain the code to collect statistics about transactions. Basically, a derivative contains the code that would be located in nested annotations in the conditional compilation solution. Derivative modules represent implementation concepts and not domain concepts, therefore they are not included in the feature model but included in the generation process automatically [16].

We illustrate this idea in Figure 2: The code that binds both features A and B together is separated into a new derivative module named $A\backslash B$. With this refactoring, it is possible to compose both features in isolation, but also in combination together with the new derivative module. If one implementation module has multiple implementation dependencies, we need a derivative module for each dependency, thus there are n derivative modules for n dependencies, instead of 2^n different implementations as in the "multiple implementations" solution above. If more than two features interact at the same time, this can be resolved with *higher-order derivative modules* (like $A\backslash B\backslash C$ in Fig. 2) in the same way [16] (similar to nested preprocessor statements).

Compared to conditional compilation, derivatives allow modular implementations and enforce a strict separation of concerns (see discussions in [16], [17]). However, while derivatives can eliminate implementation dependencies, it requires a certain effort to refactor the source code.

3.3. Summary

In Table 1, we compare the different solutions with regard to four criteria: variability, effort, efficiency, and code quality. Grades ranging from ++ (best) to – – (worst) are given roughly based on the discussion of the solutions above. Variability describes the ability to realize all variants prescribed by the feature model (not possible when ignoring implementation dependencies; only sometimes possible by changing behavior). Additional effort is required by all approaches to eliminate implementation dependencies, but with varying amount; for example, refactoring derivatives is usually a more complicated process than adding *#ifdef*'s or moving code. The fourth column characterizes whether the solution can provide an efficient implementation for each variant (moving code results in unnecessary code in some variants; changing behavior implements a different behavior than intended). Finally, code quality characterizes the apparent effects of replication, reduced modularity, and separation of concerns.

As it can be seen from the table, there is no 'best' solution; all solutions have different advantages and disadvantages (only 'multiple implementations' seems to be always weaker than 'derivatives' and 'conditional compilation'). However, this table is derived only from a qualitative discussion of these solutions. In the following, we enrich it with experience from two case studies.

4. A Real Problem – Case Studies

Given the discussed solutions, it seems that developers have to decide for the lesser evil. They can either accept an implementation dependency, which reduces the variability of the SPL, or they can apply some changes to the source code accepting some drawbacks. Especially approaches that increase the effort require evaluation on whether they are applicable in practice.

To the best of our knowledge there is no study that explores the effects of the optional feature problem and its solutions on SPLs of substantial size. Especially derivative modules have been discussed almost exclusively on small academic examples, e.g., [15], [16], [17]. However, in our experience in teaching SPL development (with AspectJ [9] and AHEAD [10]) *and* in the development of DBMS SPLs [14], [23], we encountered the optional feature problem very frequently. For this reason, we analyze the impact of the optional feature problem on SPL development and explore solutions in two case studies. First, we report our experience in decomposing Berkeley DB and afterward we analyze FAME-DBMS which we designed and implemented from scratch.

4.1. Decomposition of Berkeley DB

Oracle's Berkeley DB[4] is an open source database engine (approx. 70 000 LOC) that can be embedded into applications as a library. In two independent endeavors, we decomposed both the Java and the C version of Berkeley DB into features [23], [14]. Our aim of this decomposition was to *downsize* data management solutions for embedded systems [14]. Specifically, we aimed at generating different tailored variants, especially smaller and faster variants, by making features like transactions, recovery, or certain database operations optional. The decomposition was originally performed by extracting legacy code of a feature into a separate implementation module (technically, we used aspects written in AspectJ [9] respectively feature modules written with FeatureC++ [24]). To generate a variant, we then specified which modules to include in the compilation process. We already discussed the AspectJ-specific experience of this decomposition in SPLC'07 [23]. Here, we focus on the optional feature problem that is largely independent of the programming language and that was excluded from discussions in prior work.

Impact of the Optional Feature Problem. In the Java version, we refactored 38 features, which took about one month. Almost all features are optional and there are only 16 domain dependencies; in theory, we should be able to generate 3.6 billion different variants.[5] However, implementation dependencies occurred much more often than domain dependencies: With manual and automated source code analysis, we found 53 implementation dependencies that were not covered by domain dependencies. An excerpt of features and corresponding dependencies between their implementation modules is shown in Figure 3. We select 9 features with many implementation dependencies and omit other features and dependencies due to space restrictions (implementation dependencies marked with 'x'; direction of dependencies omitted for simplicity; there are no domain dependencies between these features, since all are optional in the feature model; see [23] for a complete list of features and description). Overall, in Berkeley DB, the optional feature problem occurred between 53 pairs of features.

Ignoring all implementation dependencies is not acceptable, because this would restrict the ability to generate tailored variants drastically. In pure numbers the reduction from 3.6 billion to 0.3 million possible variants may appear

	11	12	13	20	27	29	30	31	33
11. ATOMICTRANSACT.	▨	▨	▨	▨	▨	▨	▨	▨	▨
12. FSYNC	x	▨	▨	▨	▨	▨	▨	▨	▨
13. LATCHES	x	x	▨	▨	▨	▨	▨	▨	▨
20. STATISTICS	x	x	x	▨	▨	▨	▨	▨	▨
27. INCOMPRESSOR	x	x	x	x	▨	▨	▨	▨	▨
29. DELETEDBOP.	x		x	x	x	▨	▨	▨	▨
30. TRUNCATEDBOP.	x		x			x	▨	▨	▨
31. EVICTOR			x	x		x		▨	▨
33. MEMORYBUDGET	x		x	x		x		x	▨

Figure 3. Implementation dep. in Berkeley DB (excerpt).

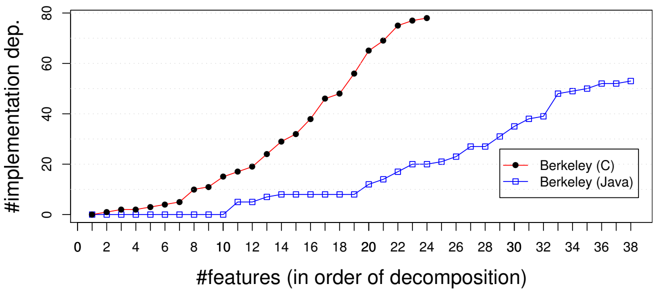

Figure 4. Number of implementation dependencies in Berkeley DB.

significant but acceptable considering that still many variants are possible. Nevertheless, having a closer look, we found that especially in the core of Berkeley DB, there are many implementation dependencies. Important features like statistics, transactions, memory management, or database operations, shown in Figure 3, must be selected in virtually all variants. This prevents many useful variants, so we have to use suboptimal variants in many scenarios, in contrast to our initial intension expressed in the feature model.

The remaining variability of 0.3 million variants is largely due to several small independent debugging, caching, and IO features. Considering all implementation dependencies, this SPL has little value for generating variants tailored to concrete use cases without statistics, without transactions, or as read-only database.

In the C version, which has a very different architecture and was independently decomposed by a different developer into a different set of features, we extracted 24 features [14]. The overall experience was similar: With only 8 domain dependencies almost 1 million variants are theoretically possible, but only 784 variants can be generated considering all 78 implementation dependencies that we found. Again, important features were de facto mandatory in every variant.

These numbers give a first insight into the impact of the optional feature problem. Furthermore, consider that with a more detailed domain analysis, we would find far more features in the domain of embedded DBMS than used in these case studies. An analysis of the number of implementation dependencies in Figure 4 shows a tendency that in an SPL with further features, many more implementations dependen-

4. http://www.oracle.com/database/berkeley-db
5. To determine the number of variants, we use the FAMA tool [25], kindly provided by D. Benavides.

cies can be expected. (Again, the numbers of the Java and the C version are not directly comparable due to different feature sets and independent development). In Berkeley DB, ignoring implementation dependencies reduces variability to a level that makes the SPL approach almost useless.

Exploring Solutions. After the analysis revealed that ignoring all implementation dependencies is not an option, we explored different solutions to eliminate implementation dependencies. Since we created the SPL from an existing application, we wanted to preserve the original behavior and, thus, discarded the solution to change behavior. Observing features with up to 9 implementation derivatives, also the solution to provide (up to 512) alternative implementations seemed not suitable. Focusing on a clean separation of concerns, we started with refactoring derivatives.

In Java, we started with 9 derivative modules to eliminate all direct implementation dependencies of the feature STATISTICS. The 9 derivative modules alone required over 200 additional pieces of advice or inter-type declarations with AspectJ. Of 1867 LOC of the statistics feature, 76 % were extracted into derivative modules (which would also be the amount of code we needed to move into different features for solution "move code"). In the C version, we refactored 19 derivatives. In both case studies, this refactoring was rather tedious and required between 15 minutes and two hours for each derivative depending on the amount of code. Due to the high effort, we refrained from refactoring all implementation dependencies into derivatives.

Next, we experimented with conditional compilation. In the C version, we directly used *#ifdef* statements inside the code of FeatureC++ modules. In the Java version, as Java has no native preprocessor, we used a preprocessor-like environment *CIDE* [7] to eliminate all implementation dependencies. Preparing the code with conditional compilation was significantly faster than refactoring derivatives since no changes to the code were necessary except introducing annotations. However, code quality suffers as there is no form of modularity, feature code is scattered and tangled (up to 300 annotated code fragments in 30 classes per feature), and there are many nested annotations up to a nesting level of 3.

In Berkeley DB, both the derivative solution and the conditional compilation solution were acceptable despite their drawbacks. While we prefer a clean separation of concerns, the required effort was overwhelming. At the current state of development a mixture of derivatives and conditional compilation appears to be a good compromise.

4.2. Design and Implementation of FAME-DBMS

The question remains whether the high number of implementation dependencies is caused by the legacy design of Berkeley DB and our subsequent refactoring or whether they are inherent in the domain and thus also appear in DBMS SPLs designed from scratch. One may assume that implementation dependencies arise solely because a legacy application was not designed with optional features in mind. With a careful redesign, we might have avoided some of these implementation dependencies. To evaluate whether implementation dependencies are 'just' a problem of legacy design, we conducted a second case study, in which we designed and implemented a database SPL called *FAME-DBMS* from scratch.

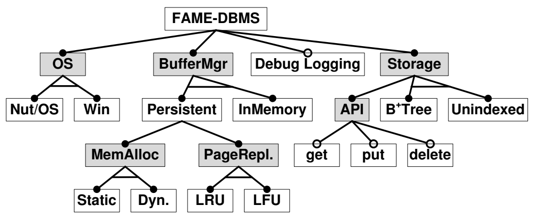

Figure 5. Initial feature model of FAME-DBMS.

FAME-DBMS is a database SPL prototype designed specifically for small embedded systems. Its goal was to show that SPL technologies can tailor data management for special tasks in even small embedded systems (e.g., BTNode with Nut/OS, 8 MHz, and 128 kB of memory). FAME-DBMS provides only essential data management mechanisms to store and retrieve data via an API. Advanced mechanisms like transactions, set operations on the data, or query processing is being added in ongoing work. The initial development, which we describe here, was performed in a project by a group of four graduate students at the University of Magdeburg, after Berkeley DB's decomposition. From SPL lectures, the students were familiar with feature modeling and implementing SPLs using FeatureC++ [24].

Design. FAME-DBMS was designed after careful analysis of the domain and existing embedded DMBS (Courgar, TinyDB, PicoDBMS, Comet, and Berkeley DB). The initial feature model of FAME-DBMS, as presented in the kick-off meeting of the project, is depicted in Figure 5; only layout and feature names were adapted for consistency. It shows the subset to be implemented in this initial phase with 14 relevant features (grayed features were not implemented directly but have been introduced for structuring the feature model). To customize FAME-DBMS, we can choose between different operating systems, between a persistent and an in-memory database, and between different memory allocation mechanisms and paging strategies. Furthermore, index support using a B^+-tree is optional, so is debug logging, and finally it is possible to select from three optional operations get, put, and delete. Regarding solely the feature model, 320 different variants of FAME-DBMS are possible.

This feature model represents domain concepts and, in particular, the variability considered to be relevant. While a different feature model might have avoided some problems later on, the domain model should be independent of any

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1. Nut/OS	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
2. Win	o	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
3. InMemory			▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
4. Persistent			o	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
5. Static	x	x	o	o	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
6. Dynamic	x	x	o	o	o	▨	▨	▨	▨	▨	▨	▨	▨	▨
7. LRU			o	o			▨	▨	▨	▨	▨	▨	▨	▨
8. LFU			o	o			o	▨	▨	▨	▨	▨	▨	▨
9. Unindexed									▨	▨	▨	▨	▨	▨
10. B$^+$-tree									o	▨	▨	▨	▨	▨
11. Put	x	x		x					x	x	▨	▨	▨	▨
12. Get	x	x							x	x	x	▨	▨	▨
13. Delete	x	x		x					x	x	x	x	▨	▨
14. Debug	x	x	x	x	x	x	x	x	x	x	x	x	x	▨

Figure 6. Domain dependencies ('o') and implementation dependencies ('x') in FAME-DBMS.

implementation. The fine granularity is deliberately chosen, reflecting the domain's requirements (typically very specific tasks; binary size is expensive in mass production) [26].

Soon after the initial design, the students realized that many of the features could not be implemented orthogonally but that there would be many implementation dependencies. In Figure 6, we show the domain dependencies ('o') and the implementation dependencies ('x') that were expected additionally. The latter are instances of the optional feature problem. For example, feature DEBUG LOGGING has an implementation dependency with every other feature (it extends them with additional debugging code) but should be independent by the feature model. Also the features GET, PUT, DELETE, NUT/OS, and WIN have many implementation dependencies. This analysis already shows that it is necessary to find a suitable implementation to prevent dependencies, because ignoring them would almost entirely eliminate variability.

Implementation. We left the implementation up to the students. We recommended the derivative solution, but did not enforce it. In the remainder of this section, we describe the implementation at the end of the project and discuss choices and possible alternatives.

First, as expected, the students eliminated most implementation dependencies to increase variability. There were only two exceptions regarding the access API features: (1) the implementation of feature PUT and DELETE were combined, i.e., they were implemented with a single module so that they cannot be selected independently; (2) implementation dependencies to the implementation module of GET were not eliminated making this feature de facto mandatory in all variants. With this choice the students reduced the number of implementation dependencies from 36 to 22 but also the number of possible variants from 320 to 80. The intension behind these decisions was the following: although there are use cases for a database that can write but not delete data, or even for a write-only database (see [26]), these variants are used so rarely that the students considered the reduced variability acceptable.

Second, the feature DEBUG LOGGING was implemented using conditional compilation. This eliminated 11 implementation dependencies, but scattered the debugging code across all implementation modules. Alternatively, some debugging code could have been moved into the base implementation causing a little runtime penalty, 11 derivative modules could have been created, or even up to 2^{11} alternative implementations. The students decided to use conditional compilation to avoid additional effort.

Third, the implementation of B$^+$-TREE always contains code to add and delete entries, even in read-only variants. In those variants, the additional code is included but never called. This implementation was chosen because the solution was straightforward and simple in this case, compared to additional effort required for derivatives. A measurement after the end of the project revealed that the unnecessary code increased binary size by 4–9 kB (5–13 %; depending on the remaining feature selection).

Fourth, the remaining 10 implementation dependencies were refactored into derivative modules, following our original recommendation. The additional effort was considered the lesser evil compared to a further reduction of variability, a further scattering of code with preprocessor annotations, or a further increase in binary size. Alternative implementations were not considered at any time.

Overall, the FAME-DBMS case study illustrates that the optional feature problem is not necessarily a problem of legacy design, but appears to be inherent in SPLs with multiple optional features. The implementation of FAME-DBMS used a combination of various solutions, but still increased the code size in some variants, reduced variability and required effort to refactor 10 derivative modules. Even in such small SPLs the optional feature problem pervades the entire implementation.

4.3. Threats to Validity

Our case studies differ in three points from many SPLs we have seen in practice, which might threat external validity. First, both case studies are from the domain of embedded DBMS, raising the question, whether these results are transferable to other domains. Our experience from several small projects indicates that the optional feature problem also occurs frequently in other domains, but further empirical evidence is necessary.

Second, as stated above, we consider only implementation approaches in which variants can be *generated* from a common code base without implementation effort during application engineering. In approaches where reusable core assets are created and combined manually to build a variant, the task of handling the optional feature problem is deferred to application engineering. For example, it may be the responsibility of the application engineer to implement the "committed transactions per second" statistics in every variant with STATISTICS and TRANSACTIONS.

Third, our case studies use features at a fine level of granularity. We extracted very small features, some of which have many implementation dependencies. This is necessary in our target domain to achieve the desired flexibility and to deal with resource restrictions efficiently [13]. This is in line with Parnas' *design for change* that suggests minimal extensions [27]. The usage of rather coarse-grained features, often seen in practice, may indicate why the optional feature problem was not noticed earlier. Alternatively, feature granularity may have been modeled coarse intentionally to avoid this problem, despite the lowered variability. This means that designers have encountered the optional feature problem, and intuitively made trade-offs, using a combination of the solutions we described, and may never have realized the generality of the problem.

5. Discussion and Perspective

In both case studies, the optional feature problem occurred very often, i.e., many features that appear orthogonal in the feature model are not independent in their implementation. Ignoring implementation dependencies however reduces the SPL's variability to a level that completely defies the purpose of the SPL approach. Instead, solutions to eliminate one or more implementation dependencies are needed very often.

Which solution a programmer should use for eliminating implementation dependencies depends on the context of the SPL project. From an academic perspective, derivatives appear to be a good solution as they do not change the behavior and can be implemented in a modular way. Nevertheless, pragmatically also other solutions with lower effort may be preferred, considering the importance of binary size, performance, or code quality, and considering whether behavioral changes are acceptable. For example, for uncritical debugging code, scattering with conditional compilation might be acceptable; moving code is a suitable pragmatic solution if only small code fragments must be moved. There is no overall best solution, but our comparison in Table 1 and the experience from our case studies can be used as a guideline for selecting the appropriate (mix of) solutions for the SPL project at hand.

In our experience, determining the number of implementation dependencies is to some degree possible without actually implementing the SPL. Typically, a developer who is familiar with the target domain can predict whether two features can be implemented with orthogonal implementation modules. Systematically analyzing probable occurrences of the optional feature problem prior to implementation (as we did in Figure 6) can help choosing a suitable solution. For example, features that cause many implementation dependencies can be reconsidered in the feature model (e.g., in FAME-DBMS the features PUT and DELETE were merged after such analysis before implementation), or implemented with a solution that requires lower effort (see DEBUG LOGGING in FAME-DBMS).

Furthermore, the following decision has to be weighted for every SPL: Are implementation dependencies eliminated during initial development or just documented while their elimination is deferred until a customer actually requires a particular variant? Eliminating implementation dependencies upfront requires a higher initial effort but reduces marginal costs of products later (in line with a proactive adoption approach [28]). Postponing the elimination may encourage 'quick & dirty' solutions with lower effort but also reduced code quality.

As there is no generally satisfying solution, further research is needed to improve the current solutions. One promising area of research are automated refactorings that can extract derivatives with little human intervention [16], [7]. For this automation it is first necessary to locate the code responsible for the implementation dependency, which is a special instance of the feature location problem [29]. To what degree this process can be automated is an open research question. Regarding other solutions, tools that estimate the impact of moving code into another feature (in terms of binary size or performance) are helpful in deciding whether moving code is a suitable option, and automated transformations of conditional compilation into modules can help to increase modularity.

Another issue which we did not discuss so far (because it is difficult to quantify) is readability and maintainability. While conditional compilation is generally regarded as a problem for understanding and maintenance [21], [22], [2], modularity and a clean separation of concerns are supposed to help [19], [8]. However, we experienced that a feature split into many modules (e.g., one implementation module and 9 derivative modules for the feature STATISTICS in Berkeley DB) can be a double-edged sword: On the one hand, we can easily reason about the core implementation of statistics or the code for "committed transactions per second", but on the other hand, to understand statistics in its entirety, it is necessary to reconstruct the behavior from several modules in the developers head. We have reasons to believe that the other solutions might be easier to understand than a feature split into too many modules. What 'too many' means is an open research question. We also found that some tool support for conditional compilation can make this solution easy to understand when used in certain limitations. For example, navigation support and even views on the source code (e.g., show all code of feature STATISTICS) can aid understanding [7]. An empirical study (in the form of a controlled experiment) to compare the effect of different solutions on understandability and maintainability appears a promising path for future research.

6. Related Work

The optional feature problem is closely related to multi-dimensional separation of concerns and the tyranny of the dominant decomposition [8], which describes that programs are decomposed into modules according to a single dominant dimension only. Other concerns, which do not align with this dominant decomposition, are difficult to modularize. Decomposing a software system along different dimensions is a fundamental problem in software engineering. The optional feature problem adds another difficulty as modularization is not only desirable regarding code quality, but also for implementing optional features. If the application is not properly decomposed, implementation dependencies or variants with inefficient binary size or performance are the result. The solution used in Hyper/J is essentially equivalent to the derivative solution and splits a concern into many small modules [8], [17].

Furthermore, the optional feature problem is closely related to research in the field of feature interactions [30]. Feature interactions can cause unexpected behavior when two optional features are combined. In the common example of a cell phone with both call forwarding and call waiting, some additional code is necessary to clarify how to decide which features actually handles an incoming call. In some cases feature interactions can be detected by implementation dependencies, however detecting feature interactions that occur during runtime, as in the cell phone example above, is a very difficult task with a long history of research (see survey in [30]). Once the interaction has been detected, different implementations for different feature combinations can be provided with the solutions discussed in this paper.

A mismatch between feature model and implementation model can be automatically detected by tools, e.g., [18]. In this paper, we explored how to solve the optional feature problem once it has been detected. Furthermore, we assume that the implementation model correctly describes all dependencies between implementation modules. Detecting a mismatch between implementation model and implementation modules (which may include code prepared for conditional compilation) is addressed in research on *safe composition* respectively *type-checking SPLs* [31], [32], [33]. Finally, the matrices in Figures 3 and 6, which we use to document implementation dependencies (and to estimate their number upfront), are similar to a *design structure matrix (DSM)* [34]. DSMs are common means to analyze dependencies in software systems and aim at design or analysis of software systems. DSMs can be used to describe implementation models and for analysis, but again do not solve implementation problems.

There are only few studies that report experience with the optional feature problem. Some studies observe the difference between feature model and implementation model and that the latter is stricter than the former. However, in most studies this reduced variability is accepted without considering alternatives, e.g., in [35]. The derivative solution of using additional modules to separate two implementations was first introduced by Prehofer [15] and similarly used in [8] and [17]. Though not targeted at variability or at reusable artifacts in an SPL, adapters that connect independently developed classes or components use similar mechanisms. When combining two independent components, the 'glue code' between them is manually written or taken of a library of existing adapters [36], [37]. Finally, Liu et al. formalized the composition of features with a mathematical model and introduced the terms 'optional feature problem' and 'derivative module' [16]. All of these publications focus on underlying mechanisms, but do not report experiences on how derivatives scale to large SPLs.

Finally, we studied Berkeley DB already in prior work [23], [7], [14]. Already during our first decomposition, we noticed the optional feature problem, however previous studies focused on different issues first (performance of embedded DBMS [14], suitability of AspectJ to implement an SPL [23], and expressiveness of different implementation mechanisms [7]). In this paper, we focused only on the optional feature problem. It is independent of whether the SPL is implemented with AspectJ, AHEAD, Hyper/J, or a framework with plug-ins, and independent of the actual preprocessor used for conditional compilation.

7. Conclusions

We have discussed the optional feature problem, illustrated how it influences SPL development in two database SPLs, and analyzed the state-of-the-art solutions and their trade-offs. The optional feature problem comes from a mismatch of desired variability of the feature model and restrictions from implementation dependencies. In our case studies, variability was reduced to a minimum that rendered the SPL almost useless. While there are several solutions to eliminate implementation dependencies and thus restore variability, they all suffer from drawbacks regarding required effort, binary size and performance, and code quality. For example, derivatives promise a modular implementation but our case studies show that in practice the required effort can be overwhelming.

When faced with the optional feature problem, developers have to decide how to handle it. Our survey of different solutions, summarized in Table 1, and experience from our case studies can serve as guideline to decide for the best (mix of) solutions in the context of the SPL at hand.

In future work, we will investigate whether we can group features into dimensions and program cubes. Furthermore, we will explore automation and visualizations to reduce effort for the respective solutions.

Acknowledgments. We thank D. Benavides, M. Pukall, and S. Schulze for helpful comments on earlier drafts of this paper. The work on embedded DBMS is part of the project FAME-DBMS (#SA 465/32-1) funded by the German Research Foundation (DFG). Apel's work was supported in part by DFG project #AP-206/2-1. Batory's work was supported by NSF's Science of Design Project #CCF-0438786 and #CCF-0724979.

References

[1] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Addison-Wesley, 1998.

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

[3] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley, 2000.

[4] K. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Software Engineering Institute, Tech. Rep. CMU/SEI-90-TR-21, 1990.

[5] R. E. Johnson and B. Foote, "Designing reusable classes," *Journal of Object-Oriented Programming*, vol. 1, no. 2, pp. 22–35, 1988.

[6] S. Jarzabek, P. Bassett, H. Zhang, and W. Zhang, "XVCL: XML-based variant configuration language," in *Proc. Int'l Conf. Software Engineering (ICSE)*, 2003, pp. 810–811.

[7] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *Proc. Int'l Conf. Software Engineering (ICSE)*, 2008, pp. 311–320.

[8] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr., "N degrees of separation: Multi-dimensional separation of concerns," in *Proc. Int'l Conf. Software Engineering (ICSE)*, 1999, pp. 107–119.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ." in *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, 2001, pp. 327–353.

[10] D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling stepwise refinement," *IEEE Trans. Softw. Eng.*, vol. 30, no. 6, pp. 355–371, 2004.

[11] S. Apel, T. Leich, and G. Saake, "Aspectual feature modules," *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 162–180, 2008.

[12] P. Pucheral *et al.*, "PicoDBMS: Scaling down database techniques for the smartcard," *The VLDB Journal*, vol. 10, no. 2-3, pp. 120–132, 2001.

[13] M. Seltzer, "Beyond relational databases," *Commun. ACM*, vol. 51, no. 7, pp. 52–58, 2008.

[14] M. Rosenmüller *et al.*, "FAME-DBMS: Tailor-made data management solutions for embedded systems," in *Proc. EDBT Workshop on Software Engineering for Tailor-made Data Management*, 2008, pp. 1–6.

[15] C. Prehofer, "Feature-oriented programming: A fresh look at objects." in *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, vol. 1241, 1997, pp. 419–443.

[16] J. Liu, D. Batory, and C. Lengauer, "Feature oriented refactoring of legacy applications," in *Proc. Int'l Conf. Software Engineering (ICSE)*, 2006, pp. 112–121.

[17] D. Batory, R. E. Lopez-Herrejon, and J.-P. Martin, "Generating product-lines of product-families," in *Proc. Int'l Conf. Automated Software Engineering (ASE)*, 2002, pp. 81–92.

[18] A. Metzger *et al.*, "Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis," in *Proc. Int'l Requirements Engineering Conf.*, 2007, pp. 243–253.

[19] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[20] C. Kästner and S. Apel, "Integrating compositional and annotative approaches for product line engineering," in *Proc. GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering*, 2008, pp. 35–40.

[21] H. Spencer and G. Collyer, "#ifdef considered harmful or portability experience with C news," in *Proc. USENIX Conf.*, 1992, pp. 185–198.

[22] D. Muthig and T. Patzke, "Generic implementation of product line components," in *Proc. Net.ObjectDays*, 2003, pp. 313–329.

[23] C. Kästner, S. Apel, and D. Batory, "A case study implementing features using AspectJ," in *Proc. Int'l Software Product Line Conference (SPLC)*, 2007, pp. 223–232.

[24] S. Apel, T. Leich, M. Rosenmüller, and G. Saake, "FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming." in *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, R. Glück and M. R. Lowry, Eds., vol. 3676, 2005, pp. 125–140.

[25] D. Benavides, P. Trinidad, and A. Ruiz-Cortes, "Automated reasoning on feature models," in *Proc. Conf. Advanced Information Systems Engineering (CAiSE)*, 2005, pp. 491–503.

[26] R. Szewczyk *et al.*, "Habitat monitoring with sensor networks," *Commun. ACM*, vol. 47, no. 6, pp. 34–40, 2004.

[27] D. L. Parnas, "Designing software for ease of extension and contraction," *IEEE Trans. Softw. Eng.*, vol. SE-5, no. 2, pp. 128–138, 1979.

[28] P. Clements and C. Krueger, "Point/counterpoint: Being proactive pays off/eliminating the adoption barrier." *IEEE Software*, vol. 19, no. 4, pp. 28–31, 2002.

[29] D. Poshyvanyk *et al.*, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *IEEE Trans. Softw. Eng.*, vol. 33, no. 6, pp. 420–432, 2007.

[30] M. Calder *et al.*, "Feature interaction: A critical review and considered forecast," *Computer Networks*, vol. 41, no. 1, pp. 115–141, 2003.

[31] K. Czarnecki and K. Pietroszek, "Verifying feature-based model templates against well-formedness OCL constraints," in *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, 2006, pp. 211–220.

[32] S. Thaker, D. Batory, D. Kitchin, and W. Cook, "Safe composition of product lines," in *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, 2007, pp. 95–104.

[33] C. Kästner and S. Apel, "Type-checking software product lines – a formal approach," in *Proc. Int'l Conf. Automated Software Engineering (ASE)*, 2008, pp. 258–267.

[34] T. R. Browning, "Applying the design structure matrix to system decomposition and integration problems: A review and new directions," *IEEE Trans. on Engineering Management*, vol. 48, no. 3, pp. 292–306, 2001.

[35] C. Zhang, D. Gao, and H.-A. Jacobsen, "Towards just-in-time middleware architectures," in *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, 2005, pp. 63–74.

[36] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2002.

[37] M. Mezini and K. Ostermann, "Integrating independent components with on-demand remodularization," in *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2002, pp. 52–67.