

# Highly Configurable Transaction Management for Embedded Systems

Mario Pukall, Thomas Leich, Martin Kuhlemann, and Marko Rosenmueller  
School of Computer Science, University of Magdeburg  
P.O. Box 4120  
D-39016 Magdeburg, Germany  
{pukall, leich, kuhlemann, rosenmueller}@iti.cs.uni-magdeburg.de

## ABSTRACT

Embedded systems are an important field of research and will gain momentum in the near future. Many of these systems require data management functionality. Due to the resource constraints in embedded environments a high customizability of the data management functionality is required that depends on the application context. Whereas storage methods and index structures can be designed to be customizable, the fine grained modularization of transaction management remains problematic. The strong interaction of transaction management functionality with other data management components makes it difficult to separate it from the remaining system. In this paper we introduce an approach for the development of highly configurable transaction management systems. The main focus of our approach is the modularization of parts of the transaction management using advanced programming paradigms. We will show that Aspectual Mixin Layers, a combination of feature-oriented and aspect-oriented programming is the appropriate technique to implement a highly configurable transaction management.

## 1. INTRODUCTION

Embedded computer systems are important for many industries and their use will increase in the future. For instance, many capabilities of modern automobiles are monitored or even completely controlled by embedded systems. Along with the increasing functionality to control and monitor these embedded systems a growing amount of data has to be processed. A survey by Volvo states that the amount of data in an automobile increases by 7 to 10 percent each year [8]. A high specialization of these embedded systems implies varying requirements regarding the data management. Current database management systems are usually less specialized and provide a lot more functionality than needed. Whereas certain index methods or optimizing strategies are configurable, it is almost impossible to replace and tailor transaction management components.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Workshop ACP4IS '07 March 12-13, 2007 Vancouver, British Columbia, Canada

Copyright 2007 ACM 1-59593-657-8/07/03 ...\$5.00.

In this paper we present a prototypical implementation of a highly configurable transaction management system including concurrency control and recovery management. Although research is done in this direction the crosscutting nature of the code in respect of the transaction management prohibited a modular and configurable implementation in recent approaches. Due to current developments in the software engineering domain, the modularization of transaction management becomes promising. TEŠANOVIĆ ET AL. proposes the application of aspect-oriented programming (AOP) to implement transaction management for COMET DBMS [7]. AOP is known to have limited extensibility of modules and does not provide adequate modularization of crosscutting concerns [22, 20]. These issues cause the AOSD evolution paradox and thus decrease reusability of aspect-oriented source code [23]. Other well known problems regard the preparation of the object-oriented code for the application of aspects [13]. Feature-oriented programming (FOP) is another modularization technique that emerged concurrently to AOP [3, 5]. Since FOP lacks implementing homogenous and advanced crosscutting concerns, it is not the appropriate technique to be used in embedded environments. Furthermore, AOP and FOP have different strengths and weaknesses. Thus, we propose aspectual mixin layers (AML) that combine both approaches [22]. In this paper we evaluate AML by extending a storage manager with a transaction management system.

- We will show that AML support the development of modular and extensible transaction management systems.

## 2. SOFTWARE ENGINEERING BACKGROUND

The modularization of program functionality has been a subject of software engineering research for decades. In this context, FOP and AOP are currently discussed intensively. Both methods adopt the paradigm of object-oriented programming and extend the existing software structuring methods in different ways.

### 2.1 Feature-Oriented Programming

FOP studies the modularity of *features* in product lines. A feature in this context is an increment in program functionality [5]. The idea of FOP is to synthesize software (individual programs) by composing features (a.k.a. *feature modules*). Typically, features refine the content of other features in an incremental fashion. Hence, the term *refinement* refers to

the set of changes a feature applies to a code base. Adding features incrementally, called *stepwise refinement*, leads to conceptually layered software designs. For simplicity, we use the terms feature and feature module synonymously.

*Mixin layers* is one approach to implement features [24, 5]. The basic idea is that features are seldomly implemented by single classes (or aspects). Typically, a feature implements a *collaboration* [17], which is a collection of roles represented by mixins that cooperate to achieve an increment in program functionality.

FOP aims at abstracting and explicitly representing such collaborations. Hence, it stands in the long line of prior work on object-oriented design and role modeling [9].

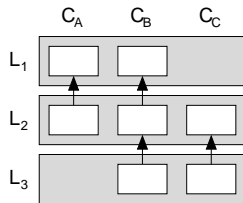


Figure 1: Stack of three mixin layers.

A mixin layer is a module that encapsulates fragments of several different classes (roles) so that all fragments are composed consistently. Figure 1 depicts a stack of three mixin layers ( $L_1 - L_3$ ) in top down order. These mixin layers crosscut multiple classes ( $C_A - C_C$ ). White boxes represent classes or mixins; gray boxes denote the enclosing feature modules; filled arrows refer to *mixin-based inheritance* [10] for composing mixins.

## 2.2 Aspect-Oriented Programming

AOP aims at separating and modularizing crosscutting concerns [11]. Using object-oriented mechanisms, crosscutting concerns result in tangled and scattered code [11, 12]. The idea of AOP is to implement crosscutting concerns as *aspects* where the core (non-crosscutting) features are implemented as components. Using *pointcuts* and *advice*, an aspect weaver glues aspects and components at *join points*. Pointcuts specify sets of join points in aspects and components, advice defines code that is applied to (or executed at) these points, and introductions (a.k.a. *inter-type declarations*) inject new members into classes. With aspects, a programmer is able to refine a program coherently at multiple join points. Typically, aspects introduce new members to existing classes and extend existing methods. Figure 2 shows two aspects ( $A_1, A_2$ ) that extend three classes at multiple join points (dashed arrows denote aspect weaving) in classes ( $C_A - C_C$ ).

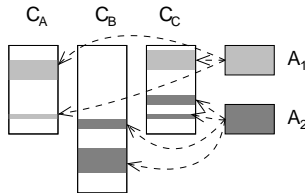


Figure 2: Two aspects extend three classes.

## 2.3 Symbiosis of Aspects and Features

Aspects and features in their current incarnation are intended for solving problems at different levels of abstraction [16, 14, 22]. Whereas aspects in AspectJ act on the level of classes and objects in order to modularize crosscutting concerns, features act on an architectural level. That is, a feature decomposes an object-oriented architecture into a composition of collaborations.

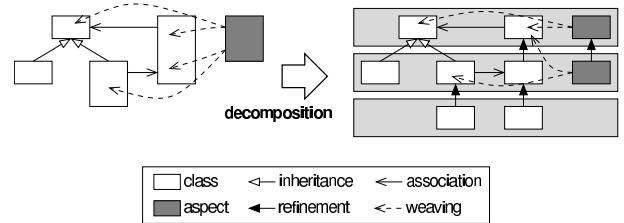


Figure 3: Feature-driven decomposition of an aspect-oriented architecture (features are depicted light-gray).

However both, AOP and FOP, are able to modularize crosscutting concerns and have their specific strength and weaknesses [16, 22]. *Homogeneous crosscutting concerns* represent functionality that causes similar or even equal source code elements that are distributed over large parts of a software. Implementing those homogeneous crosscuts with FOP results in replication of source code fragments. Contrary AOP handles these concerns in an effective way and avoids code replication. Another type of crosscutting concern are heterogeneous concerns that change existing source code in different ways. While this kind of concern can be easily implemented with FOP it results in a complex and incoherent implementation when using AOP since mostly multiple aspects and an introduction of classes is necessary. Furthermore, problems arise if considering the evolution of AOP approaches [23].

Hence the next logical step is to combine both approaches by decomposing *aspect-oriented architectures* (i.e., object-oriented architectures with aspects) and include them into features. Figure 3 shows an aspect-oriented architecture on the left and features that decompose and structure this architecture on the right. With this decomposition, a feature encapsulates fragments of classes *and* aspects that collaborate together to implement an increment in program functionality. Note that the original aspect was split into two pieces (a base and a subsequent refinement).

## 2.4 Aspectual Mixin Layers

Aspectual Mixin Layers integrate AOP and FOP. AML extends the notion of mixin layers by encapsulating both mixins and aspects (see Fig. 4).

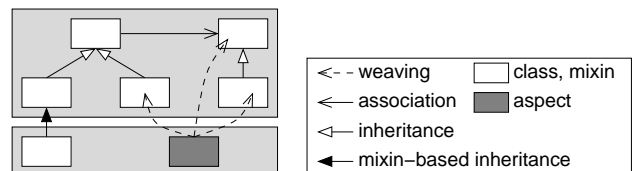


Figure 4: Aspectual Mixin Layers.

That is, an AML encapsulates both collaborating classes and aspects that contribute to a feature. An AML may refine a base program in two ways: (1) by using common mixin-composition or (2) by using aspect-oriented mechanisms, in particular pointcuts and advice. The most important contribution of AML is probably that programmers may choose the appropriate technique – mixins or aspects – that fits a given problem best.

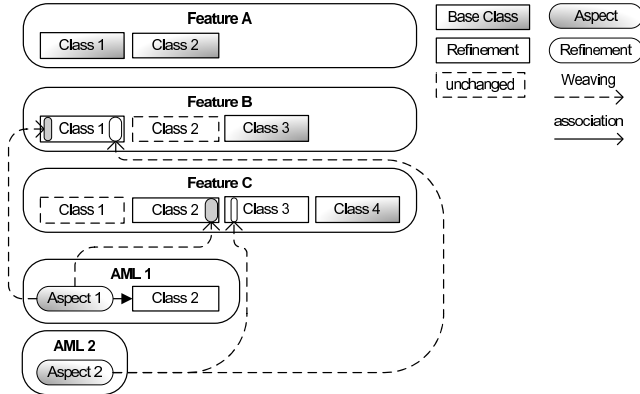


Figure 5: FOP and Aspectual Mixin Layers.

Figure 5 shows a modified representation of FOP and AML that we use in the remainder of this paper. For simplicity inheritance relations are omitted and classes that are not refined are explicitly displayed as rectangles with dotted lines (e.g., *Class 1* in *Feature C*).

## 2.5 FeatureC++

The combination of FOP and AOP via AML is realized in *FeatureC++*<sup>1</sup>. It is a feature-oriented extension to the programming language C++ that also allows the use of AML based on *AspectC++*<sup>2</sup>. For a detailed introduction to *FeatureC++* we refer to [21]. The following case study implementation is based on *FeatureC++*.

## 3. RELATED WORK

In the past, there have been few attempts to adopt the idea of modularization to transaction management. The solutions discussed in this paper should, as representatives of different classes, illustrate the general problems occurring in this area of research [15].

With *KIDS* [15], GEPPERT ET AL describe an approach to organize the concerns of DBMS within components, that are combined to a complete DBMS after their configuration. TEŠANOVIĆ ET AL. [6] describe with COMET-DBMS a toolkit-based DBMS, whose elements are functional software units, which can be recombined for each individual application. Especially specific transaction management functions are encapsulated in (a few) aspects in this approach (Figure 6).

The PLENTY-system, developed by HASSE, describes a kernel-based approach of a configurable DBMS [2]. The problem of this approach is that it requires detailed knowledge of the system to create customized DBMS.

<sup>1</sup>[http://wwwiti.cs.uni-magdeburg.de/iti\\_db/fcc/](http://wwwiti.cs.uni-magdeburg.de/iti_db/fcc/)

<sup>2</sup><http://aspectc.org>

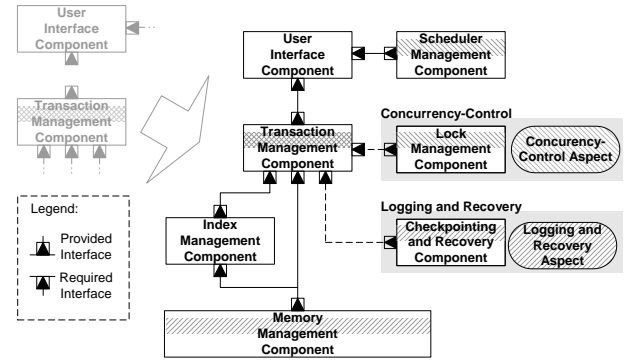


Figure 6: Architecture of COMET-DBMS.

The main problem of all these approaches is the insufficient configurability. In each approach, the portfolio of functional units only includes a few unspecific modules leaving little space for the development of highly customized solutions. Therefore we have to focus on the scalability using FOP and AOP in conjunction.

## 4. A CONFIGURABLE TRANSACTION MANAGEMENT SYSTEM

Components designed on the basis of FOP and AOP should allow the creation of extensively configurable transaction management systems as part of a DBMS. This requires the identification of the main features of transaction management. The aim is to choose the granularity of the components which encapsulates these features in a way, that each module fulfills a unique task (functionally identical modules with alternative implementations are also possible). This ensures adequate configurability.

### 4.1 Identification of Modules

Figure 7 illustrates an extract of the identified features. The initial layer represents a storage manager (*SM*). Excluding any other feature, we consider the storage manager as a minimal DBMS. Later enhancements of non transaction specific modules (like storage manager) are still possible. For instance we added locking functionality to the storage manager when we decided to provide 2-phase lock protocols as functions of transaction management (Figure 7 - layer BLS).

The second mixin layer (*BS* - base scheduler) provides basic functions of a transaction management system. This layer represents the minimum functionality of a transaction system. The following feature modules extend this basic feature with additional functions of classic transaction management systems like concurrency control and recovery. Since each component, starting with *SM* (storage manager) including *BRM* (basic recovery manager) introduces unique program code (heterogeneous crosscutting concerns), we recommend the usage of FOP.

The two lower layers illustrated in Figure 7 are AML which include the logging capabilities. The aspect calls the logging method of class *LogList* and affects different parts of each type of the class *Scheduler*. Encapsulating recurring and distributed calls of the logging method (homogeneous crosscutting concern) in aspects (Figure 7 - layers LRM, LRMP) offers the advantage of central maintenance of log-

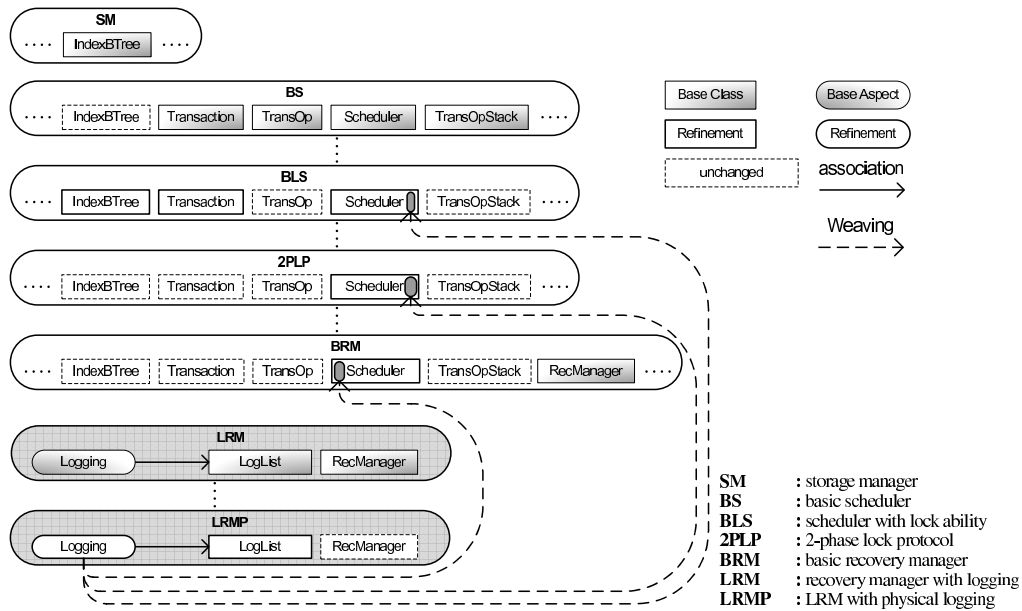


Figure 7: Extract layer architecture of the prototype.

ging functionality.

The first AML (Figure 7 - *LRM*) implements basic logging in combination with recovery functionality and is refined in AML *LRMP* that provides physical logging in addition.

In total, 13 features (10 Mixin Layers, 3 AML) have been identified and implemented for the transaction management. The 3 AML include the basic logging aspect, the physical and the logical logging (implemented as refinements of the basic aspect LRM). The components modeled here only include a few very interesting parts of transaction management and do currently not create a complete transaction system.

## 4.2 Implementation

As shown in the design of the layer model (Figure 7) the combined feature- and aspect-oriented implementation is a good solution as transaction management addresses heterogeneous as well as homogeneous crosscutting concerns. In the following we will describe the implementation of the study using a few examples.

**Features.** One of the main elements within the implemented prototype of the transaction management system is the class *Scheduler*. Figure 8 shows an extract of the class definition for the basic scheduler. In addition to the constructor definition it includes several method definitions to insert and parse transaction operations. In one of the following refinement steps the functionality of the 2-phase lock protocol is added to the scheduler. Figure 9 shows the corresponding class definition. The refinement is stated by the word *refines* (Line 1).

**Aspects.** The implemented aspect of the prototype encapsulates the commitment of the log entry and calls the methods provided by the class *LogList*. The class *Scheduler* requires a log entry each time a transaction operation has been committed. That is why each protocol (serial, lock-based, etc.) which is called to commit a schedule has to implement the logging call. If these are implemented with FOP each corresponding method of class *Scheduler* has to

```

1 class Scheduler {
2   public:
3     /*Scheduler-Constructor*/
4     Scheduler();
5     /*Input-Schedule*/
6     TransOpStack* getInput();
7     void setInput(TransOpStack*);
8     void addInput(TransOp*);
9     /*transaction vector*/
10    vector<Transaktion*> parseOps();
11 };
  
```

Figure 8: Basic Scheduler (BS)

```

1 refines class Scheduler {
2   public:aha
3     /*2Phase-Lock Protocol*/
4     void _2PL();
5 };
  
```

Figure 9: Scheduler with 2-phase lock protocol (2PLP)

be extended by hand. Figure 10 shows an extract of the implementation of the basic aspect. Lines 7 to 11 define the *Pointcut* at which the advice code (Line 16) is woven in. That is the call of the method *schedule\_pop(...)*. The AspectC++ keywords *result(...)*, *that(...)*, and *args(...)* help to get information about the environment into which the advice code is woven.

In [20], APEL ET. AL describe how to add new functions to the aspect code using stepwise refinement. Figure 11 illustrates the extension of the log entry to include status information (physical log). First the code of the basic aspect is executed (Line 4). This is followed by the calculation of the extension (indicated in Line 5).

## 4.3 Rule-Based Program Composition

```

1 aspect Logging {
2     virtual void logExecution(TransOp* n,
3                             Scheduler* s, TransOpStack* t){
4         /*call of LogList methods*/
5     }
6     /*Pointcut*/
7     pointcut log(TransOp* n, Scheduler* s,
8                 TransOpStack* t)
9         =execution
10            ("% Scheduler::schedule_pop()")
11            &&result(n)&&that(s)&&args(t);
12     /*Advice*/
13     advice log(n, s, t):after
14         (TransOp* n, Scheduler* s,
15          TransOpStack* t){
16         logExecution(n, s, t);
17     }
18 };

```

Figure 10: Aspect logging (LRM).

```

1 refines aspect Logging {
2     void logExecution(TransOp* n,
3                     Scheduler* s, TransOpStack* t){
4         super::logExecution(n, s, t);
5         /*storing After/Before-Image*/
6     }
7 };

```

Figure 11: Refinement physical log (LRMP).

The composition of concrete software products using FOP is done by selecting the required features. These features are specified in so-called *composition equations* [5]. The composition follows design rules that define dependencies between selected features. Composition equations can be incomplete<sup>3</sup> which results in a non-functioning transaction management system. To prevent this, it is necessary to identify the interdependencies of each layer (Mixin Layer, Aspectual Mixin Layer) with all remaining layers and to record it within a control equipment for an automated check of the validity of each composition equation [4]. The automated validation of program compositions is an important part within the overall concept, since only on this basis it is possible to design effective highly configurable transaction management systems. This ensures a lasting improvement of the configurability and reusability of systems created this way.

## 5. EVALUATION

In the last section we have presented a prototypical implementation of a simple transaction management system using AOP, FOP, and their combination AML. To analyze the expected benefits regarding configurability, reusability, and resource consumption we will now evaluate the developed prototype. We will also analyze the profit of the combination of AML.

**Configurability and Reusability.** FOP allows an easy generation of a concrete software by simply selecting the required features. This leads to a number of possible configurations based on the existing features and their dependencies that are expressed in design rules. Figure 12 shows a tree

<sup>3</sup>E.g., a component is missing, whose functionality is required by other components.

of combinations that illustrates the valid configurations for our prototype. The following symbols are used:

- **AddOn** - child optional selectable (operator of addition)
- **$\mathcal{P}$**  - power set over included layer
- **$\times$**  - cartesian product over included layer
- ***exor*** - exclusive OR
- **$\setminus$**  - Difference

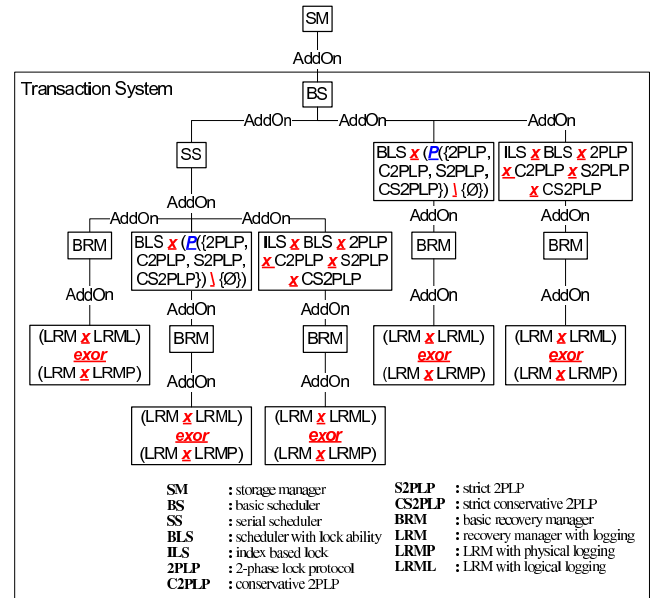


Figure 12: Tree of combinations.

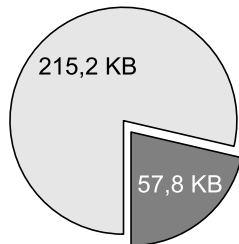
The calculation of the number of valid configurations is based on the idea that each path in the tree, starting with the root node (Storage Manager – SM), corresponds to a valid configuration and each visited node provides additional functionality that corresponds to a feature. In our case the minimal possible program is represented by the root node (SM) and implements a storage management system of a DBMS. To simplify the illustration some features have been combined into a single node. The calculation of the number  $C_T$  of valid and reasonable program compositions is based on the rules of set theory and results in the following equation:

$$C_T = 2 + (4) + (15 * (3)) + (4) + (15 * (3)) + (4) = 104$$

Based on only 13 features this value is a remarkable result and proves that the developed system provides a high level of configurability. If using the developed modules virtually every case scenario can be handled with a highly customized program composition. This is especially important in embedded environments, since resources are very constrained and applications have varying requirements towards a data management system.

Contrary, the existing approaches mentioned in section 3 allow only restricted configuration for complete DBMS. Figure 12 also illustrates the high reusability of the individual components since many of them can be used in different program configurations.

**Resources.** Based on high configurability a minimal size of a data management application can be achieved. Figure 13 illustrates this benefit that is essential for resource constraint environments. A functionally complete transaction system consisting of 13 features results in a binary size of 273 kilobyte. In contrast to this a system which only allows serial execution of transactions requires only 57.8 kilobyte. In general, the fine-grained configurability of the prototype allows to reduce the size of the binary code for every application that does not require a functionally complete transaction management system.



**Figure 13: Size of binary code of a system using complete transaction management (273 KB) and a system with serial scheduling (57.8 KB).**

**AOP vs. FOP.** Transaction management functionality crosscuts large parts of a data management system and leads to entangled source code. TEŠANOVIĆ ET AL. showed that AOP can be used to separate transaction management functionality and to achieve configurability [7]. But AOP is also known to have deficits regarding the modularization of crosscutting concerns [1, 14, 18] and FOP is often the better alternative to implement heterogeneous crosscuts [19].

With the implementation of our prototypical transaction management system we could demonstrate that FOP is an adequate technique to implement transaction management functionality with fine-grained customizability. We implemented 13 features and found that only 3 of them contained homogeneous crosscuts. This underlines that most of the source code can be implemented via FOP. To implement the homogeneous crosscuts we used the AML approach with one aspect that was refined in two subsequent features.

The vast use of FOP naturally results from the fact that the implementation of new features requires the writing of new program code which is so specific that it is only used once within the program [19]. In our case even the logging of the transaction execution is not entirely implemented as an aspect, but as a combination of features and aspects. The aspect code is mostly limited to method calls for writing log entries since these occur in large parts of the program. Hence, AOP cannot be abandoned since the implementation of homogeneous crosscutting concerns via FOP results in code replication.

In contrast to a pure AOP implementation our approach increases the modularity of the implemented features by combining aspects and classes that correspond to one feature. Furthermore the bounding of aspects to already existing source code enhances the evolvability [20]. Thus AML allow to benefit from both, AOP and FOP, while avoiding some of their deficits.

## 6. CONCLUSION

This article focused on the possibilities to modularize transaction management as a part of complex DBMS's in order to achieve high configurability and reusability. We have shown this by implementing a customizable transaction management system. The used approach to combine FOP and AOP provides an elegant solution, that allows the attempted modularization and reusability of the transaction management system. We examined 13 features including 3 AML in the prototype implementation of selected components of a transaction management system. Based on the developed components, 104 functionally different configurations can be derived using the established design rules. Thus a highly customized system can be provided for virtually every case scenario which preserves the resources of embedded systems. In addition, we revealed that transaction management mainly addresses heterogeneous crosscutting concerns which have to be implemented using FOP. The remaining homogeneous crosscutting concerns (logging) can be implemented with AML, thereby increasing modularity and improving evolvability unlike the exclusive use of AOP.

## 7. REFERENCES

- [1] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena and A. v. Staa. Modularizing Design Patterns with Aspects: A Quantitative Study. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*. ACM Press, 2005.
- [2] C. Hasse. *Inter- und Intratransaktionsparallelität in Datenbanksystemen: Entwurf, Implementierung und Evaluation eines Datenbanksystems mit Inter- und Intratransaktionsparallelität*. PhD thesis, Departement of Computer Science, ETH Zürich, 1995.
- [3] 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. Springer, 1997.
- [4] D. Batory. A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite. In *Generative and Transformational Techniques in Software Engineering, International Summer School (GTTSE)*, 2006.
- [5] D. Batory, J.N. Sarvela and A. Rauschmayer. Scaling Step-Wise Refinement. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2003.
- [6] D. Nyström, A. Tešanović, C. Norström and J. Hansson. The COMET Database Management System. Technical report, Mälardalen University, 2003.
- [7] D. Nyström, A. Tešanović, C. Norström and J. Hansson. COMET: A Component-Based Real-Time Database for Automotive Systems. In *Proceedings of the Workshop on Software Engineering for Automotive Systems at 26th International Conference on Softwareengineering (ICSE)*, Edinburgh, Scotland, 2004. IEEE Computer Society Press.
- [8] D. Nyström, A. Tešanović, C. Norström, J. Hansson and N-E. Bånkestad. Data Management Issues in Vehicle Control Systems: A Case Study. In

- Proceedings of the 14th Euromicro Conference on Real-Time Systems (ECRTS)*, 2002.
- [9] F. Steimann. On the Representation of Roles in Object-Oriented and Conceptual Modeling. *Data and Knowledge Engineering (DKE)*, 2000.
- [10] G. Bracha and W. R. Cook. Mixin-Based Inheritance. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) and the European Conference on Object-Oriented Programming (ECOOP)*. ACM Press, 1990.
- [11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science. Springer, 1997.
- [12] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
- [13] K. Gybels and J. Brichau. Arranging Language Features for More Robust Pattern-Based Crosscuts. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*. ACM Press, 2003.
- [14] K. J. Lieberherr, D. Lorenz and J. Ovlinger. Aspectual Collaborations – Combining Modules and Aspects. *The Computer Journal*, 46(5), 2003.
- [15] K.R. Dittrich and A. Geppert. Component Database Systems, 2000.
- [16] M. Mezini and 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.
- [17] M. VanHilst and D. Notkin. Using Role Components in Implement Collaboration-based Designs. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM Press, 1996.
- [18] R. Lopez-Herrejon, D. Batory and C. Lengauer. A Disciplined Approach to Aspect Composition. In *Proceedings of the International Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*. ACM Press, 2006.
- [19] S. Apel and D. Batory. When to Use Features and Aspects? A Case Study. In *Proceedings of ACM SIGPLAN 5th International Conference on Generative Programming and Component Engineering (GPCE)*, 2006.
- [20] S. Apel, C. Kästner, T. Leich, and G. Saake. Aspect Refinement. Technical Report 10, School of Computer Science, University of Magdeburg, Germany, 2006.
- [21] S. Apel, M. Rosenmüller, T. Leich and 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)*, volume 3676 of *Lecture Notes in Computer Science*. Springer, 2005.
- [22] S. Apel, T. Leich and G. Saake. Aspectual Mixin Layers: Aspects and Features in Concert. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM Press, 2006.
- [23] T. Tourwé, J. Brichau and K. Gybels. On the Existence of the AOSD-Evolution Paradox. In *Workshop on Software-Engineering Properties of Languages for Aspect Technologies (SPLAT)*, 2003.
- [24] Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2002.