Data & Knowledge Engineering 68 (2009) 1493–1512

Contents lists available at ScienceDirect

Data & Knowledge Engineering

journal homepage: www.elsevier.com/locate/datak

Tailor-made data management for embedded systems: A case study on Berkeley DB

Marko Rosenmüller [a,*], Sven Apel [b], Thomas Leich [c], Gunter Saake [a]

[a] School of Computer Science, University of Magdeburg, Germany
[b] Department of Informatics and Mathematics, University of Passau, Germany
[c] Metop Research Institute, Magdeburg, Germany

ARTICLE INFO

Article history:
Received 6 November 2008
Received in revised form 23 July 2009
Accepted 23 July 2009
Available online 26 July 2009

Keywords:
Tailor-made data management
Embedded systems
Software product lines
Feature-oriented programming
FeatureC++

ABSTRACT

Applications in the domain of embedded systems are diverse and store an increasing amount of data. In order to satisfy the varying requirements of these applications, data management functionality is needed that can be tailored to the applications' needs. Furthermore, the resource restrictions of embedded systems imply a need for data management that is customized to the hardware platform. In this paper, we present an approach for decomposing data management software for embedded systems using *feature-oriented programming*. The result of such a decomposition is a *software product line* that allows us to generate *tailor-made data management* systems. While existing approaches for tailoring software have significant drawbacks regarding customizability and performance, a feature-oriented approach overcomes these limitations, as we will demonstrate. In a non-trivial case study on Berkeley DB, we evaluate our approach and compare it to other approaches for tailoring DBMS.

1. Introduction

Today 98% of all computing systems are embedded systems [1]. Frequently cited examples are sensors, smartcards, and cellphones. Applications for these systems have different requirements on data management, ranging from simple data storage functionality, over stream processing, to complex data management including transactions, recovery, and replication. Separation of data management and application logic is needed to avoid redevelopment of data management in these systems. This can be achieved with a general data management infrastructure [2].

There are several challenges for data intensive applications in embedded systems, as we will illustrate in the following by the example of automotive systems. The amount of data that is processed in automobiles increases by 7–10% per year [3]. Data is captured in sensors and distributed and stored in working or persistent memory. Depending on the application scenario, different data management functions are needed including transaction management, query processing, and security mechanisms. The data stored in such systems ranges from single values and simple structures like arrays, as in sensors [4], to few tables or complete databases, as in navigation systems. Also advanced mechanisms like transaction processing or recovery are required in certain situations. The actually needed functionality depends on the application scenario and alternative implementations of functionality are required to support diverse hardware, e.g., special algorithms for storing data in EEPROM.

* Corresponding author. Tel.: +49 3916712994.
E-mail addresses: rosenmue@ovgu.de (M. Rosenmüller), apel@uni-passau.de (S. Apel), thomas.leich@metop.de (T. Leich), saake@ovgu.de (G. Saake).

0169-023X/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.datak.2009.07.013

1.1. Scale invariance of data management

As in automobiles, data management is required in most computing systems. In contrast to contemporary desktop and server systems, the resources of embedded devices are very limited. This includes memory, computing power, and power consumption. When developing data intensive applications such constraints have to be taken into account.

The memory limitations in embedded systems range from a few kilobytes (e.g., in sensors) to moderate restrictions of a few megabytes of memory (e.g., in cellphones). This situation is similar to the situation in desktop computing systems in the early '80s as well as for business applications in the '70s. For all of these systems data management was needed that had to operate under restricted resources. This trend seems to be continued with ubiquitous computing [5] and in the future with developments such as smart dust [6]. It may continue up to the smallest possible devices limited by the currently known physical laws [7]. We formulate this trend as the **law of scale invariance of data management**:

> There will be always small computing devices that operate with very constrained resources and independent of the size of these systems there is a need for dedicated data management.

1.2. Data management for embedded systems

Considering the limited resources and very special requirements on data management, traditional *database management systems (DBMS)* are inappropriate for use in embedded environments [8–11]. There have been approaches that try to scale down data management technology for embedded systems [9,10]. These approaches concentrate on supporting special hardware or application scenarios by manually creating customized solutions. When developing such customized systems, data management is often reinvented to satisfy computational and memory constraints as well as new kinds of requirements [11]. This practice leads to an increased time to market, high development costs, poor quality of software, and bugs. We argue that appropriate techniques are needed to develop tailor-made DBMS that attain high customizability and reuse without loss of performance.

Customizable software can be built with a number of different approaches such as components or preprocessors [12]. All these approaches have benefits and drawbacks. For example, components allow to modularize functionality but often degrade performance [11]. Furthermore, the crosscutting structure of some features hinders the use of components [13]. For example, encapsulation of a transaction management system as well as a B-tree into dedicated components is difficult to achieve because the transaction management system cuts across many other components of a DBMS including the B-tree. Preprocessors, e.g., the C preprocessor using `#ifdef` statements, do not have these problems but are known to pollute the source code and complicate maintenance and evolution of software [14,15]. As a result, neither components nor preprocessors are an optimal solution to build applications for embedded systems and new programming paradigms might be employed [13,16].

1.3. Contribution

We argue that *feature-oriented programming (FOP)* [17,18] is a promising technique to develop highly customizable DBMS for embedded systems in order to avoid the problems mentioned above. Using FOP, the functional requirements on a DBMS are represented by *features* that can be implemented in a modular way. FOP enables us to generate different applications by composing features which results in a *product line* of similar applications. We will show that FOP can be used to build a product line of data management software that (i) fulfills special requirements of a diverse set of applications, (ii) allows for generating different DBMS variants in a short period of time, and (iii) decreases resource consumption by including only required functionality. We also show that, in contrast to other approaches, the customizability has no negative impact on performance and resource consumption. We evaluate our approach by a refactorization and analysis of Oracle's embedded database system Berkeley DB.[1] Using FOP, we could decrease the binary size of a minimal Berkeley DB variant by about 50% and increase performance by about 16% for a reading benchmark. We could achieve these improvements while increasing customizability, i.e., we provide customizability also of small features in the refactored DBMS.

2. Tailor-made data management systems

The development of tailor-made data management software has been in the focus of research in recent years. Approaches for tailoring DBMS can be separated into solutions that focus on manual tailoring and solutions that achieve customizability of DBMS. Especially for embedded systems there are a number of manually tailored data management systems that address special application scenarios. For example, Bobineau et al. have developed PicoDBMS, a DBMS that supports special algorithms for smartcards [9]. Sen et al. have developed DELite, a data management system to be used on embedded devices [10]. They propose special storage mechanisms and query processing. TinyDB is a DBMS intended for sensor networks [19]. It is tailored to the needs of resource constrained nodes of a sensor network and includes TinySQL, a tailor-made lan-

[1] http://www.oracle.com/database/berkeley-db/db/.

guage for querying sensor networks. All these systems are tailored for a special use case and demonstrate the need for customizability in the embedded domain. However, such systems do not provide a general approach for customization that allows for reusing functionality for similar systems of a domain. The result is redevelopment of data management functionality for different DBMS. In order to avoid these problems, customization techniques can be used that provide an extensible architectures or allow to generate tailor-made DBMS, e.g., by composing components. In the following, we present an overview of such systems and analyze their benefits and drawbacks.

2.1. Component-based approaches

Component-based approaches are getting popular for development of DBMS. They allow to encapsulate functionality of software in separate modules [20,11,13,21]. Unfortunately, components have to deal with several difficulties. For example, components introduce an overhead to support dynamic composition and require an infrastructure to manage components [21]. This is even more daunting for very small components, which are necessary to provide high customizability [22]. Chaudhuri and Weikum propose components in the style of RISC processors that have limited functionality and are thus easier to develop and maintain; however, when decreasing the size of components the communication overhead further increases [11]. This limits customizability because a component cannot be arbitrary small. Additionally, separating functionality into components is difficult when it is scattered over large parts of the source code and other components. The result is a coarse-grained customizability as proposed by Geppert et al. with KIDS [20]. Hence, using components on embedded devices is limited to special use cases because they do not address the resource constraints of embedded systems and do not provide the required customizability.

2.2. Components and embedded systems

In order to overcome these limitations, component systems can be combined with approaches that provide fine-grained customizability like *aspect-oriented programming (AOP)* [23]. AOP allows for modularization of crosscutting functionality that is hard to encapsulate in a component. With COMET, Nyström et al. provide a component-based approach that uses AOP to tailor components by *weaving* customization code into a component if it is required [13]. In another approach, Tešanović et al. examined AOP for DBMS customization by modularizing crosscutting code into aspects [16]. They evaluated their approach using Berkeley DB but have shown only customizability for small parts and not the whole system. Unfortunately, none of these approaches (and there is no other approach that we are aware of) could show concrete implementations of a complete DBMS nor detailed evaluations. Furthermore, the customizability of these approaches is very limited (less than 10 features) and does not allow for building highly customized solutions that satisfy all requirements of embedded systems.

2.3. Preprocessors

As stated above, also preprocessors can be used to create customizable DBMS as it is done in Berkeley DB, a customizable DBMS written in the C programming language. The Berkeley DB developers use `#ifdef` statements and macros to achieve fine-grained customizability. In contrast to components, preprocessors do not introduce additional overhead because the composition of a concrete DBMS occurs before compilation and does not hinder compiler optimizations. Furthermore, also customizability of crosscutting features like a transaction management system can be achieved, e.g., by surrounding all code belonging to a feature with conditional preprocessor statements. In Section 4, we analyze Berkeley DB in more detail. Here we would like to point out that preprocessor statements are known to have a negative affect on maintenance of software [14] and because of missing modularization, also the evolution of software and even the elimination of dead features is problematic [15].

2.4. Kernel systems, frameworks, and others

Beside approaches that can be used on embedded systems, there are approaches to provide customizable or extensible DBMS intended for server systems. For example, Batory and Thomas used code generation to customize DBMS [24]. They aimed at creating special language extensions, e.g., to ease the use of cursors. As one of the origins of FOP, Batory et al. concentrated on customizing DBMS with Genesis [25]. In contrast to FOP, as it is known today, there was no support for object-oriented programming (OOP) and the complexity of the technique decreased usability. There have been many other developments to support extensibility of DBMS in the last 20 years. These approaches found their way into current DBMS (e.g., kernel systems) but cannot provide appropriate customizability to support embedded systems. Other disadvantages include detailed knowledge that is needed to implement a concrete DBMS or extend existing ones (e.g., kernel systems, frameworks) [21].

2.5. AOP and infrastructure software

AOP has shown to be appropriate to provide customizability, e.g., in operating systems [26–28] and middleware [29–31]. These studies show that AOP can be used to decompose infrastructure software with respect to crosscutting features. Evaluations of these solutions show that AOP can be used with negligible impact on performance and resource consumption.

In this paper, we present an approach that focuses on software product lines and is based on FOP, which is similar to AOP but provides support for modularizing the features of software. We will show that FOP can also be used to separate crosscutting concerns without negative impact on performance. AOP and FOP are conceptually different and some studies propose that collaboration based designs like FOP should be preferred when developing customizable software [32–35].

3. Methodology

In this section, we introduce *feature-oriented programming (FOP)* [18,17], a programming paradigm for the development of customizable software. With *FeatureC++* [36] we developed an FOP language extension for the C++ programming language. This allows us to apply FOP to software systems intended for resource constrained environments. Software development based on features was applied successfully in different domains [35,37–40,31,27,41–43,30], but there was less research regarding the application to data management or embedded systems [44]. We will demonstrate that FOP can be used to develop customizable data management software that satisfies the requirements of embedded systems.

3.1. Feature-oriented programming

A *software product line (SPL)* is a family of similar programs that can be distinguished in terms of *features*. A *feature* is a property of an SPL which is relevant to some stakeholder and used to discriminate between programs of that SPL [12]. Examples for features of a DBMS are transaction management or query processing. But there may also be smaller features like a lock protocol used to implement a transaction management system. The idea behind FOP is to modularize the features of an SPL and generate programs by selecting only required features.

3.1.1. Feature modules

A *feature module* implements a feature as an increment in program functionality [18]. In FOP, feature modules are kept separate from each other to comply with the principle of *separation of concerns* [45]. FOP supports implementing different variants of a feature to provide solutions specialized for different use cases. To generate a tailor-made program a user selects a subset of the available features and variants of features. In contrast to object-oriented software development, feature modules can be freely composed to derive different programs which is the reason for scalability of the approach.

A simple example of a feature-oriented DBMS design is given in the left part of Fig. 1. It shows a base program and two features Transaction and Replication. Basic database functionality is implemented in module Base and is extended by modules Transaction and Replication that implement transaction processing and replication functionality of a DBMS. Based on this decomposition, i.e., separation of features, different variants of concrete DBMS (right part of Fig. 1) can be generated automatically by composing the required feature modules. In our example, possible DBMS support transactions as well as replication (DB_{TxnRep}), only basic functionality, or one of the features (e.g., DB_{Txn}).

3.1.2. Decomposition of classes

FOP can be used as an extension of *object-oriented programming (OOP)*. When considering features of object-oriented software, we observe that only a fraction of a class belongs to a feature and the rest to other features. Consequently, using FOP on top of OOP, the classes have to be decomposed with respect to the features of the software. In Fig. 2, we depict a decomposition of three DBMS classes `DB`, `Cursor`, and `B-tree` with regard to the features of Fig. 1. That is, for every class, basic functionality is separated from feature-specific functionality. Classes in module Base are *refined* in features Transaction and Replication, which we denote by an arrow. In this example, the basic implementation of class `Cursor` is refined to implement

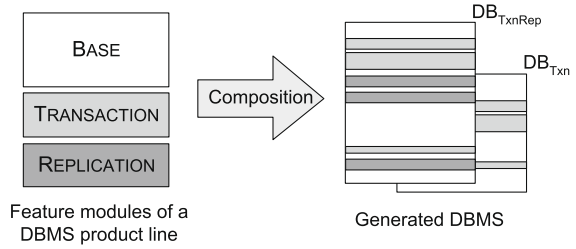


Fig. 1. Generating database applications from feature modules.

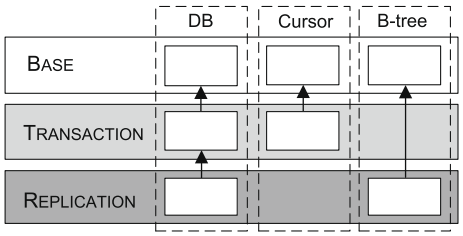


Fig. 2. Decomposition of classes (vertical bars) with respect to features (horizontal bars).

```
1  //Base implementation
2  class DB {
3    bool put(Key& key, Value& val) { ... }
4  };
```

```
5  //Extensions needed for feature Transaction
6  refines class DB {
7    TXN* begin_transaction() { ... }
8
9    bool put(Key& key, Value& val) {
10     //Transaction specific code
11     ...
12
13     //Invocation of the refined method
14     return super::put(key,val);
15   };
16 };
```

```
17 //Extensions needed for feature Replication
18 refines class DB {
19   int rep_start() { ... }
20 };
```

Fig. 3. Source code of class `DB` written in FeatureC++.

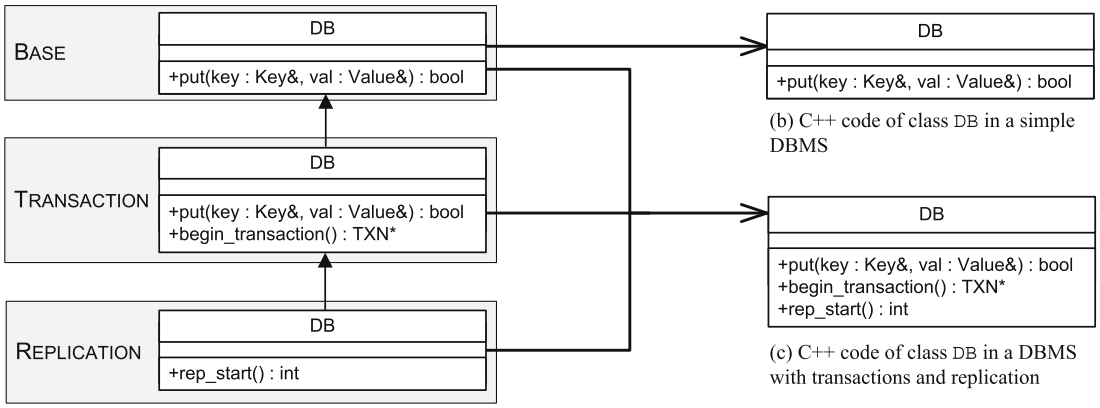
feature TRANSACTION, which is in contrast to class `DB` that is refined in both features. Features TRANSACTION and REPLICATION cut across the entire source code of a DBMS and are called *crosscutting features* [23].

3.1.3. FeatureC++

For refactoring Berkeley DB we use FeatureC++,[2] a feature-oriented programming language [36,46,47]. As FeatureC++ is built on top of C++ it has to support a decomposition of classes as introduced above. An excerpt of the FeatureC++ source code of class `DB` is shown in Fig. 3. The basic implementation (Lines 1–4) includes functionality needed in every DBMS variant. It is extended by refinements in features TRANSACTION and REPLICATION (Lines 5–20), declared by keyword `refines`. In FeatureC++, refinements can introduce new members (Lines 7 and 19) and extend existing methods (Lines 9–15). In method extensions the refined method is invoked using the keyword `super` (Line 14). Based on this decomposition of classes and a user's selection of features, classes are composed to include only functionality necessary for a specific DBMS. Code that is not needed is removed from a generated program to reduce its binary size, the amount of needed working memory, and to increase performance. For example, the code in Lines 5–16 of Fig. 3 is not present in a DBMS if feature TRANSACTION is not selected. As a result of the feature selection process, the code size of class `DB` is reduced and also the size of instantiated objects can be decreased.

FeatureC++ source code is processed by the FeatureC++ precompiler. It uses a source-to-source transformation from FeatureC++ code into C++ code which is then compiled by an ordinary C++ compiler. An example for the code transformation

[2] http://www.iti.cs.uni-magdeburg.de/iti_db/fcc/.



(a) FeatureC++ code of class DB

(b) C++ code of class DB in a simple DBMS

(c) C++ code of class DB in a DBMS with transactions and replication

Fig. 4. FeatureC++ code transformation: UML representation of class DB in FeatureC++ (a) and two different variants of the composed class in C++ (b and c).

of class DB (cf. Fig. 3) is shown in Fig. 4. Depending on the feature selection, different variants of class DB can be generated. For example, a simple variant (Fig. 4b) is derived by using only the implementation defined in module BASE and a full variant, including features TRANSACTION and REPLICATION, is derived by composing all refinements into a single class (Fig. 4c).

The FeatureC++ precompiler performs source code optimizations to avoid any overhead at runtime that might be introduced by decomposing a class into multiple refinements. For example, the generated C++ code is optimized for inlining of method refinements to achieve performance that is equal to a C++ implementation. For the composed class shown in Fig. 4c this means that code of refinements of method `put`, defined in BASE and TRANSACTION (Fig. 4a), is inlined into a single method by the C++ compiler. This avoids an overhead for additional method calls as it is usually found when using object-oriented concepts for extensions like virtual methods.

3.1.4. Expected benefits of FOP

FOP has a number of benefits that are of interest for software development, which have been observed in several studies [39,44,40–42]. Especially of interest for embedded systems are:

- Small binary size (footprint) of an application.
- Reduced amount of needed working memory.
- Enhanced performance.
- Improved comprehensibility of source code.
- Improved customizability and reusability.

Application footprint is important especially for the embedded domain and performance is highly important for DBMS. Therefore, we analyze these properties in detail in our evaluation.

3.2. Refactoring

Most contemporary data management systems are written in the C programming language. In some cases also object-oriented languages like Java or C++ have been used. Implementations of DBMS are usually highly tuned and cannot be reimplemented from scratch using a novel programming paradigm like FOP without a huge amount of work and the risk of degrading performance or introducing errors. We will demonstrate that a refactoring approach can be used to decompose legacy DBMS. That means, we show how an existing DBMS can be restructured with respect to features to yield a DBMS product line.

For refactoring, we propose minimal invasive code transformations that do not change the design in order to avoid errors and to preserve the behavior of the software [48]. By partially automating this process, we are able to transform even large applications into a product line. In our case study, we present the refactoring of Berkeley DB using FeatureC++.

4. Berkeley DB: a case study

Berkeley DB is an embedded DBMS for use in server systems but also in embedded systems. In our case study, we refactored the C version of Berkeley DB[3] into an SPL. In the following, we give a short overview of Berkeley DB and describe the

[3] We used the C version 4.4.20 of Berkeley DB.

```
1  __rep_queue_filedone(...) {
2  #ifndef HAVE_QUEUE
3    COMPQUIET(rep, NULL);
4    COMPQUIET(rfp, NULL);
5    return (...);
6  #else
7    db_pgno_t first, last;
8    u_int32_t flags;
9    int empty, ret, t_ret;
10 #ifdef DIAGNOSTIC
11   DB_MSGBUF mb;
12 #endif
13   ... //92 Lines of Code
14 #endif
15 }
```

Fig. 5. Static configuration in Berkeley DB using nested preprocessor statements.

refactoring process. Since we used the C version of Berkeley DB, we propose a two step refactoring process: (i) the conversion from C to C++ and (ii) the conversion from C++ to FeatureC++.

4.1. An overview of Berkeley DB

The source code of Berkeley DB contains 96 thousand lines of C code, excluding comments. To use Berkeley DB in resource constrained environments, developers use static configuration to decrease the size of the binary code by removing unneeded source code. The configuration mechanism is implemented using C preprocessor statements and macros.

The low binary size of Berkeley DB and the high performance achieved by its integration into client applications enable Berkeley DB to be used in a wide range of computing systems. Examples are Amazon's inventory of products, set top boxes from Sony, Samsung's digital videorecorder, Motorola's smartphone A768.[4] Nevertheless, with a minimal binary size of 484 KB, Berkeley DB is still too large to be used in deeply embedded devices, in which only a few KByte of memory are available. We demonstrate that FOP is an appropriate approach to scale down data management systems like Berkeley DB for the use in deeply embedded systems.

4.1.1. Comprehensibility of source code

Berkeley DB developers maximized the performance by avoiding function calls with macros, *goto*-statements, and very long methods (up to several hundred LOC). Preprocessor statements are used for the configuration of Berkeley DB. In Fig. 5, we show a code sample from Berkeley DB where different features are surrounded by preprocessor statements. Very long code sections between `#ifdefs` ($>$ 100 LOC) do often not allow to easily decide if a part of the source code belongs to a particular concern or another. The preprocessor statements are nested and enclose code that belongs to different features. It has been observed that nesting of preprocessor statements degrades readability, comprehensibility, and maintainability of the source code [14,15]. The intermingled functionality and the size of the functions make local behavior hard to understand.

4.1.2. Customizability and reusability

In Berkeley DB, static configuration is used to create different variants. However, the customizability that is achieved this way is limited: some features cannot be removed and others can only be removed partially, in particular, where code is not entangled with the remaining source code. This is caused by features that crosscut each other. A decomposition of crosscutting features is also possible with preprocessor statements but would lead to more nested statements. Furthermore, this hinders reuse of features in different other places than intended at development time, e.g., for developing a different DBMS.

4.2. Refactoring Berkeley DB

As FeatureC++ builds on C++, we discuss some issues related to performance before we present our results of refactoring Berkeley DB.

[4] http://www.oracle.com/database/berkeley-db/db/index.html.

4.2.1. C++ and performance

The C programming language is still widely in use. In some cases the use of C++ is refused because poor performance and high resource consumption is assumed. According to Stroustrup there is no evidence for this argument [49]:

> Contrary to popular myths, there is no more tolerance of time and space overheads in C++ than there is in C. The emphasis on runtime performance varies more between different communities using the languages than between the languages themselves. In other words, overheads are found in some uses of the languages rather than in the language features.

It is important to use the language constructs of C++ in an appropriate way. Especially in embedded systems one should avoid expensive language constructs like exception handling or virtual methods. In the refactoring of Berkeley DB, we considered this and avoided those language constructs.

4.2.2. Transformation from C to C++

Software written in C is usually based on a decomposition of the source code into files of similar functionality and `structs` that encapsulate data. To simplify the process of refactoring C to C++, we transformed existing files and `structs` of the Berkeley DB code base into C++ classes. The advantage of this approach is the direct mapping from C code to C++ code and thus a conversion process that can be automated. There are some tools that support a C to C++ conversion[5]; nevertheless, we developed our own tool, because we are generating mainly static methods. This allows us to avoid arbitrary classes to be instantiated which would increase resource consumption. As a result, our approach generates C++ source code that is equivalent to the original C source code, but uses classes with static methods. The following overview describes the used code transformations:

1. *File to class conversion:* For each C source file a C++ class is generated and C functions are by default converted into static methods of the generated classes.
2. *Conversion of structs:* Structs usually encapsulate data fields that are members of a corresponding class in an object-oriented design. The classes generated in (1) usually contain methods that operate on these structs. This is caused by an object-based programming style in C which is used in Berkeley DB. In this case, fields of structs have to be moved to the according generated class that operates on this data. This conversion means to merge structs and the newly generated classes.
3. *Function conversion:* Functions that operate on fields previously defined in structs are transformed into regular methods and not into static methods as described in (1). Pointers to structs are usually passed as arguments to these functions and can be removed because they are implicitly provided as `this`-pointers.
4. *Removing function pointers:* Function pointers are often stored in structs to enable object-based programming. The pointers refer to C functions that usually operate on fields defined in these structs thus emulating objects. The pointers can usually be removed since non-static methods have been generated from the according C functions as described in (3). Code for initializing function pointers can be removed as well since functions are replaced by ordinary OOP methods.

The presented conversions can be applied in a step-wise manner. Using only a conversion into classes with static methods (1) is the simplest case and can be fully automated. Conversions (2)–(4) replace the object-based programming style often found in C source code and introduce regular classes. This means avoiding function pointers and manual pointer initialization.

4.2.2.1. Object-oriented redesign. Based on this refactoring, an object-oriented redesign can follow to improve the maintainability and resource consumption but with the risk of introducing errors. In case of Berkeley DB, this seemed to be necessary, because code replication in some methods was obvious and a simple inheritance based redesign could remove the redundancies. We limited our redesign to only seven classes to form an inheritance hierarchy in order to avoid fundamental changes. For example, classes used for implementing different index structures now reuse the same code implemented in a common base class. However, we did not use any virtual methods to avoid a performance impact and to preserve real-time capabilities.

4.2.3. Transformation to FeatureC++

In a second step, we transformed the C++ code into FeatureC++ code. We developed a tool that transforms the code semi-automatically and extracts user-defined features of Berkeley DB into separate modules.

4.2.3.1. Extracting features. Besides a decomposition of software into files or classes, programmers often apply a further structure to the source code by arranging files in a directory hierarchy. This decomposition can also be found in Berkeley DB and is used to separate feature code from the base implementation. For example, a folder `hash` is used in Berkeley DB to group files for the HASH index feature. In contrast to FOP, however, it is not a complete decomposition since features that crosscut other parts of the source code (e.g., other structs or classes) cannot be modularized this way. Nevertheless, we

[5] For example, http://www.scriptol.org/ctocpp.php.

utilize this inherent structure and our tool creates a feature for each existing directory as an initial feature-oriented decomposition.

This basic decomposition of the source code into features is extended with a user-defined mapping of classes to features, i.e., the user defines which classes should be part of which features of the generated FOP code. We thus selected features for modularization that provide functionality which is not necessarily needed for every application and can significantly decrease the size of Berkeley DB, e.g., for transaction management and recovery. We furthermore selected features that are mandatory for a DBMS but increase the modularity of the source code. This basic transformation does not include a decomposition of classes into refinements as it is needed for clean separation of concerns. Nevertheless, it is a proper foundation for a simple transformation process. Based on such a mapping our tool (i) parses C++ files, (ii) moves the files to folders that store the feature a class was assigned to, (iii) removes includes not needed for FeatureC++, and (iv) merges implementation (`.cc`, `.cpp`) and header files (`.h`, `.hh`) of a class.

This basic transformation from C++ to FeatureC++ does not include a decomposition of classes into refinements, but it results in FeatureC++ code which is equivalent to the underlying C++ code. In combination with our transformation from C to C++ the source code of the refactoring is equal to the original C code.

4.2.3.2. Decomposing classes. The refactoring described so far is sufficient for features that are restricted to some files and do not affect classes of other features. This, however, is usually not the case and a manual refactoring has to follow to completely modularize features. Such crosscutting features often affect large parts of the source code and beside the base program also other features. An examples is the transaction management that crosscuts many classes and features in a DBMS (more than 30 classes and 11 features in Berkeley DB). For example, B-Tree classes are extended to introduce code for transaction management. Due to the absence of appropriate tool support, we extracted the features manually. That is, we decomposed the classes into base implementation and refinements according to crosscutting features. This means that code encapsulated by `#ifdef` statements is refactored into a class refinement of the according feature, e.g., moving a method into a refinement.

In some situations we needed to create *hook methods* to decompose methods according to features. Hook methods, often used in frameworks, are usually abstract methods that introduce extension points into other methods that call these hook methods [50]. Their use in FOP is very similar and they enable subsequent features to introduce specific code into the middle of a method via refining hook methods [42]. For example, in order to provide an extension point for cursor initialization, we use a hook method `InitNewCursor` that can be overridden by subsequent features to provide initialization code. This hook method is called while creating new cursors and its base implementation is empty and is overridden in features like B-TREE. The use of hook methods can be complicated if local variables are involved. These variables have to be passed as arguments to hook methods to allow their use in method extensions [51].

In the C version of Berkeley DB, function pointers are not only used to provide an object-based programming style but also for configuration purposes. In this case, function pointers are initialized depending on a feature selection in combination with static configuration based on preprocessors. For example, when distributed transactions are used, function pointers for transactions refer to the functions for distributed transactions instead regular transactional functions. This approach is semantically equal to method refinements of FOP. For example, code for distributed transactions can be implemented in a method refinement that overrides the original implementation and is only available if distributed transactions are used. Replacing such function pointers with refinements of FOP means an automation of the manual configuration process, i.e., replacing manual pointer initialization by FOP code transformation.

4.2.3.3. Summary. Overall, we extracted 35 (cf. Fig. 6) features with 24 of them being optional (11 optional features in the original version). The remaining 11 features are mandatory for each variant of Berkeley DB and thus cannot be removed

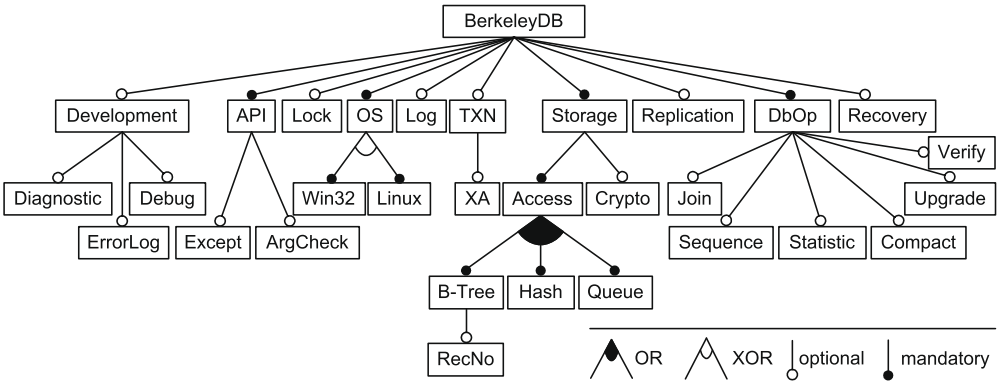


Fig. 6. Excerpt of the feature diagram of Berkeley DB after refactoring. Only optional and alternative features are shown.

Table 1
Optional and alternative features of Berkeley DB and their size in LOC after refactoring.

Feature	Description	LOC
Base	Core database functionality	33,346
B-Tree	B-tree index structure	12,363
Hash	Hash index structure	8449
Replication	Data replication in a distributed scenario	6087
Queue	FIFO optimized index structure	5009
Transaction	Transaction management subsystem	4208
Log	Logging; used for transactions and recovery	3248
Recovery	Recovery after failure	2474
Crypto	Rijndael/AES data encryption	2222
Verify	Database verification	1982
Others	Remaining features (e.g., distributed transactions)	3487

Table 2
Analyzed variants of benchmark applications with Berkeley DB embedded into a benchmark application. Shown are the used features and the binary size for C and FeatureC++ variants of the application.

Configuration	Features[a]	Binary Size [KB]	
		C	FeatureC++
(1)	B-Tree, TXN, Others, Queue, Rep, Hash, Crypto	664	636
(2)	B-Tree, TXN, Others, Queue, Rep, Hash	644	620
(3)	B-Tree, TXN, Others, Queue, Rep	580	552
(4)	B-Tree, TXN, Others, Queue	528	484
(5)	B-Tree, TXN, Others	492	452
(6)	B-Tree, TXN	416	376
(7)	B-Tree	N/A	224
(8)	Queue	N/A	184

[a] TXN: transactions (includes logging and recovery), Others: other small features, Rep: replication.

in a concrete configuration. However, it is still useful to separate these features to increase comprehensibility and to allow for alternative implementations of such features. In Table 1, we give an overview of the resulting features and their size in LOC.[6] Feature TRANSACTION is one of the features that was not optional in the original version of Berkeley DB. Using FOP we were able to modularize it and thus can generate variants of Berkeley DB without transaction management.

5. Evaluation

For evaluation we analyze the feature-refactored version of Berkeley DB with respect to customizability and resource consumption. We compare several variants with the original C version of Berkeley DB.[7]

5.1. Customizability

In its original version 11 features are optional and can be individually disabled when generating a concrete instance of Berkeley DB. This results in about one thousand different variants. Fig. 6 shows an extract of the *feature diagram* [12] of Berkeley DB after refactoring; mainly optional and alternative features. Features denoted with an empty dot are optional and increase the number of possible variants. Features denoted with empty and filled arcs are exclusive and inclusive alternatives. In the refactored version, we extracted 24 optional features or variants of features. Based on all existing constraints between features, there are more than 400,000 valid combinations. That is, we have significantly increased the variability.

For our analysis we use eight different variants of Berkeley DB embedded into a small benchmark application, as shown in Table 2. The size of the generated application is decreasing from configuration 1–8. Configurations 1–6 use the same features in the C and FeatureC++ variants. Configuration 6 is the smallest possible C variant using the index structure B-tree. Configurations 7–8 are minimal variants using B-tree (7) and Queue (8) as index structures. Both variants are not available in the C versions of Berkeley DB because features like TRANSACTION have been removed in these variants which is not possible in the original version.

[6] The displayed values are the number of lines in the refactored FeatureC++ version. These are slightly smaller than in the original version of Berkeley DB.
[7] The used source code of Berkeley DB is available under http://wwwiti.cs.uni-magdeburg.de/iti_db/BerkeleyDB/.

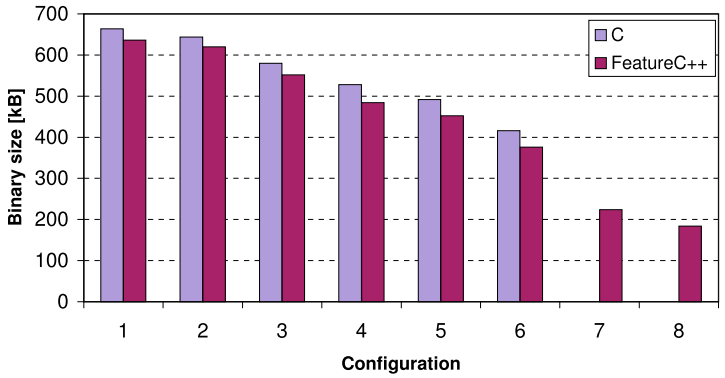


Fig. 7. Binary size of different C and FeatureC++ variants of Berkeley DB. See Table 2 for a description of shown configurations.

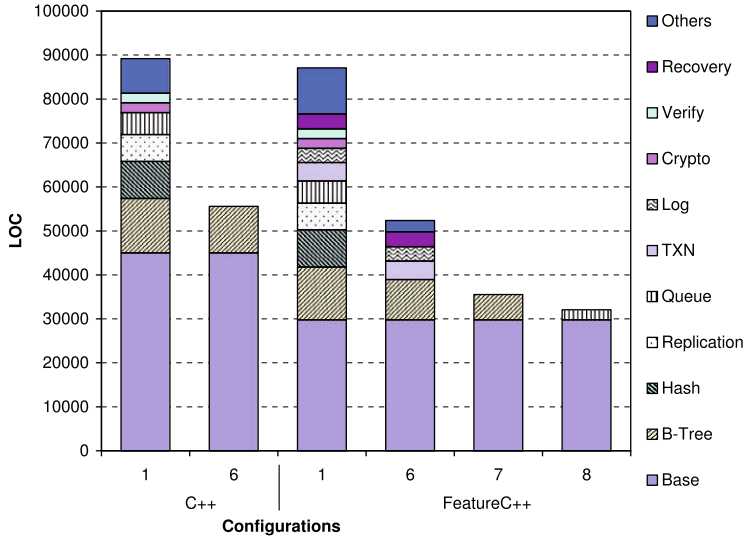


Fig. 8. Source code comparison of features in C++ and FeatureC++ variants of Berkeley DB. Complete (configuration 1) and minimal variants (configurations 6–8) are depicted. See Table 2 for a description of shown configurations.

5.2. Footprint

The binary size (footprint) of an application depends on the size of executable code and static data. Comparing our feature-refactored version of Berkeley DB with the original version, we observe roughly equal binary sizes for equivalent configurations (i.e., the same set of configured features). Table 2 and Fig. 7 summarize the measured binary sizes of different configurations of the benchmark application. Reasons for a reduction of the footprint in FeatureC++ variants (e.g., from about 664 KB to 636 KB in Configuration 1) are mainly differences in the programming paradigm. For example, in the C version of Berkeley DB function pointers are used to mimic an object-based programming; these have to be manually initialized when instantiating objects and increase the binary size. These differences are negligible compared to the overall size of the application.

The binary size of applications that use Berkeley DB can be further decreased by removing features that are not needed. When comparing the smallest variants of the feature-oriented refactoring of Berkeley DB (7 and 8) to the smallest C variant (6), we see significant differences (code reduction of about 50%).

5.3. Size of source code

In order to get more insight into the reasons for a reduced binary size, we analyzed the source code of Berkeley DB. In Fig. 8, the number of lines of Berkeley DB code before and after feature-oriented refactoring are depicted.[8] While in complete configurations (1) the features are nearly of equal size, they differ when comparing minimal configurations (6).

[8] We had to omit a comparison to the original C-version since it is significantly larger (6 KLOC). This is due to the object-based programming style in C and the differences between C and C++.

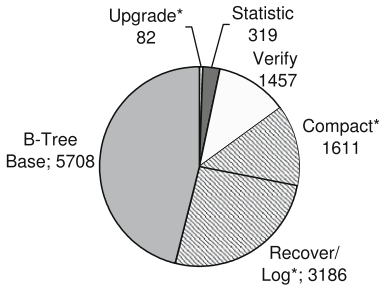


Fig. 9. Lines of code of feature B-tree consisting of the basic implementation (*Base*) and portions of other crosscutting features.

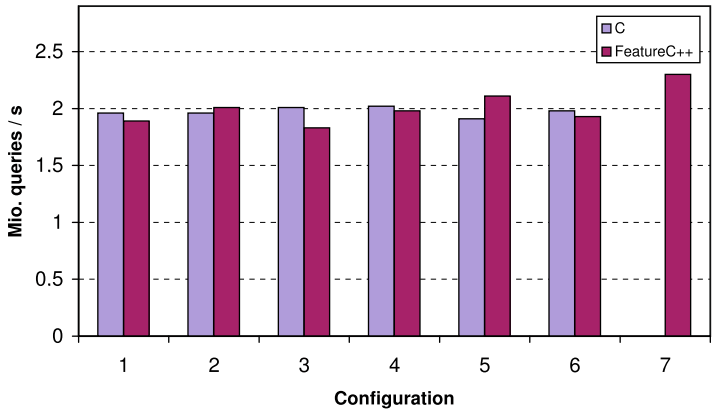


Fig. 10. Performance comparison (Oracle benchmark) of C and FeatureC++ variants for different feature selections. Higher values mean better performance. See Table 2 for a description of shown configurations.

The difference between the minimal configurations (6) is not only caused by differences in the programming paradigm but also by additional extraction of crosscutting features (e.g., Recovery). Thus, the size of features also decreases when crosscutting features are removed. This trend is continued in configuration (7) which is only available in the feature-refactored version. For example, feature B-tree as shown in configuration (1) is about 7 KLOC larger when compared to configuration (7) which is caused by removing all features that crosscut the B-tree. Fig. 9 shows the crosscutting features in detail. For example, feature Recovery makes up a large part of the B-tree implementation which is needed for recovery of the index. By omitting the feature Recovery the size of the B-tree source code decreases by 3186 LOC.

5.4. Performance

As discussed earlier, C++ has to be used carefully to avoid performance penalties. We considered this when imposing an object-oriented design on Berkeley DB. For example, we did not use virtual methods. In Fig. 10, we depict performance comparisons between the C and FeatureC++ variants of Berkeley DB.[9] We use the same configurations as described above. The performance for configurations 1–6 is roughly equivalent when comparing C with FeatureC++ variants. Configuration 7 is the smallest FeatureC++ configuration using the B-tree index structure.[10] Among others, feature Transaction was removed in this FeatureC++ variant (cf. Table 2). It results in a performance improvement of about 16% compared to the minimal C version. This is caused by the removal of dynamic checks that are evaluated even when features like Transaction are not used. Especially for transactions, numerous of such tests are utilized, which explains the better results for configuration 7 in Fig. 10. This is not the case for Configurations 1–6.

6. Discussion

Our evaluation has shown that we are able to preserve the performance characteristics of Berkeley DB when transforming it from C into FeatureC++ and even when applying a more fine-grained decomposition. This also shows that C++ and

[9] We used an Intel Core 2 system with 2.4 GHz and operating system Windows XP. The used benchmark is a reading benchmark for Berkeley DB available from Oracle: http://www.oracle.com/technology/products/berkeley-db/pdf/berkeley-db-perf.pdf.

[10] Configuration 8 (as shown in Fig. 7) was omitted since a comparison is not meaningful due to the use of the *Queue* index structure instead of a *B-tree*.

FeatureC++ do not necessarily have a negative impact on performance. In contrast to dynamic configuration, e.g., using `if` statements and function pointers, a performance improvement is possible by using static composition of features. However, also qualities of the source code like *separation of concerns* and comprehensibility are important with respect to maintainability and extensibility of software. In the following, we discuss our observations and summarize the benefits of the presented decomposition.

6.1. Separation of concerns

All features that we modularized in Berkeley DB cut across large parts of the base program. Crosscutting features also affect other features, e.g., the transaction management interacts with 11 other features. When extracting such features in Berkeley DB, we made some observations regarding structure and modularity of the source code.

6.1.1. Static customization

The C version of Berkeley DB already supports static composition of some features using C preprocessor directives. The programmers of Berkeley DB use function pointers to support object-based programming and to exchange the implementation of a method depending on the configuration, i.e., depending on the feature selection. In order to provide this customizability, function pointers are initialized when running Berkeley DB. The according initialization code has to be written manually by the programmer.

When using FeatureC++, object-oriented concepts, i.e., classes with methods, eliminate the need for function pointers. Furthermore, static composition of method refinements replaces function pointer based configuration code as well as static composition based on the C preprocessor. This automatic composition of methods reduces implementation effort and is less error-prone than a manual set up of function pointers. It introduces safety for composition of variants: a method is always correctly initialized, avoiding runtime errors like accessing a null function pointer, and consistent configuration of all classes is achieved automatically. Furthermore, a type system can assure static type safety of all possible variants of an SPL [52]. Technical aspects like memory consumption and performance also benefit from using FeatureC++. The reason is that function pointers do not have to be stored in objects and the compiler can inline method refinements, which is not possible for functions referenced by pointers.

6.1.2. Decomposition of large methods

Many methods in Berkeley DB are quite large and contain entangled functionality of different features. Reasons for this lack of separation of concerns at function level are performance optimizations which, however, can also be achieved by inlining capabilities of modern compilers. For those methods, we created up to seven methods each by using the extract method refactoring [48]. In some cases, we used *hook methods* to decompose large methods. This results in additional effort for decomposing such methods but tool support could significantly ease the refactoring of large applications [53,54].

Decomposing methods into smaller methods and method extensions, including the use of hook methods, does not have a negative effect on performance. The reason is method inlining which can be employed because all methods are part of the same compilation unit in the generated C++ code.

6.1.3. Variable method signatures

Crosscutting features may not only extend a method body but also affect parameters of a method, e.g., add a parameter that is required for executing the method. We also observed this in Berkeley DB when extracting the transaction management. Some method extensions require an additional transaction parameter and thus extend the signature of refined methods. For example, there is a method `put` in Berkeley DB similar to the method shown in Fig. 3. In order to support transactions, an additional parameter of type `TXN` is required which is introduced in feature Transaction. In the base implementation without any transaction code, the parameter is not needed and should not be part of the method signature. This leads to variability of the signature for different configurations, i.e., with or without transactions. We analyzed different approaches to handle such method extensions and found that the C preprocessor (i.e., using `#ifdefs`) is also inappropriate because it results in a variable method signature which hinders interaction with other software [55,56]. In our decomposition of Berkeley DB, we avoided variable signatures which is possible, e.g., by providing empty classes that are used as arguments if these classes are defined in optional features [57].

Overall, the effort to modularize a feature depends strongly on its crosscutting nature. Thus when deciding to extract a feature, the effort and the resulting benefit has to be taken into account. In our case study, we did not remove all preprocessor statements because of their large number and the effort involved. We think that tool support on top of FOP is necessary to reduce the refactoring effort and handle fine-grained decompositions of the source code [58].

6.2. Comprehensibility of source code

Comprehensibility of source code depends on a number of properties, such as separation of concerns, size and complexity of methods, etc. The C preprocessor does not support modularization of feature code as it is possible with FOP. The result is code of features that is entangled with other code and scattered all over the program. As a result, local behavior can only be understood by inspecting large parts of the code [14].

Table 3
Comparison of different approaches for developing customizable data management solutions for embedded systems. Properties better supported or fulfilled by an approach are rated with a filled dot. Insufficient support or a negative impact of an approach to a property is shown with an empty dot. Neutral ratings are denoted by partially filled dots.

	Kernel systems	Components	Components and AOP	CPP	FOP
Performance	○/◐[a]	○	◐	●	●
Footprint	○/◐[a]	○	◐	●	●
Granularity	○	○	◐	●	◐
Separation of concerns	◐	◐	●	○	●
Comprehensibility	◐	◐	◐	○	◐

[a] The impact on performance and footprint depends on the composition technique and supported extensibility.

Using FOP, features are modularized and separated from each other. Parts of classes and methods can be related easily to their functionality (i.e., their feature) and can usually be understood without considering code of other features. One of the major features that crosscuts large parts of the source code is the transaction management system (4208 LOC). In Berkeley DB, we extracted the transaction management system, as other features, and separated it from the remaining functionality. This decreases the code size of other features. For example, feature B-tree consists of core functionality and other crosscutting features (cf. Fig. 9). As a result, one has to inspect only about 50% of the complete feature to understand the B-tree. Furthermore, features that crosscut the B-tree implementation can be easily inspected by browsing the according separated implementations. On the other hand, to understand a particular feature sometimes also the code of other features has to be considered. Nevertheless, there are usually still features that can be ignored. For example, to understand the recovery implementation of the B-tree, a programmer has to understand the features B-tree and Recovery but can ignore all other features that crosscut the B-tree, which make up about 25% (cf. Fig. 9).

FOP can also have a negative effect on readability of the source code when comparing it to techniques that do not support customization. For example, if hook methods are used to introduce extension points into methods, these may degrade the comprehensibility and do not completely follow the principle of separation of concerns. Thus, interactions between features cannot be completely modularized if these occur at the level of statements within methods. In this case, the decomposed source code may be difficult to understand since the program flow switches between a number of methods and refinements. On the other hand, hook methods can improve comprehensibility since they can avoid decomposition of a method into a high number of smaller methods. Finally, the degradation of readability is a result of increasing customizability and can not only be found in FOP.

C/C++ macros can also be used to implement functionality that is only available if the according feature is present in a program variant.[11] Actually, this is the way preprocessor statements should be used in practice to increase readability of the source code [14]. Hook methods are similar to macros in the sense that they define functionality depending on the configured features. Comparing hook methods and macros, there is no difference with respect to readability of the source code in general. In contrast to macros, hook methods can be easily refined by multiple features when using FeatureC++. Furthermore, hook methods are part of the programming language and the type system. They support type safety, allow for overloading, and are encapsulated in the corresponding class. Additionally, complex `#ifdef` constructs, e.g., nesting, cannot be avoided when using macros since they are still needed for macro definitions. In case of Berkeley DB, 66% of the source code are part of optional or alternative features and are accessed via `#ifdef` constructs or macros obfuscating most of the source code. Using FeatureC++, we are able to properly modularize such code into feature modules and separate it from other features.

6.3. Comparison to other approaches

As discussed in Section 2, there are different approaches for development of customizable data management software. In the following, we review these approaches and compare them to our approach. Specifically, we compare properties like *separation of concerns*, which are related to the software development process, and properties like *footprint*, that are especially important for resource constrained environments. The result of our comparison is depicted in Table 3. All values are derived from our experience with Berkeley DB or result from analysis of previous research. The comparison should be considered only as an evaluation of approaches with respect to their applicability for developing customizable data management systems for the embedded domain and not as a general evaluation of these approaches. Some of the analyzed properties are difficult to evaluate and cannot be generalized. We begin with an overview of the compared properties:

- *Performance and footprint* represent the impact of the respective approach on execution time and binary size of an application. A negative value (empty dot) means a negative impact and a positive value means that performance might be increased or binary size might be reduced due to optimizations.

[11] In this case `#ifdef`-statements are used to define macros depending on configured features.

- *Granularity* means the potential degree of decomposition of a software. A fine granularity results in high customizability which is needed for highly resource constrained environments. For example, a decomposition of the Berkeley DB core into further features might be necessary for deeply embedded devices to reduce the binary size but is usually not required for server systems. In Table 3, better values for granularity (filled dots) mean better support for fine-grained decomposition.
- *Separation of concerns* means that different concerns (i.e., features) of an application can be separated, e.g., into different source code modules. Separating and modularizing functionality of an application is important when independent development is needed (e.g., development of extensions by a third party) and it simplifies removal or exchange of functionality with a different implementation. Especially crosscutting concerns are often difficult to separate from other code [23].
- *Comprehensibility of source code* is difficult to measure. When estimating comprehensibility we consider only our experience with Berkeley DB and do not intend to rate the approaches in general. Maintainability of software is affected by its comprehensibility and not considered separately here.

Based on these properties, we rated the approaches introduced in Section 2 as shown in Table 3. The following overview reviews them to explain the ratings.

6.3.1. Kernel systems

Kernel systems have been used to develop server DBMS and might also be applied to embedded systems but achieve only coarse-grained customizability because a fixed DBMS core is used that provides functionality required by most DBMS [21]. Depending on the concrete extensibility mechanism, e.g., using function pointers or virtual methods, there might be a negative impact on performance and footprint. Kernel systems are insufficient to provide DBMS for highly differing application scenarios in the embedded domain in general. However, a small kernel system might provide a foundation for implementing DBMS for a range of applications with similar requirements for embedded devices. Furthermore, FOP might be used as an extensibility mechanism for kernel systems thus avoiding performance degradation. Comprehensibility of the source code is rated as neutral since there are no negative effects as found for the C preprocessor. In contrast to the other approaches, kernel systems are not a general mechanism for achieving customizability. They obey similar properties like components and components can also be used to implement extensions for kernel systems [21].

6.3.2. Components

In contrast to kernel systems, components provide a better customizability since also the kernel of a DBMS can be decomposed into multiple components. The mechanisms used to provide extensibility, however, are similar to kernel systems (e.g., using virtual methods) and introduce an overhead when considering performance and footprint [21,13]. Hence, the granularity of decomposition that can be achieved with components is limited due the introduced runtime overhead which dramatically increases for very small components [11]. Furthermore, components can be used only to modularize functionality that is limited to non-crosscutting concerns, i.e., code that can be well localized [23,13]. We think that the comprehensibility of a component approach is similar to that of FOP, i.e., better compared to preprocessors, due to improved separation of concerns. However, in case of small modules and fine-grained extensibility the comprehensibility might be degraded as already discussed for FOP, e.g., caused by hook methods that are used in both approaches.

6.3.3. Components and AOP

In order to avoid some drawbacks of the component approach, AOP can be used to implement crosscutting concerns [23]. This was also shown by Nyström et al. who modularized crosscutting concerns like concurrency control to allow modularization of the transaction management subsystem [13]. The result is improved separation of concerns and fine-grained customizability without negative impact on performance and footprint. Unfortunately, the comprehensibility of source code may be degraded when using aspects as it is suggested by studies that analyzed AOP [59,51,60]. However, currently there is too less known about the comprehensibility of AOP code for a detailed evaluation.

6.3.4. C preprocessor (CPP)

When using `#ifdef` statements and the C preprocessor there is no negative impact on performance or resource consumption as long as the approach is used correctly. The reason is that the preprocessor is working on the source code and does not introduce additional overhead as it is the case for components. A negative impact on performance as it might be caused by using function pointers to provide extension points, e.g., in Berkeley DB, is not a direct result of the use of preprocessors. Nevertheless, the use of function pointers suggests that developers strive for simpler mechanisms for customizability that provide some level of abstraction, i.e., using function pointers that can be easily exchanged. The C preprocessor allows a developer to provide fine-grained customizability by modularizing even single statements which is not possible with any of the other analyzed approaches. In contrast to components or FOP, the C preprocessor does not directly support separation of concerns in source code. This is partially possible using separately defined macros instead of scattered `#ifdef` statements but it still provides less separation of concerns compared to components or FOP. The lack of separation of concerns is also the reason for reduced maintainability and comprehensibility of the source code [15]. Comprehensibility is significantly degraded by nested `#ifdefs` which might be reduced by using macros but cannot be completely avoided [14].

6.3.5. FOP

Similar to the C preprocessor, static composition of feature modules avoids any negative impact on performance or footprint when using FOP [61]. Feature modules are similar to components but they can be also used to modularize crosscutting concerns and static composition of feature modules does not inhibit method inlining as virtual methods do. In contrast to a combination of components and AOP, FOP provides a uniform mechanism for customization, i.e., *refinements*, and not a complex combination of different mechanisms, i.e., aspects and virtual methods. However, it is not possible to introduce single statements into the middle of a method, as it is possible with C preprocessor directives, and hook methods have to be used instead. We argue that this is needed only seldom but a combination of annotative approaches with FOP might be beneficial in this case [62]. In contrast to all other analyzed approaches, we could show that *safe composition* is possible when using FOP, i.e., we can ensure that every generated variant is correct with respect to syntax and semantics of the used programming language [63]. Separation of concerns can improve the comprehensibility of large software systems because not the whole source code has to be inspected to understand a particular feature [23]. Nevertheless, separation of concerns may also have a negative impact on code comprehensibility, e.g., when hook methods have to be used. For that reason we think that tool support, similar to the virtual separation of concerns for annotative approaches [58], can further improve the comprehensibility of FOP code.

The presented comparison does not mean that a particular approach cannot be used for the embedded domain at all or that FOP is always the best approach. For example, even though components introduce an overhead with respect to resource consumption they can be used for devices that provide more memory or when lower performance and higher power consumption are not important. On the other hand, it might be better for a company to stay with a preprocessor based approach instead of FOP in order to avoid the additional effort for a migration, e.g., to FeatureC++ [62]. Hence, it has to be analyzed for each concrete scenario which approach should be applied. The comparison presented above provides a basic guideline for such decisions.

6.4. Summary

The key results of our case study can be summarized as follows:

- We used a refactoring approach to decompose a medium size DBMS into features to reduce the code size of tailored instances.
- We modularized also complex crosscutting features, e.g., transaction management, that crosscut up to 11 other features.
- By using FOP, we could achieve high customizability of Berkeley DB based on 24 optional features ($> 400,000$ variants).
- The binary size of Berkeley DB did not increase when applying an FOP approach. By removing unused features we could decrease the binary size by about 50%.
- Comparing different variants of the refactored Berkeley DB to the original version, the performance did not decrease even though a more fine-grained decomposition was used. By extending dynamic configuration with static configuration we could achieve a performance improvement of about 16% for a reading benchmark.

These results show that when FOP is used, it is indeed appropriate to build highly customizable applications for use in embedded systems. In resource constrained environments, the requirements on resource consumption and specialization are most important. Both can be satisfied by using FOP. The still existing deficiencies regarding comprehensibility of the source code in case of hook methods are a result of the *design for reuse* and also apply for alternative approaches, e.g., for macros in C/C++ code or for hooks as commonly applied in frameworks [50].

7. Conclusion and perspective

We have presented an approach to customize and downsize DBMS in order to tailor data management for embedded systems. We used FOP to generate specialized DBMS based on a common architecture and code base. The fine-grained customizability supported by FOP is the basis for tailoring DBMS to satisfy the resource constraints of embedded systems.

We have evaluated our approach by means of the medium-sized commercial DBMS Berkeley DB. While other approaches for developing customizable data management software have drawbacks regarding resource consumption and customizability, we have shown that performance is not degraded but can be enhanced by about 16% when comparing reading operations with the original DBMS. Furthermore, we reduced the minimal binary size of the DBMS by about 50% by removing features. Comparing FOP to other approaches, it combines high customizability, similar to that of preprocessors, with proper modularity and structure of the source code as it is found in component-based approaches. Our study shows that FOP is well suited to replace preprocessors usually applied to provide customizability for embedded systems and seems promising to be applied to other domains.

The presented approach is also promising for developing customizable DBMS in general because it overcomes the limitations of components and kernel systems. Nevertheless, there are still several interesting open issues regarding the application of FOP to DBMS development in the embedded domain and also in other domains as we discuss in the following.

7.1. Granularity of decomposition

Currently, there is little known about the appropriate granularity for decomposing DBMS. While FOP supports a fine-grained decomposition this might be inappropriate for large systems. In other case studies, we developed customizable data management solutions from scratch [44,64–66]. For example, we used FeatureC++ to develop FAME-DBMS, a data management system for embedded devices [65].[12] FAME-DBMS is a very small DBMS product line intended for use on devices with highly constrained resources, e.g., with a program memory of less than 128 KB. We could show that an extremely fine-grained decomposition of a DBMS is possible but increases the development effort. For example, by increasing the number of features also the number of feature interactions increases [67,68] and thus the development effort. A fine-grained decomposition is essential to generate very small DBMS that are to be used in deeply embedded systems [44]; however, this is not required for larger DBMS. A mixed granularity might be a proper solution. In order to find an appropriate granularity for DBMS product lines, further analysis of DBMS features and their use in different applications has to follow. Therefore, we aim at a more detailed decomposition of Berkeley DB as well as an extension of the FAME-DBMS product line.

7.2. Query processing

Structured Query Language (SQL) grows with every new standard and supports many features while only a small subset is actually used by applications. Developments like *Structured Card Query Language (SCQL)* [69] and SQL extensions for sensor networks [19] or stream processing [70] show that the current SQL standard is not the appropriate solution for every use case and extensions are needed as well. In prior work, we could show that SQL (i.e., the grammar and the parser) can be decomposed with a feature-oriented approach which results in a product line of *SQL dialects* [71]. SQL dialects can be created for different domains, e.g., for sensor networks or stream processing systems, and can be the foundation for developing customizable DBMS [72].

7.3. DBMS architecture and other domains

DBMS architectures have to anticipate constantly changing requirements and are still in the focus of current research [2]. By combining tailor-made SQL dialects and customizable DBMS we think that one can build DBMS that can be fully tailored to an application domain or a special use case within a domain while providing high reuse. The need for tailor-made data management is not only apparent in the embedded domain but also in other domains [8,73]. Examples are tailor-made lock protocols for XML databases [74] or special solutions needed in the domain of distributed databases [75].

Creating a fully customizable DBMS to support tailor-made solutions for different domains is a challenging task. It involves customizability on all levels of a DBMS such as the query processor and optimizer, the transaction subsystem, or the storage system. Because of interdependencies between those subsystems, an architecture has to be developed that is able to handle the resulting complexity. For example, changing the query language affects the whole DBMS including the query optimizer which is highly connected with all other parts of a DBMS. New extensibility mechanisms as provided by feature-oriented programming can help in building such variable systems including a variable architecture. Based on the FAME-DBMS prototype, we are currently developing a customizable query engine to achieve customizability of a whole DBMS for resource constrained devices. In the long run, this could be done for many other domains as well.

Acknowledgements

We thank Christian Kästner, Martin Kuhlemann, and Norbert Siegmund for comments on drafts of this paper. Marko Rosenmüller is funded by German Ministry of Education and Research (BMBF), project number 01IM08003C. Sven Apel's work is funded partly by the German Research Foundation (DFG), project AP 206/2-1. The presented work is part of projects FAME-DBMS,[12] ViERforES,[13] and FeatureFoundation.[14]

References

[1] D. Tennenhouse, Proactive computing, Communications of the ACM (CACM) 43 (5) (2000) 43–50.
[2] T. Härder, DBMS architecture – still an open problem, in: Datenbanksysteme in Business, Technologie und Web (BTW), 2005, pp. 2–28.
[3] L. Casparsson, A. Rajnak, K. Tindell, P. Malmberg, Volcano – a revolution in on-board communications, in: Volvo Technology Report, No. 1, 1998, pp. 9–19.
[4] D. Nyström, A. Tešanović, C. Norström, J. Hansson, N.-E. Bånkestad, Data management issues in vehicle control systems: a case study, in: Proceedings of Euromicro Conference on Real-Time Systems (ECRTS), IEEE Computer Society Press, 2002, pp. 249–256.
[5] M. Weiser, Some computer science issues in ubiquitous computing, Communications of the ACM (CACM) 36 (7) (1993) 75–84.
[6] B. Warneke, M. Last, B. Liebowitz, K.S.J. Pister, Smart dust: communicating with a cubic-millimeter computer, Computer 34 (1) (2001) 44–51.
[7] R.P. Feynman, There's plenty of room at the bottom, in: Feynman and Computation: Exploring the Limits of Computers, Perseus Books, 1998, pp. 63–76.

[12] http://fame-dbms.org/.
[13] http://vierfores.de/.
[14] http://fosd.de/FeatureFoundation/.

[8] M. Stonebraker, U. Cetintemel, One size fits all: an idea whose time has come and gone, in: Proceedings of the International Conference on Data Engineering (ICDE), 2005, pp. 2–11.
[9] C. Bobineau, L. Bouganim, P. Pucheral, P. Valduriez, PicoDMBS: scaling down database techniques for the smartcard, in: Proceedings of the International Conference on Very Large Data Bases (VLDB), Morgan Kaufmann, 2000, pp. 11–20.
[10] R. Sen, K. Ramamritham, Efficient data management on lightweight computing devices, in: Proceedings of the International Conference on Data Engineering (ICDE), IEEE Computer Society Press, 2005, pp. 419–420.
[11] S. Chaudhuri, G. Weikum, Rethinking database system architecture: towards a self-tuning RISC-style database system, in: Proceedings of the International Conference on Very Large Data Bases (VLDB), Morgan Kaufmann, 2000, pp. 1–10.
[12] K. Czarnecki, U. Eisenecker, Generative Programming: Methods Tools and Applications, Addison-Wesley, 2000.
[13] D. Nyström, A. Tešanović, M. Nolin, C. Norström, J. Hansson, COMET: a component-based real-time database for automotive systems, in: Proceedings of the Workshop on Software Engineering for Automotive Systems, IEEE Computer Society, 2004, pp. 1–8.
[14] H. Spencer, G. Collyer, #ifdef Considered harmful, or portability experience with C news, in: Proceedings of the USENIX Summer 1992 Technical Conference, 1992, pp. 185–197.
[15] I.D. Baxter, M. Mehlich, Preprocessor conditional removal by simple partial evaluation, in: Proceedings of the Working Conference on Reverse Engineering (WCRE), IEEE Computer Society Press, 2001, pp. 281–290.
[16] A. Tešanović, K. Sheng, J. Hansson, Application-tailored database systems: a case of aspects in an embedded database, in: Proceedings of International Database Engineering and Applications Symposium, IEEE Computer Society Press, 2004, pp. 291–301.
[17] C. Prehofer, Feature-oriented programming: a fresh look at objects, in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science, vol. 1241, Springer Verlag, 1997, pp. 419–443.
[18] D. Batory, J.N. Sarvela, A. Rauschmayer, Scaling step-wise refinement, IEEE Transactions on Software Engineering (TSE) 30 (6) (2004) 355–371.
[19] S.R. Madden, M.J. Franklin, J.M. Hellerstein, W. Hong, Tinydb: an acquisitional query processing system for sensor networks, ACM Transactions on Database Systems 30 (1) (2005) 122–173. <http://doi.acm.org/10.1145/1061318.1061322>.
[20] A. Geppert, S. Scherrer, K.R. Dittrich, KIDS: Construction of Database Management Systems based on Reuse, Tech. Rep. ifi-97.01, Department of Computer Science, University of Zurich, 1997.
[21] K.R. Dittrich, A. Geppert, Component Database Systems, dpunkt.Verlag, 2001.
[22] T.J. Biggerstaff, The library scaling problem and the limits of concrete component reuse, in: Proceedings of the International Conference on Software Reuse (ICSR), IEEE Computer Society, 1994, pp. 102–109.
[23] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science, vol. 1241, Springer Verlag, 1997, pp. 220–242.
[24] D. Batory, J. Thomas, P2: a lightweight DBMS generator, Journal of Intelligent Information Systems (JIIS) 9 (2) (1997) 107–123.
[25] D. Batory, J.R. Barnett, J.F. Garza, K.P. Smith, K. Tsukuda, B.C. Twichell, T.E. Wise, GENESIS: an extensible database management system, IEEE Transactions on Software Engineering (TSE) 14 (11) (1988) 1711–1730.
[26] Y. Coady, G. Kiczales, M. Feeley, G. Smolyn, Using AspectC to improve the modularity of path-specific customization in operating system code, in: Proceedings of the European Software Engineering Conference, ACM Press, 2001, pp. 88–98.
[27] Y. Coady, G. Kiczales, Back to the future: a retroactive study of aspect evolution in operating system code, in: Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD), ACM Press, 2003, pp. 50–59.
[28] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, W. Schröder-Preikschat, A quantitative analysis of aspects in the eCos kernel, in: Proceedings of the International EuroSys Conference (EuroSys), ACM Press, 2006, pp. 191–204.
[29] C. Zhang, H.-A. Jacobsen, Quantifying aspects in middleware platforms, in: Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD), ACM Press, 2003, pp. 130–139.
[30] C. Zhang, H.-A. Jacobsen, Resolving feature convolution in middleware systems, in: Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM Press, 2004, pp. 188–205.
[31] A. Colyer, A. Clement, Large-scale AOSD for middleware, in: Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD), ACM Press, 2004, pp. 56–65.
[32] M. Odersky, M. Zenger, Scalable component abstractions, in: Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM Press, 2005, pp. 41–57.
[33] K.J. Lieberherr, D. Lorenz, J. Ovlinger, Aspectual collaborations – combining modulesand aspects, The Computer Journal 46 (5) (2003) 542–565.
[34] M. Mezini, K. Ostermann, Variability management with feature-oriented programming and aspects, in: Proceedings of the International Symposium on Foundations of Software Engineering (FSE), ACM Press, 2004, pp. 127–136.
[35] S. Apel, T. Leich, G. Saake, Aspectual feature modules, IEEE Transactions on Software Engineering (TSE) 34 (2) (2008) 162–180.
[36] S. Apel, T. Leich, M. Rosenmüller, G. Saake, FeatureC++: on the symbiosis of feature-oriented and aspect-oriented programming, in: Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE), Lecture Notes in Computer Science, vol. 3676, Springer Verlag, 2005, pp. 125–140.
[37] D. Batory, L. Coglianese, M. Goodwin, S. Shafer, Creating reference architectures: an example from avionics, in: Proceedings of the Symposium on Software Reusability (SSR), ACM Press, 1995, pp. 27–37.
[38] D. Batory, S. O'Malley, The design and implementation of hierarchical software systems with reusable components, ACM Transactions on Software Engineering and Methodology (TOSEM) 1 (4) (1992) 355–398.
[39] D. Batory, C. Johnson, B. MacDonald, D.v. Heeder, Achieving extensibility through product-lines and domain-specific languages: a case study, ACM Transactions on Software Engineering and Methodology (TOSEM) 11 (2) (2002) 191–214.
[40] R. Lopez-Herrejon, D. Batory, From Crosscutting Concerns to Product Lines: A Function Composition Approach, Tech. Rep. TR-06-24, University of Texas at Austin, 2006.
[41] A.F. Garcia, C. Sant'Anna, C. Chavez, V.T. da Silva, C.J.P. de Lucena, A. von Staa, Separation of concerns in multi-agent systems: an empirical study, in: Software Engineering for Large-Scale Multi-Agent Systems (SELMAS), Springer Verlag, 2003, pp. 49–72.
[42] S. Trujillo, D. Batory, O. Diaz, Feature refactoring a multi-representation program into a product line, in: Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE), ACM Press, 2006, pp. 191–200.
[43] B. Xin, S. McDirmid, E. Eide, W.C. Hsieh, A Comparison of Jiazzi and AspectJ for Feature-Wise Decomposition, Tech. Rep. UUCS-04-001, School of Computing, The University of Utah, 2004.
[44] T. Leich, S. Apel, G. Saake, Using step-wise refinement to build a flexible lightweight storage manager, in: Proceedings of the East-European Conference on Advances in Databases and Information Systems (ADBIS), Lecture Notes in Computer Science, vol. 3631, Springer Verlag, 2005, pp. 324–337.
[45] E.W. Dijkstra, On the role of scientific thought, in: Selected Writings on Computing: A Personal Perspective, Springer Verlag, 1982, pp. 60–66.
[46] S. Apel, T. Leich, M. Rosenmüller, G. Saake, Combining feature-oriented and aspect-oriented programming to support software evolution, in: Proceedings of the Second ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'05), in: 19th European Conference on Object-Oriented Programming (ECOOP'05), 2005, pp. 3–16.
[47] S. Apel, T. Leich, G. Saake, Aspectual mixin layers: aspects and features in concert, in: Proceedings of the International Conference on Software Engineering (ICSE), ACM Press, 2006, pp. 122–131.
[48] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.
[49] B. Stroustrup, C and C++: siblings, The C/C++ Users Journal 20 (7) (2002) 28–36.
[50] G. Hedin, J.L. Knudsen, On the role of language constructs for framework design, ACM Computing Surveys (CSUR) (2000) 8–12.

[51] C. Kästner, S. Apel, D. Batory, A case study implementing features using AspectJ, in: Proceedings of the International Software Product Line Conference (SPLC), 2007, pp. 223–232.
[52] S. Apel, C. Lengauer, B. Möller, C. Kästner, An algebra for features and feature composition, in: AMAST, 2008, pp. 36–50.
[53] T. Leich, S. Apel, L. Marnitz, G. Saake, Tool support for feature-oriented software development – FeatureIDE: an eclipse-based approach, in: Proceedings of the OOPSLA Workshop on Eclipse Technology eXchange (ETX), ACM Press, 2005, pp. 55–59.
[54] C. Kästner, T. Thüm, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, S. Apel, FeatureIDE: tool framework for feature-oriented software development, in: Proceedings of the International Conference on Software Engineering (ICSE), IEEE Computer Society Press, 2009, pp. 611–614. formal Demonstration paper.
[55] D.L. Parnas, Designing software for ease of extension and contraction, IEEE Transactions on Software Engineering (TSE) SE-5 (2) (1979) 264–277.
[56] D.L. Parnas, Software product-lines: what to do when enumeration wont work, in: Proceedings of the Workshop on Variability Modelling of Software-intensive Systems, 2007, pp. 9–14.
[57] M. Rosenmüller, M. Kuhlemann, N. Siegmund, H. Schirmeier, Avoiding variability of method signatures in software product lines: a case study, in: GPCE Workshop on Aspect-Oriented Product Line Engineering (AOPLE), 2007, published at the workshop Web site: <http://www.softeng.ox.ac.uk/aople/>.
[58] C. Kästner, S. Apel, M. Kuhlemann, Granularity in software product lines, in: Proceedings of the International Conference on Software Engineering (ICSE), ACM Press, 2008, pp. 311–320.
[59] F. Steimann, The paradoxical success of aspect-oriented programming, in: Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM Press, 2006, pp. 481–497.
[60] M. Kuhlemann, C. Kaestner, Reducing the complexity of AspectJ mechanisms for recurring extensions, in: Second GPCE Workshop on Aspect-Oriented Product Line Engineering (AOPLE), 2007, pp. 14–19, published at the workshop Web site: <http://www.softeng.ox.ac.uk/aople/>.
[61] M. Rosenmüller, N. Siegmund, S. Apel, G. Saake, Code generation to support static and dynamic composition of software product lines, in: Proceedings of the 7th International Conference on Generative Programming and Component Engineering (GPCE), ACM Press, 2008, pp. 3–12.
[62] C. Kästner, 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), no. MIP-0802, Department of Informatics and Mathematics, University of Passau, 2008, pp. 35–40.
[63] S. Apel, C. Kästner, C. Lengauer, Feature featherweight java: a calculus for feature-oriented programming and stepwise refinement, in: Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE), ACM Press, 2008, pp. 101–112.
[64] M. Pukall, T. Leich, M. Kuhlemann, M. Rosenmüller, Highly configurable transaction management for embedded systems, in: AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software, 2007.
[65] M. Rosenmüller, N. Siegmund, H. Schirmeier, J. Sincero, S. Apel, T. Leich, O. Spinczyk, G. Saake, FAME-DBMS: tailor-made data management solutions for embedded systems, in: EDBT'08 Workshop on Software Engineering for Tailor-made Data Management (SETMDM), 2008, pp. 1–6.
[66] G. Saake, M. Rosenmüller, N. Siegmund, C. Kästner, T. Leich, Downsizing data management for embedded systems, Egyptian Computer Science Journal (ECS) 31 (1) (2009) 1–13.
[67] C. Kästner, S. Apel, S.S. ur Rahman, M. Rosenmüller, D. Batory, G. Saake, On the impact of the optional feature problem: analysis and case studies, in: Proceedings of the 13th International Software Product Line Conference (SPLC), Software Engineering Institute, 2009.
[68] J. Liu, D. Batory, C. Lengauer, Feature-oriented refactoring of legacy applications, in: Proceedings of the International Conference on Software Engineering (ICSE), ACM Press, 2006, pp. 112–121.
[69] International Organization for Standardization (ISO), Part 7: Interindustry Commands for Structured Card Query Language (SCQL), in: Identification Cards – Integrated Circuit(s) Cards with Contacts, ISO/IEC 7816-7, 1999.
[70] S. Zdonik, N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, M. Cherniack, U. Cetintemel, R. Tibbetts, Towards a streaming SQL standard, Proceedings of the VLDB Endowment 1 (2) (2008) 1379–1390.
[71] S. Sunkle, M. Kuhlemann, N. Siegmund, M. Rosenmüller, G. Saake, Generating highly customizable SQL parsers, in: Workshop on Software Engineering for Tailor-made Data Management (SETMDM), 2008, pp. 29–33.
[72] M. Rosenmüller, C. Kästner, N. Siegmund, S. Sunkle, S. Apel, T. Leich, G. Saake, SQL à la Carte – toward tailor-made data management, in: 13. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW), 2009, pp. 117–136.
[73] M. Stonebraker, C. Bear, U. Çetintemel, M. Cherniack, T. Ge, N. Hachem, S. Harizopoulos, J. Lifter, J. Rogers, S.B. Zdonik, One size fits all? part 2: benchmarking studies, in: Third Biennial Conference on Innovative Data Systems Research, 2007, pp. 173–184.
[74] M. Haustein, T. Härder, Optimizing lock protocols for native xml processing, Data and Knowledge Engineering (DKE) 65 (1) (2008) 147–173.
[75] L. Zhu, Y. Tao, S. Zhou, Distributed skyline retrieval with low bandwidth consumption, IEEE Transactions on Knowledge and Data Engineering 21 (3) (2009) 384–400.

Marko Rosenmüller received his Diploma in Computer Science from the University of Magdeburg, Germany in 2005. From 2000 to 2006 he was a software developer at the icubic AG in Magdeburg. Since 2006 he is a Ph.D. student at the University of Magdeburg. His research interests include software product lines, tailor-made data management, and programming languages for product line development.

Sven Apel is a post-doctoral associate at the Chair of Programming at the University of Passau, Germany. He received a Ph.D. in Computer Science from the University of Magdeburg, Germany in 2007. His research interests include advanced programming paradigms, software product lines, and algebra for software construction.

Thomas Leich is currently working toward the Ph.D. degree in Computer Science at the University of Magdeburg, Germany. He is the head of the Department of Applied Informatics at the Metop Research Institute, Magdeburg, Germany. His research interests are tailor-made and embedded data management and software product lines.

Gunter Saake is a full professor of Computer Science. He is the head of the Database and Information Systems Group at the University of Magdeburg, Germany. His research interests include database integration, tailor-made data management, object-oriented information systems, and information fusion. He is a member of the IEEE Computer Society.