Application-Tailored Database Systems: a Case of Aspects in an Embedded Database*

Aleksandra Tešanović Ke Sheng Jörgen Hansson
Department of Computer Science
Linköping University, Linköping, Sweden
{alete,jorha}@ida.liu.se

Abstract

Current techniques used to design and implement database systems do not provide support for efficient implementation of crosscutting concerns in the database software, e.g., failure detection, database policies such as concurrency control and scheduling, and synchronization. Aspect-oriented software development (AOSD) is a new technique that provides an efficient way of modularizing crosscutting concerns in software systems. In this paper we evaluate the effectiveness of applying AOSD to database systems thereby paving way for successful application of aspect languages to the database domain. Our focus is on embedded database systems, as a representative for a class of database systems. We show, by analyzing and re-engineering one commercial well-known embedded database system (Berkeley database), that aspect-orientation has promise, especially in enabling development of tailorable, maintainable, and evolvable database systems.

1. Introduction

Current techniques used to design and implement database systems do not provide support for efficient implementation of crosscutting concerns in the database software. Crosscutting concerns are features of the system that cannot cleanly be encapsulated into functions, modules, objects, and components. Typical examples of crosscutting concerns in database systems are synchronization, error handling, and logging and recovery.

Aspect-oriented software development (AOSD) has emerged as a new principle for software development that provides an efficient way of modularizing crosscutting concerns in software systems [9]. AOSD enables encapsulation of crosscutting concerns of a system in "modules", called aspects. The application of AOSD to separate concerns in database systems has promise as the use of aspects in a database system development would allow high reusability, tailorability and maintenance of the database software.

In this paper we address the problem of designing tailorable and maintainable databases by investigating the impact of using AOSD for database system development. Component-based databases [5, 16, 3, 8, 13, 7, 1], which can be partially or completely assembled from a pre-defined set of components with well-defined interfaces, are suited for tailoring a database system towards an application. However, there are aspects of database systems that cannot be encapsulated into components with well-defined interfaces, e.g., failure detection, synchronization, and database policies such as concurrency control. A database component is typically developed independently of other components and, therefore, each developed component has its crosscutting concerns independently implemented by the component developer. This can lead to code that is complex, inefficient, and difficult to develop and maintain, hence, making the process of changing and upgrading the database software difficult and error-prone.

The contribution of this paper is a case study that identifies benefits and drawbacks of applying aspect-orientation and aspect programming languages to the design and implementation of database systems. We show, by analyzing and re-engineering one commercial well-known embedded database system[1] (Berkeley database), that aspect-orientation has promise. We found that re-engineering the Berkeley database to support aspects improves maintain-

* This work is supported by the Swedish Foundation for Strategic Research (SSF) via the SAVE project and the ARTES network, and the Center for Industrial Information Technology (CENIIT) under contract 01.07.

1 In contrast to an application-embedded database hidden inside an application, an embedded database is device-embedded and resides in an embedded system.

ability, independent development of crosscutting concerns, and testability of the database software. Furthermore, tailorability and evolvability of the database are also improved. In the re-engineered database changes to the database software are localized into aspects, which improves comprehensibility by allowing to reason about different parts of the database software and their interaction separately.

The paper is organized as follows. Background information on AOSD with a focus on aspect language constructs is given in section 2. In section 3 we outline the problem addressed in the paper, i.e., limited experience when it comes to the impact of applying aspect languages to database software. We then introduce the case study of the Berkeley database in section 4, and provide the re-engineered solution of the database that supports aspects in section 5. Related work is discussed in section 6. The paper finishes with the main conclusions and directions for our future work in section 7.

2. Aspect-oriented software development

Typically, AOSD implementation of a software system has the following constituents: (i) components, written in a component language, e.g., C, C++, and Java; (ii) aspects, written in a corresponding aspect language, e.g., AspectC [4], AspectC++ [20], and AspectJ [2] developed for Java; and (iii) an aspect weaver, which is a special compiler that combines components and aspects.

Components used for system composition in AOSD are *white box* components. A white box component is a piece of code, e.g., traditional program, function, and method, completely accessible by the component user. In AOSD one can modify the internal behavior of a component by weaving different aspects into the code of the component. Aspects are commonly considered to be properties of a system that affect its performance or semantics, and that crosscut the functionality of the system [9].

In existing aspect languages, each aspect declaration consists of advices and pointcuts (see figure 1). A *pointcut* in an aspect language consists of one or more join points, and is described by a pointcut expression. A *join point* refers to a point in the component code where aspects should be woven, e.g., a method or a type (struct or union). Figure 1 shows the definition of a named pointcut `getLockCall`. This pointcut refers to all calls to the function `getLock()` and exposes a single integer argument to that call (this example is written in AspectC++.). Hence, `getLock()` is the join point in the program code. The syntax of the pointcut is illustrated in figure 2. The first two pointcuts (`call` and `execute`) match join points (i.e., places in the code of the program) that have the same signature as the join point m. While the `call` pointcut refers to the point in code

```
aspect printID{
  pointcut getLockCall(int lockId)=
    call("void getLock(int)")&&args(lockId);
  advice getLockCall(lockId):
    void after (int lockId){
      cout<<"Lock requested is"<<lockId<<endl;}
}
```

Figure 1. An example of the aspect definition

```
p ∈ {pointcuts}
m ∈ {function|method_signatures}
v ∈ {identifiers_with_types}
p::=call(m)|execute(m)|target(v)|args(v)|p&&p|p||p|!p
```

Figure 2. A typical pointcut syntax

where some function/method is called, the `execute` pointcut refers to the execution of the join point (i.e., after the call has been made and a function started to execute). The pointcuts `target` and `args` match any join point that has values of a specified type; in this case v. Operators &&, $||$, and ! logically combine or negate pointcuts.

An *advice* is a declaration used to specify the code that should run when the join points, specified by a pointcut expression, are reached. Different kinds of advices can be declared, such as: (i) *before advice*, which is executed before the join point, (ii) *after advice*, which is executed immediately after the join point, and (iii) *around advice*, which is executed in place of the join point. In figure 1 an example of an after advice is shown. With this advice each call to `getLock()` is followed by the execution of the advice code, i.e., printing of the lock id.

3. Crosscutting concerns in databases

In this section we show that there is limited experience about benefits and drawbacks of using aspects in the database domain (section 3.1). This is followed by a discussion on different types of aspects in the database domain in section 3.2. Finally (in section 3.3) we give the overall goal of the paper and discuss the methodology we employ to reach the goal.

3.1. Problem description

Modularizing crosscutting concerns in software systems using aspects is an open research challenge. The impact of applying AOSD to different application domains has been investigated intensively in recent years; for example, case studies on applying AOSD to operating systems [12, 4] and distributed real-time dependable systems [6] have been made, and benefits and drawbacks in these domains have been identified.

We summarize the main benefits of using aspect languages for developing software systems as follows:

(B1) independent development, implying that aspects of a software system can be developed independently with clear interfaces towards the software with which aspects should be woven;

(B2) localized changes, implying that a software system can easily be modified by simply modifying the code of the aspect that is maintained in a separate module;

(B3) extensibility, implying that a software system can be extended with new functional and non-functional features by defining and weaving new aspects;

(B4) improved comprehensibility, implying that having different features of a software system encapsulated into aspects allows reasoning about different parts of the software and their interaction separately;

(B5) tailorability, allowing software to be tailored towards systems with which the software is embedded;

(B6) improved testability, implying that the software developed independently of additional, typically non-functional, features introduced by aspects can be more efficiently tested (as less software should be tested); and

(B7) improved maintainability of software, implying that aspects encapsulated into modules and separated from the main software functionality enable more efficient maintainability of software as less software needs to be maintained. This combined with B2 allows the entire software systems (with aspects) to be more efficiently maintained.

In the area of database systems there is limited experience about impacts of AOSD to database development. An overview over few existing approaches that use separation of concerns in databases, developed by the aspect-oriented database (AOD) initiative, is given in [19]. However, a quantitative study on how aspects impact the development of database systems as compared to traditional approaches to development of database software has not been studied; in AOD it is assumed that aspects are beneficial for databases based on the studies performed of the general-purpose software.

However, it is essential to identify benefits and potential drawbacks of a novel technique, such as AOSD, before it can efficiently be applied in the domain of database systems. For example, a study [10] shows that crosscutting concerns such as concurrency and failures, which have been successfully modularized into aspects in general-purpose software, cannot easily be aspectualized, and are not as beneficial in a real-world distributed system. Therefore, a study that addresses the impact of AOSD to database system development, i.e., study of database system software to show whether benefits B1-B7 can be identified and confirmed, is needed.

3.2. Aspects in database systems

Aspects in a database system can be classified in two levels [19, 21]:

- database management system (DBMS) level aspects, which provide features affecting the software architecture of the database system and allowing the tailoring of a database system architecture and features towards a specific system with which the database is going to work, and
- database level aspects, which relate to the data maintained by the database and their relationship, i.e., the database schema.

We have identified a number of aspects in database systems on DBMS level by considering a feature as a crosscutting concern if it is spread over multiple subsystems, functions, and/or code modules of the database system, but performs the same function, or a part of the function, in the system. Based on these criteria, we have identified the following aspects that provide tailoring on the DBMS level:

- synchronization, e.g., in a DBMS there exist many data areas spread over the entire DBMS that should be protected by semaphores, which can be encapsulated into aspects and automatically woven into the DBMS;
- failure detection, e.g., keeping data consistent in the database requires employing failure detection, which is typically spread over the entire DBMS in order to capture failures that can occur, and therefore can be considered as an aspect;
- logging and recovery, e.g., in order to recover from a failure, logging is performed whenever database changes occur, and this often require logging routines to be spread out the entire software, and, thus, easily classified as an aspect;
- error handling, e.g., different errors that can occur in the execution of the database software could be detected by monitoring the execution of a program by an error handling aspect;
- transaction model, e.g., in real-time and embedded systems transactions are associated with different temporal properties such as deadlines and/or periods and these can be woven by means of aspects into a transaction model (hence, tailoring it to suit the needs of the underlying application);
- database policies such as scheduling policy and concurrency control policy, e.g., real-time and embedded systems require different real-time scheduling policies that can be plugged-in by means of aspects; and

- security, e.g., different encryption algorithms could be suitable for different database applications and these could be encapsulated into aspects and woven into the database to tailor it for a specific application.

Additionally, databases can make use of so-called development-type aspects such as debugging, which can also be classified as a DBMS level aspect.

Aspects on the database level are identified in [19], where they are applied to development of database schema in the SADES database. In SADES the following features are considered database level aspects: (i) changes to links among entities, such as predecessor/successor links between object versions or class versions, inheritance links between classes, etc.; (ii) changes to version strategy for object and class versioning; (iii) extending the system with new meta classes; and (iv) data object persistence. Database level aspects are specific to a particular implementation of a database schema, and for each database system these could differ, i.e., different parts of database schema could be more applicable to represent as aspects.

3.3. Goals and methodology

The goal of the work presented in this paper is to identify and study benefits and drawbacks of using aspect languages for development of DBMS software. For that purpose we have chosen to perform the case study on one well-known commercial and open source embedded database, namely Berkeley Database (DB) [1]. The reason for choosing an embedded database lies in the fact that designing a database customized for a particular application is essential for an embedded database system and therefore bears even greater importance than for the traditional database systems. This is true since the main objectives for an embedded database are low memory usage, portability to different operating system platforms, efficient resource management, e.g., minimization of the CPU usage, and ability to run for long periods of time without administration [14].

Given the broad range of aspects that can be identified and used in a database system, for the purpose of our study, we have chosen a subset of DBMS level aspects most likely to be found in every database system: failure detection, synchronization and error handling. Thereby we are able to illustrate benefits and drawbacks of AOSD on an embedded database and further generalize results to other database domains.

Our aim is to investigate whether the benefits B1-B7 hold when aspect languages are used for database software, and we do that by showing the impact of re-engineering Berkeley DB to support aspects. We investigate B1-B5 based on our implementation of aspects in the database, while B6-B7 are investigated based both on the implementation of aspects and on the number of lines of code that are decreased when aspects are used (as compared to the original implementation of the database). For providing quantitative support of our findings with respect to B6 and B7 we use the measurements in terms of number of lines (NoL) of the source code in DBMS software, since reducing the NoL of the code enables more efficient testing and maintaining of the database software as less code should be tested and maintained.

4. Berkeley database: a case study

Here we first present a brief overview of the Berkeley DB in section 4.1, while in the remaining sections we focus on a detailed description and analysis of the following crosscutting database features: failure detection (section 4.2), synchronization (section 4.3), and error handling (section 4.4).

4.1. Berkeley DB: an overview

Berkeley DB is an embedded database system, implemented as a classical C-library toolkit that can be linked directly into an application. The database provides application programming interfaces (APIs) for applications written in other programming languages, such as C++ and Java. Berkeley DB consists of the following subsystems [1]: access methods, memory pool, transaction, and locking.

The *access methods subsystem* provides support for creating and accessing database files. The files are accessed using key/data pairs to identify desired elements within the database. The *memory pool subsystem* is a general-purpose memory buffer pool that allows multiple processes and threads within the process, to share access to the database. The *transaction subsystem* implements the Berkeley DB transaction model. It enforces strict ACID transaction semantics. The *locking subsystem* uses two-phase locking to provide interprocess (multiple threads within process) and intraprocess (multiple processes) concurrency control. It uses page-level locking by default. The *logging subsystem* ensures that committed changes to the database survive failures in an application, system, or hardware. It uses write-ahead logging, thus, logging the information about changes in the database before the change actually occurs.

Berkeley DB provides separate interfaces for each subsystem. This implies that each subsystem is implemented as an independent module, and can even be used outside the context of Berkeley DB. Hence, an application developer, when using the database, can specify which subsystems his/her Berkeley DB configuration should contain based on the database services required by a particular application. For example, if the application needs fast, single-user, and non-transactional B-tree data storage, the locking and transaction subsystems do not have to be included in the

database configuration, i.e., they can be disabled, thus, reducing the overhead of locking and logging. In contrast, if an application needs to support multiple concurrent users, but does not require transactions, the locking subsystem can be included in the configuration without the transaction subsystem. Moreover, applications that need concurrent, transaction-protected database access can configure the database such that all subsystems are enabled, i.e., all subsystems exist in the configuration. Hence, Berkeley DB is an embedded reconfigurable database that makes the case study even more interesting as it allows investigation of impact of applying AOSD on a reconfigurable database.

All database management functions provided by Berkeley DB can be accessed via operations defined in the interfaces of the subsystems. When using the database, an application should first create a structure, referred to as an object handle, and then call the functions of that structure (representing the methods of that handle). One of the most important data structures in Berkeley DB is the database environment, which represents an encapsulation of all database states and holds the information about the current status and the configuration of the database. The database environment is accessed using its handle (`db_env`), which is created by passing parameters from the application to the function `db_env_create`.

4.2. Failure detection

Failure detection in Berkeley DB consists of recovery detection and run-time configuration detection.

Recovery detection routines in Berkeley DB detect, in every subsystem interface and its operations, if there is a need for performing the recovery. Since applications can tailor the database to suit their requirements, Berkeley DB itself cannot determine whether recovery is required as it is not aware of the current database configuration. The application, thus, should determine when recovery is required based on the results from the recovery detection routines run by the Berkeley DB.

A recovery detection routine is implemented through the `panic_check` function. To detect failures, e.g., when log files are physically destroyed or when the underlying file system is corrupted, `panic_check` probes the state of the existing database environment. If the state of the environment indicates that the failure happened, `panic_check` returns an error value (`db_runrecovery`) to the caller, i.e., an application or an internal database function that propagates the recovery information to the application. Typically, `panic_check` is invoked at the beginning of the functions implementing database functionality within each of the subsystem, and immediately after variable initialization.

Analysis of the Berkeley DB source code resulted in the observation that 55 different functions call recovery detection routine `panic_check`. The calls are spread over all five database subsystems, hence, making the recovery detection a crosscutting concern in the Berkeley DB.

Run-time configuration detection detects whether a call to the database from the application is made using a method or a function within the current database configuration. Due to the configurability feature of the Berkeley DB, it is important to detect if an application makes a call within the right configuration. If the function or method call is made to one of the subsystems that are not in the current database configuration, a warning message is displayed and an error value (`einval`) returned to the application. The run-time configuration checks are performed by invoking the configuration detection routine `env_requires_config` within the functional modules of Berkeley DB subsystems.

Analysis of the source code exposed 26 different function calls to the `env_requires_config` routine spread over four database subsystems, namely locking subsystem, logging subsystem, memory pool subsystem, and transaction subsystem. Hence, it can be observed that run-time configuration detection is a crosscutting concern as it affects a number of functions in different subsystems.

Failure detection in, e.g., the memory pool subsystem, is performed in the following manner. When an application invokes `memp_register` method in the memory pool subsystem, it first invokes `panic_check` to check whether the environment is not damaged. After invoking the `panic_check`, if everything is correct and an error is not detected, `env_requires_config` is called to check whether the application has set up a memory pool subsystem in its environment. If the memory pool subsystem has not been set up in the current configuration, a warning message is printed and an error value (`einval`) is returned to the application.

4.3. Synchronization

Berkeley DB synchronizes access to shared memory data structures, such as the lock table, in-memory buffer pool, in-memory log buffer, etc. Each independent subsystem uses mutexes[2] to protect its shared data structures, denoted regions. There are four types of operations used for implementing synchronization in Berkeley DB:

- `mutex_lock/unlock`, which is a lock/unlock operation on the memory buffer,
- `mutex_thread_lock/unlock`, which is a lock/unlock operation on a thread,

2 A mutex in this context denotes a mutual exclusion semaphore.

- `r_lock/unlock`, which is a lock/unlock operation on a region, and
- `lockregion/unlockregion`, which is a lock/unlock operation on regions that are related to locking operations.

`mutex_lock/unlock` are the basic synchronization operations, while all other operations are realized by calling `mutex_lock/unlock` to acquire/release the desired locks. Synchronization operations do not have a fixed position in the functional code of each subsystem, rather they are subsystem-dependent, i.e., depend on the functional behavior of a particular subsystem. For example, in `memp_pgread` function within the memory pool subsystem, when an application wants to read a page in memory, it invokes `mutex_lock` to block other threads from updating the page that the application is operating on. After the reading is done, `mutex_unlock` is invoked to release the buffer.

The analysis of the Berkeley DB source code indicated 116 different functions within all the five subsystems that use the synchronization operations in various places within their code. Given the number of functions crosscut by the synchronization operations, the synchronization can be viewed as a crosscutting concern of the Berkeley DB.

4.4. Error handling

Error handling in Berkeley DB is implemented using if-statements in the functional code of the database. The placement of if-statements throughout the code is done ad hoc as it only depends on the desired functionality of the code and undesired conditions that may occur. When the error is detected (within the if-statement), an output message is displayed[3], and an error value is returned to the caller, e.g., application or other database functions. The return values for an error can be grouped into the following three categories: (i) `ret=0`, indicating the successful completion of an operation; (ii) `ret>0`, indicating a system error, e.g., unable to allocate memory; and (iii) `ret<0`, indication a condition that is not a system failure, but is not an unqualified success either, e.g., a routine to retrieve a key/data pair from the database may return `db_notfound` when the key/data pair does not appear in the database as opposed to the value of 0, which would be returned if the key/data pair were found in the database.

Analysis of the source code exposed 194 different functions calls to the error handling routine spread out over all five major subsystems of Berkeley DB. Hence, we view error handling as a crosscutting concern in Berkeley DB.

3 The output message depends on the content of the environment configuration.

5. Modularizing crosscutting concerns

This section shows how aspects encapsulate the crosscutting concerns we identified and discussed in the previous section. We show the aspect-oriented implementation of failure detection, synchronization and error handling, and provide analysis of the impact of the aspectualization of the Berkeley DB in terms of the expected benefits B1-B7.

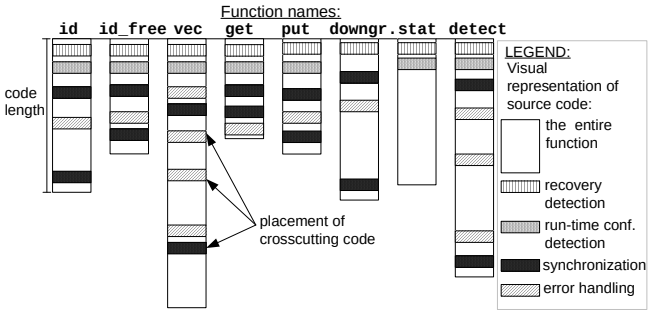

Figure 3. Illustration of crosscutting

Figure 3 represents the visualization of crosscutting in the locking subsystem in Berkeley DB. We observed similar crosscutting effects in each subsystem of Berkeley DB and, therefore, found that it is enough to show the effects of crosscutting on one of the subsystems in order to illustrate code entanglement. As can be seen from figure 3, the code of each function within the locking subsystem is crosscut with several crosscutting concerns. Furthermore, each function has its own set of crosscutting features, e.g., the function `downgrade` is crosscut with code for recovery detection, synchronization, and error handling, while within `stat` function there is the code for recovery detection and run-time configuration detection. Hence, even with only a subset of possible crosscutting concerns of the database system, their effects to the database code entanglement are easily noticeable.

5.1. Failure detection aspect

As mentioned, failure detection consists of recovery detection and run-time configuration detection. Hence, the aspect-oriented implementation of the failure detection can be done in two variants. In the first variant, the code for recovery detection and run-time configuration detection are implemented as separate aspects, i.e., recovery detection aspect and run-time configuration detection aspect. In the second variant, recovery detection and run-time configuration detection are implemented as one aspect, called the failure detection aspect.

```
1: aspect recovery_detection{
2:  pointcut RD(DBenv dbenv) =
3:        (call("% DbEnv::%_stat(...)") ||
                                    ...
19:                (target(DbEnv) || target(Dbc) ||
                                    ...
22:  advice RD(dbenv) : void before(){
23:    if(!F_ISSET((dbenv), DB_ENV_NOPANIC) &&
24:      (dbenv)->reginfo != NULL && ((REGENV *)
25:      ((REGINFO *)(dbenv)->reginfo)->primary)->envpanic != 0)
26:     return(DB_RUNRECOVERY);
27:   }
28: }
```

Figure 4. The recovery detection aspect

First variant: recovery detection aspect and run-time configuration aspect Figure 4 shows the *recovery detection* implemented as an aspect. As can be seen from the figure, the recovery detection aspect consists of one pointcut and one advice. The pointcut syntax corresponds the syntax given in figure 2, with the subset of pointcuts, `call` and `target`. In the `call` pointcut all functions that are using the recovery detection routine are listed. An example of a pointcut is shown in figure 4 (lines 2-3) where the wildcard % indicates that the pointcut named `RD` refers to all calls to the functions with signatures containing `DbEnv`. The body of the advice in lines 23-26 is the same as the original implementation of the `panic_check`. Note that having the body of the advice implemented and formatted as in the original implementation of `panic_check` allows us to make a fair comparison of the number of lines of code reduced by aspect weaving. The advice is executed before the call is made to any of the functions with the specified signature in the pointcut `RD`. The implementation of the recovery detection aspect does not change the architecture or the functionality of the Berkeley DB. This is true since the same recovery detection code that was encapsulated by the routine `panic_check` is now executed within the advice, before the call to any of the functions that need to perform the recovery detection is made. This flow of execution also corresponds to the original implementation since `panic_check` was always executed first in the function in which it exists.

Run-time configuration detection aspect is illustrated in figure 5. This aspect has four named pointcuts: `lock_CD`, `log_CD`, `mem_CD`, and `txn_CD`, each consisting of a number of `execution` pointcuts. The `execution` pointcuts describe the execution of the function with the signature given in the pointcut. Four named pointcuts are required to identify the executions of functions in each of the four different subsystems: locking, logging, memory, and transaction subsystem. The aspect also contains four advices each corresponding to one of the named pointcuts. The shaded

```
1: aspect configuration_detection{
2:    pointcut lock_CD(DB_ENV dbenv) =
3:      execution("% DbEnv::lock_vec(...)") || ...
4:      execution("% DbEnv::lock_get(...)") || ...
5:      execution("% DbEnv::lock_detect(...)") &&
6:      target(dbenv);
7:    pointcut log_CD(DB_ENV dbenv) =
            (execution("__log_%register%(dbenv, ...)") ||
                                    ...
12:   pointcut mem_CD(DB_ENV dbenv) =
            (execution("% DbEnv::memp_register(...)") ||
                                    ...
16:   pointcut txn_CD(DB_ENV dbenv) =
          (execution("% DbEnv::txn_checkpoint(...)") ||
                                    ...
19:   advice lock_CD(dbenv) : void before(DB_ENV dbenv){
20:     if (dbenv->lk_handle == NULL){
21:       cout << JoinPoint::signature() << " interface requires"
22:       cout << " an environment configured for the locking subsystem";
23:       return (EINVAL); }
24:   }
25:   advice log_CD(dbenv) : void before(DB_ENV dbenv){...}
                                    ...
31:  advice mem_CD(dbenv) : void before(DB_ENV dbenv){...}
                                    ...
37:  advice txn_CD(dbenv) : void before(DB_ENV dbenv){...}
                                    ...
43: }
```

Figure 5. The run-time configuration detection aspect

parts in figure 5 give the pointcut/advice pair for the locking subsystem. The pointcut `lock_CD` refers to the executions of each of the functions within the locking system that require run-time configuration detection. The before advice `lock_CD` identifies whether a join point described by the pointcut `lock_CD`, in which the advice is currently executing, is part of the current database configuration (described by lines 20-23; `JoinPoint::Signature` in line 21 identifies the current join point).

Given the original implementation of recovery detection and run-time configuration detection, and their aspectual implementation, we conclude the following about benefits B1-B7. Considering that recovery detection and run-time configuration detection are both implemented only within one routine (`panic_check` and `env_requires_config`) the positive effects of aspectualization to these routines with respect to independent development (B1) and localized changes (B2) can be observed only in the introduction of pointcuts. The pointcuts allow localized changes in the sense that all the impact of using this functionality to the overall database is localized. When using the aspectual solution, extensibility (B3) and tailorability (B5) of the database are further improved. Namely, new modules of the database can be developed and added to the database independently of the failure detection, and this feature can easily be added to the parts of the new modules by defining new pointcuts in the aspects. Comprehensibility (B4) with respect to the implementation of the recovery and run-time configuration detection is not significantly improved (due to the fact that these are implemented with only one routine). How-

Feature	Invocations[NoL] original	Invocations[NoL] aspect	Implementation[NoL] original	Implementation[NoL] aspect	Total[NoL] original	Total[NoL] aspect	**Change** [%]
Failure detection	114	0	41	66	155	66	-57
Recovery detection	64	0	6	40	70	40	-43
Configuration detec.	50	0	35	43	85	43	-49
Synchronization	436	0	425	503	861	503	-42
Error handling	650	0	142	1201	792	1201	+65

Table 1. Comparison of the number of lines (NoL) used in the original and aspectual implementation of the Berkeley DB.

ever, the overall comprehensibility of the database code and the effects of these two features to the database are indeed improved as all the functionality of failure detection is localized in aspects and described in terms of advices and pointcuts (which are straightforward to understand). Furthermore, as shown in the table 1, the use of aspects significantly reduces the amount of code that deals with recovery detection. In the original implementation of the Berkeley DB, the number of invocations of `panic_check` for performing recovery detection is 64, while the number of lines implementing the `panic_check` routine itself is six. After encapsulating recovery detection into an aspect, `panic_check` is not invoked in the code of the database, i.e., the number of invocations is zero, and the number of lines of code used to implement the recovery detection aspect is 40. Hence, this results in 43% reduction of the code that handles recovery detection, which improves testability (B6) and maintainability (B7) of the code. Similarly, we obtained 49% reduction of the code that handles run-time configuration detection, implying that benefits B6 and B7 hold.

Second variant: failure detection aspect that encapsulates both recovery detection and run-time configuration If failure detection is implemented using only one aspect that encapsulates recovery and run-time configuration detection, the code reduction increases to 57% as compared to the independently implemented aspects. The decrease is due to a large number of pointcuts shared by the recovery detection and run-time configuration detection. Therefore, testability and maintainability (B6 and B7) of the code are further improved (as compared to the first variant). Of course, encapsulating the failure detection into an aspect enables better comprehensibility of the failure detection in general (B4). Also, encapsulating the overall failure detection into an aspect, without disturbing the original architecture of the database, allows development of the failure detection code independently of the overall database software (B1). However, knowledge of the functions that require failure detection, i.e., pointcuts, is required. Changes to the overall failure detection code can be done in a localized manner, within the failure detection aspect, i.e., B2 holds. The implementation of failure detection as one aspect reduces the tailorability (B5) of database features, as recovery detection and run-time configuration detection are not independent and cannot be exchanged and modified independently. Hence, there is a tradeoff between B5 and B7, as improving tailorability could result in decreased maintainability of the software.

```
aspect synchronization{
  int lock(db_env dbenv, db_mutex mutexp){..}

  int unlock(db_env dbenv, db_mutex mutexp){..}

  pointcut mutex_lock
  pointcut mutex_unlock
  pointcut mutex_thread_lock
  pointcut mutex_thread_unlock
  ....
  advice mutex_lock(dbenv) :
    void before(DB_ENV dbenv){
      lock(dbenv, &((dbenv->lk_handle)->
      reginfo.primary)->rp->mutex);)}
}
```

Figure 6. The synchronization aspect

5.2. Synchronization aspect

We have encapsulated synchronization operations into the synchronization aspect. The synchronization aspect consists of the internal methods `lock/unlock` that implement the core synchronization operations `mutex_lock` and `mutex_unlock`, providing the same functionality as the original implementation of operations `mutex_lock/unlock`. The advices, corresponding to other synchronization operations, e.g., `r_lock/unlock`, `mutex_thread_lock/unlock`, call the aspect methods `lock/unlock` to perform required locking/unlocking (see figure 6).

Based on the implementation of the synchronization aspect we can conclude the following. We are able to develop the synchronization aspect (B1) independently of the database code, given the knowledge about join points in the database software where synchronization operations need

to be performed. Furthermore, the comprehensibility (B4) of the aspect, and the database in general, improves significantly as the extensive lock/unlock operations of different types are removed from the code. Changes to the synchronization routines are localized within the aspect (B2 holds). Moreover, due to the nature of synchronization operations we achieved significantly better localization than in the case of the failure detection aspect. The introduction of a new functionality into the database (B3) is also improved as the only change to the synchronization aspect needs to be done in the pointcut declaration. The database can now be tailored further (B5) by simply including or excluding the functions that require synchronization in the pointcut declaration. From table 1 we observe that the total number of invocations of synchronization operations in Berkeley DB is 436, while the number of lines that implement synchronization operations is 425. Hence, the total number of lines used for the synchronization operations in the Berkeley DB original code is 861. When implementing the synchronization as an aspect, 70% of the lines of code used to implement the aspects are the lines used for defining the pointcuts in the code. This results in total aspect code of 503 lines, and significant reduction of code of 42% as compared to the original, non-aspectual, implementation. Hence, the code of the overall database decreases making it easier to maintain and test database software (B6 and B7 hold).

5.3. Error handling aspect

The structure of error handling aspect follows the typical structure of an aspect. However, when error handling is encapsulated into aspects, the number of pointcuts is significant (total of 1072 lines). This is partly due to that error handling is done using if-statements, which are not directly supported in the pointcut syntax (see figure 2), and partly due to a variety of different conditions that are used to detect errors in the code.

Hence, the aspect solution of error handling resulted in a (surprising) 65% increase of the code as compared to the original database code, as can be seen from table 1. The cause of this increase is the tangled if-statement context-dependent error handling style that induced great number of pointcuts in the aspectual implementation of the error handling, increasing the total volume of the code. The significant increase of the amount of the code decreases maintainability and testability of the overall system (i.e., B6 and B7 do not hold). Although the implementation of error handling as an aspect increases comprehensibility (B4) of the overall database code, the comprehensibility of the error handling aspect has not been increased as the pointcut declarations showed to be overly complex and difficult to understand. Based on the error handling aspect and the intricate pointcut definition it is difficult to claim that the error handling could be developed independently (B1) if the architecture of the system is not modified to a large extent. However, changes to the error handling routine are localized in the case of aspects, i.e., benefit B2 holds, but the code is not easily extensible when it comes to the code of the aspect and the overall database due to the error handling technique employed in the original implementation of the Berkeley DB.

5.4. Lessons learned

The study we performed on the Berkeley DB provided a valuable insight how a database system could be programmed to enhance its tailorability, maintainability, testability, and comprehensibility. Aspect-oriented approach in designing and implementing databases improves the maintainability of the system and allows efficient changes in the database software as the crosscutting concerns in the system, e.g., failure detection and synchronization, can be maintained separately, localized in aspects, and then automatically woven into the overall system. By re-engineering the original Berkeley DB code, we showed that the implementation of failure detection and synchronization aspects, provides a way of reducing (up to 57%) the code needed for implementing these crosscutting concerns in the database system. We have identified that there is a trade-off between requirements for configurability and maintainability of the system when aspects are used, namely, recovery detection and run-time configuration detection implemented in separate aspects allow greater flexibility and tailorability of the database, but they also require maintenance of the two separate aspects. In contrast, implementing the two detection routines as one failure detection aspect reduces the code further and allows for easier maintenance as the designer should only maintain one aspect.

Encapsulating error detection into an aspect in Berkeley DB produced a significant increase of code needed for error detection (65% increase), and exposed the drawbacks in the way error handling is currently implemented in Berkeley DB. This result is different from [11] where it is shown that the code reduction by factor four can be made in the best-case scenario when using aspects for error handling. The system studied in [11] is a Java-based object-oriented framework for interactive business applications. The system detects and handles errors by throwing exception and using a catch-statement to handle exception. In contrast, Berkeley DB is a C library and uses if-statement to detect and handle errors. Hence, defining the pointcuts without changing the original architecture of the database system requires a significant amount of code to describe conditions under which errors may occur. However, if an error detection and handling model is implemented using AOSD from scratch the result could improve significantly. Hence, not only would the re-engineering the existing database using aspects to en-

- not supported
+ supported in a limited form
++ fully supported

Implementation	Supported characteristics						
	B1	B2	B3	B4	B5	B6	B7
original	-	-	+	-	+	+	-
re-engineered	+	++	+	++	++	++	++

B1- independent development of aspects
B2- localized changes in database software
B3- extensibility of database
B4 - comprehensibility of database functionality
B5- tailorability of database towards a particular system
B6 - improved testability
B7 - improved maintainability of database software

Figure 7. The overall effect of re-engineering Berkeley DB to support aspects

capsulate crosscutting concerns provide benefits for an exiting system; the beneficial impact would be even more noticeable if the database system is designed with aspects in mind. Given that the constructs provided by the aspect languages (pointcuts and advices) are powerful and yet simple enough to capture most of the crosscutting concerns, when developing the database software the programmers should only focus on the core functionality of the database system, while all other crosscutting issues, such as logging, recovery and synchronization can be implemented using aspects.

Figure 7 shows an overview of the types of benefits that could be observed generally for the database software, when comparing the original Berkeley DB implementation with the re-engineered one that supports aspects. As can be seen from figure 7 we could identify that most of the benefits B1-B7 are true for the re-engineered database, and that we obtained in overall significant improvements over the original implementation, e.g., localized changes in the database software, comprehensibility, and maintainability. Issues such as tailorability of the database were also improved with aspects. This is an interesting observation for this type of a configurable database as it implies that if the already configurable database could be improved further with respect to enabling tailorability, then tailorability in a monolithic database could be significantly improved by introducing aspects. Finally, the impact of aspectualizing the database does not reflect negatively on the functionality or performance of the database, i.e., database re-engineered with aspects exposes the same functionality as the original database.

6. Related work

In the area of database systems, the aspect-oriented databases initiative aims at bringing the notion of separation of concerns to databases [19]. A semi-autonomous database evolution system (SADES) [18], a product of the initiative, uses aspects to allow customization of the database system. The main focus of SADES is aspect support on the database level, e.g., aspects are used to denote changes to links among entities, such as predecessor/successor links between object versions or class versions, and inheritance links between classes. In contrast, in this paper we investigated the impact of using aspect languages for customization of the database management software, thus, using aspects on the level of database software.

Component-based database management systems (CDBMSs), which can be partially or completely assembled from a pre-defined set of components, allow tailoring of the database system towards a specific application. Different component-based databases enable different degrees of tailoring the database system for a particular application. Four different categories of CDBMSs have been identified [5]: (i) *extensible DBMSs* extend existing DBMS with non-standard functionality, e.g., Oracle8*i* [16], Informix Universal Server with its DataBlade technology [8], Sybase Adaptive Server [15], and DB2 Universal Database [3]; (ii) *database middleware* integrates existing data stores into a database system and provides users and applications with a uniform view of the entire system, e.g., OLE DB [13]; (iii) *DBMS service* provides database functionality in a standardized form unbundled into services, e.g., CORBAService [17]; (iv) *configurable DBMS* enables composition of a non-standard DBMS out of reusable components, e.g., KIDS [7]. Berkeley DB can also be viewed as a configurable CDBMS as it allows configuring the database depending on the application requirements. Common to all CDBMSs is that they are assembled out of components that encapsulate certain functionality. However, support for crosscutting concerns in CDBMSs is not provided. Database components are developed independently, and therefore each developed component has its crosscutting concerns implemented by the component developer independently of other components. This can lead to the code that is complex and difficult to maintain and develop.

7. Summary

Increasing complexity in development of database systems accompanied by the demand for enabling their tailorability requires the integration of aspect-oriented software development (AOSD) with database system development. However, it is essential to identify benefits and potential drawbacks of a novel technique, such as AOSD, before

it can efficiently be applied in the domain of database systems.

We have presented a case study, using the well-known embedded database system, Berkeley database, that identifies benefits and drawbacks of applying aspect-orientation and aspect programming languages to the design and implementation of database systems. The reason for choosing an embedded database lies in the fact that designing a database customized for a particular application is essential for an embedded database system and therefore has even greater importance than for the traditional database systems.

Our study shows that using the aspect-oriented approach when designing and implementing a database improves maintainability of the system and allows efficient changes in the database software as crosscutting concerns in the system can be maintained separately, localized in aspects, and then automatically woven into the overall system, e.g., failure detection and synchronization. The study also reveals that implementing error handling in the form of an aspect could result in an increase of the code size and thereby in degraded testability and maintainability of the system. Furthermore, we identified that there is a trade-off between requirements for tailorability and maintainability of the system when aspects are used.

Our on-going work focuses on implementation of a highly reconfigurable embedded real-time database, called COMET [21], which is being built using both aspect-oriented and component-based software engineering techniques.

References

[1] Berkeley DB, http://www.sleepycat.com. Sleepycat Software Inc.

[2] *The AspectJ Programming Guide*, September 2002. Available at: http://aspectj.org/doc/dist/progguide/index.html.

[3] M. J. Carey, L. M. Haas, J. Kleewein, and B. Reinwald. Data access interoperability in the IBM database family. *IEEE Quarterly Bulletin on Data Engineering; Special Issue on Interoperability*, 21(3):4–11, 1998.

[4] Y. Coady and G. Kiczales. Back to the future: A retroactive study of aspect evolution in operating system code. In *Proceedings of the Second International Conference on Aspect-Oriented Software Development*, pages 50–59. ACM Press, 2003.

[5] K. R. Dittrich and A. Geppert. *Component Database Systems*, chapter Component Database Systems: Introduction, Foundations, and Overview. Morgan Kaufmann Publishers, 2000.

[6] A. Gal, W. Schröder-Preikschat, and O. Spinczyk. On aspect-orientation in distributed real-time dependable systems. In *Proceedings of the Seventh IEEE International Workshop on Object-oriented Real-time Dependable Systems*, 2002.

[7] A. Geppert, S. Scherrer, and K. R. Dittrich. KIDS: Construction of database management systems based on reuse. Technical Report ifi-97.01, Department of Computer Science, University of Zurich, September 1997.

[8] Developing DataBlade modules for Informix-Universal Server. Informix Corporation, 22 March 2001. Available at http://www.informix.com/datablades/.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the ECOOP*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, 1997.

[10] J. Kienzle and R. Guerraoui. AOP: Does it make sense? The case of concurrency and failures. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002)*, volume 2374 of *Lecture Notes in Computer Science*, pages 37–61, Malaga, Spain, June 2002. Springer-Verlag.

[11] M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. Technical Report CSL-99-1, Utah Univeristy, 1999.

[12] D. Mahrenholz, O. Spinczyk, A. Gal, and W. Schröder-Preikschat. An aspect-orientied implementation of interrupt synchronization in the PURE operating system family. In *Proceedings of the 5th ECOOP Workshop on Object Orientation and Operating Systems*, Malaga, Spain, June 2002.

[13] Universal data access through OLE DB. OLE DB Technical Materials. OLE DB White Papers, 12 April 2001. Available at http://www.microsoft.com/data/techmat.htm.

[14] M. A. Olson. Selecting and implementing an embedded database system. *IEEE Computers*, 33(9):27–34, Sept. 2000.

[15] S. Olson, R. Pledereder, P. Shaw, and D. Yach. The Sybase architecture for extensible data management. *Data Engineering Bulletin*, 21(3):12–24, 1998.

[16] All your data: The Oracle extensibility architecture. Oracle Technical White Paper. Oracle Corporation, February 1999.

[17] M. T. Özsu and B. Yao. *Component Database Systems*, chapter Building Component Database Systems Using CORBA. Data Management Systems. Morgan Kaufmann Publishers, 2000.

[18] A. Rashid. A hybrid approach to separation of concerns: the story of SADES. In *Proceedings of the third International REFLECTION Conference*, volume 2192 of *Lecture Notes in Computer Science*, pages 231–249, Kyoto, Japan, September 2001. Springer-Verlag.

[19] A. Rashid. *Aspect-Oriented Database Systems*. Springer, 2004.

[20] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: an aspect-oriented extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Sydney, Australia, February 2002. Australian Computer Society.

[21] A. Tešanović, D. Nyström, J. Hansson, and C. Norström. Aspects and components in real-time system development: Towards reconfigurable and reusable software. *Journal of Embedded Computing*, February 2004.