AOP: Does it Make Sense?
The Case of Concurrency and Failures

Jörg Kienzle[1] and Rachid Guerraoui[2]

[1] Software Engineering Laboratory
[2] Distributed Programming Laboratory
Swiss Federal Institute of Technology in Lausanne (EPFL), Switzerland

Abstract. Concurrency and failures are fundamental problems in distributed computing. One likes to think that the mechanisms needed to address these problems can be separated from the rest of the distributed application: in modern words, these mechanisms could be *aspectized*. Does this however make sense?

This paper relates an experience that conveys our initial and indeed biased intuition that the answer is in general *no*. Except for simple academic examples, it is hard and even potentially dangerous to separate concurrency control and failure management from the actual application.

We point out the very facts that (1) an *aspect-oriented* language can, pretty much like a macro language, be beneficial for code factorization (but should be reserved to experienced programmers), and (2) concurrency and failures are particularly hard to aspectize because they are usually part of the phenomenon that objects should simulate. They are in this sense different than other concerns, like for instance tracing, which might be easier to aspectize.

Keywords. Aspect-oriented programming, abstraction, objects, concurrency, failures, exceptions, transactions.

1 Introduction

The job of any engineer is to manage complexity in designing and implementing systems. This is in particular true for software engineering: most research in the field has to do with how to manage the complexity of programs by providing better structuring mechanisms and methodologies.

Object-oriented programming comes with the intuitive idea that a program supposed to solve a real-world problem should be decomposed into a set of self-contained abstractions, each simulating a specific phenomena of the real-world problem. The abstraction is self-contained in the sense that it encapsulates state and behavior. One can follow the object-oriented programming discipline in any language, but object-oriented languages provide support to help such a programming discipline through mechanisms like encapsulation, sub-typing, inheritance, etc. [1].

Aspect-oriented programming (AOP) is the modern terminology given now to a branch of techniques that aim at deconstructing objects into several aspects (or concerns) and promoting each aspect to the level of a first-class citizen. Again, one could adopt this programming discipline in any language, but AOP languages help support

this kind of separation through mechanisms like join points, weaving, etc. [2]. Typically, one might talk about functional and non-functional aspects. The very notion of *functional part* has never been precisely defined, but is usually used to denote what average programmers are supposed to master. In essence, the notion of *functionality* is relative. These days, mechanisms that deal with concurrency and failures are, for instance, considered as *non-functional* aspects of the application. It is tempting to separate these aspects from the other functionalities of the application. This is very legitimate and it does not take long to convince any sensible programmer that such a separation would be great. The requirements of distributed applications vary tremendously, and it is appealing that concurrency control and failure management concerns can be configured separately by some distribution specialist to fit the application's needs.

Can the dream however come true? In other words, does it indeed make sense to use AOP techniques to separate concurrency control and failure management concerns from the other parts of a distributed application? The motivation of our work is precisely to address this question.

Our conclusion is that, except for simple (academic) examples, the answer is *no*. To get a reasonable behavior for any non-trivial distributed application, concurrency control, together with failure management, should in principle be dealt with as a full part of the application semantics, i.e., should be mixed up with the actual functionalities. One can indeed use an AOP language to achieve *some* level of syntactical separation, but the programmer should be aware of its very *syntactic-only* nature.

In our experiment, we use `AspectJ` [2] as a representative of aspect-oriented programming languages and *transactions* [3] as a fundamental paradigm to handle concurrency and failures. We proceed in an incremental manner, where we try to achieve three goals: from the most ambitious to the less ambitious one.

- First, we figure out the extent to which one can *aspectize transaction semantics*. That is, we figure out the extent to which one can completely hide transactional semantics from the programmer and have these semantics implicitly associated to the program a posteriori and in an automatic manner. This actually means that the programmer does not have to care about transactions. We give a set of illustrated examples why this is clearly impossible.
- Second, we figure out the extent to which one can *aspectize transaction interfaces.* That is, we figure out the extent to which one can completely separate transactional interfaces (*begin*, *commit*, *abort*, etc.) from the main (functional) object methods, and have these encapsulated within code invoked through specific aspects. We show by example that in certain cases this separation might be artificial, and that it leads to rather confusing code.
- Third and finally, we figure out the extent to which one can *aspectize transaction mechanisms.* That is, we figure out the extent to which one can completely separate the mechanisms needed to ensure the ACID [3] properties of transactions (i.e., concurrency control and failure management) from the main (functional) program (objects) and have these encapsulated within code invoked through specific aspects. We show that, syntactically speaking, an AOP language like `AspectJ` provides indeed a nice way of separating these mechanisms from the functional part of

the code. Just like with macros however [4], this separation should be handled with care, especially whenever the actual functionality does change. In short, the programmer must be aware that the physical separation does not imply a semantic decoupling.

It is important to notice that from our experience, especially in a non-rigorous area such as software engineering, we cannot draw any conclusion on the general applicability of AOP and AOP languages. The scope of our experience is indeed limited to (a) two concerns: concurrency and failures, (b) one paradigm to handle these concerns: transactions, and (c) a given subset of application scenarios that we have taken from our distributed computing background. The goal here is simply to provide some elements for a more general discussion of what dangers a misunderstanding of the capability of AOP might create, as well as what and when features of an AOP language might be useful *and* safe. The paper should be viewed as a warning to the new comers entering the AOP arena with very high expectations, rather than as an argumentation with AOP founders, who usually know the limitations.

The rest of the paper is organized as follows: Section 2 provides background information on AOP and transactions; Section 3 presents our experimental setting; Section 4 to Section 6 describe our attempts at achieving the three levels of aspectization mentioned above; Section 7 relates our experience to *Enterprise Java Beans* [5]; Section 8 discusses the limitation and possible generalization of our experiment, and Section 9 summarizes the results of this work.

2 Background

2.1 Aspect-Oriented Programming

Aspect-oriented programming (AOP) is the name given to a set of techniques based on the idea that software is better programmed by separately specifying the various *concerns (*or *aspects)*, properties, or areas of interest of a system, and describing their relationships [2]. Ultimately, the programmer relies on the underlying AOP environment to *weave* (or *compose*) the concerns together into a coherent program. Separating the expression of multiple concerns in programming systems promises increased readability, simpler structure, adaptability, customizability and better reuse.

One of the main elements of an AOP language is the *join point model*. It describes the “hooks” where enhancements may be added, and thus determines the structure of crosscutting concerns. AOP languages are supposed to provide means to identify join points, specify behavior at join points, define units that group together join point specifications and behavior enhancements, and provide means for attaching such units to a program.

2.2 Transactions

Transactions [3] have been used extensively to cope with concurrency and failures. A transaction groups an arbitrary number of simple actions together, making the whole appear indivisible with respect to other concurrent transactions. Using transactions, data updates that involve multiple objects can be executed without worrying about

concurrency and failures. Transactions have the so-called ACID properties: *Atomicity*, *Consistency*, *Isolation* and *Durability* [3]. If something happens during the execution of a transaction that prevents the operation from continuing, the transaction is aborted, which will undo all state changes made on behalf of the transaction. A transaction can also be aborted voluntarily by the application programmer. The ability of transactions to hide the effects of concurrency, and at the same time act as firewalls for failures, makes them appropriate building blocks for structuring reliable distributed applications in general.

Multiple transactions may execute concurrently, but classic transaction models only allow one thread to execute inside a given transaction. Such models therefore support competitive concurrency only, since transactions, and hence the threads running within them, are isolated from each other. There is no way for threads to perform cooperative work inside the same transaction. More sophisticated transaction models, i.e., the *open multithreaded transaction model* [6, 7], allow multithreading inside a transaction. Threads participating in the same transaction can cooperate by accessing the same objects.

2.3 Transaction Interfaces

Typically, transactional systems offer a procedural interface to transactions including three operations:

- `void beginTransaction()`, which starts a new transaction or a nested transaction within an already ongoing one,
- `void commitTransaction() throws TransactionAbortedException`, which attempts to commit the current transaction,
- `void abortTransaction()`, which forces the transaction to rollback.

Multithreaded transaction models, e.g. open multithreaded transactions, provide additional operations to allow threads to join an ongoing transaction:

- `void joinTransaction(Transaction t)`, which allows the calling thread to join the transaction `t`,
- `void beginOrJoinTransaction(String name)`, which creates a new transaction with the name `name`, or, if a transaction with this name already exists, joins the calling thread by associating it with the same transaction context.

3 Experimental Setting

We briefly describe below the basic tools of our experimental setting: `AspectJ` and our OPTIMA transactional framework. We mainly overview here the elements that are used in our context.

3.1 AspectJ

We based our experiment on `AspectJ` [8], an aspect-oriented programming environment for the Java language.

In `AspectJ`, the join points are certain well-defined points in the execution flow of a Java program. These include method and constructor calls or executions, field

accesses, object and class initialization, and others. *Pointcut designators* allow a programmer to pick out a certain set of join points, which can further be composed with boolean operations to build up other pointcuts. It is also possible to use wild cards when specifying, for instance, a method signature.

The following code defines a pointcut named `CallToAccount` that designates any call to a public method of the `Account` class:

```
pointcut CallToAccount () : call (public * Account.*(..));
```

To define the behavior at a join point, `AspectJ` uses the notion of *advice*. An advice contains code fragments that execute *before*, *after* or *around* a given pointcut. Finally, *aspects* are provided that, very much like a class, group together methods, fields, constructors, initializers, but also named pointcuts and advice. These units are intended to be used for implementing a crosscutting concern.

Since aspects potentially crosscut an entire application, the integration of aspects with the programming environment is of great importance. Programmers should get visual feedback of the effects of a given aspect on other parts of the program. The developers of `AspectJ` are aware of this, and hence provide extensions that integrate `AspectJ` with popular programming environments, such as Borland's JBuilder, Sun's Forte and GNU Emacs.

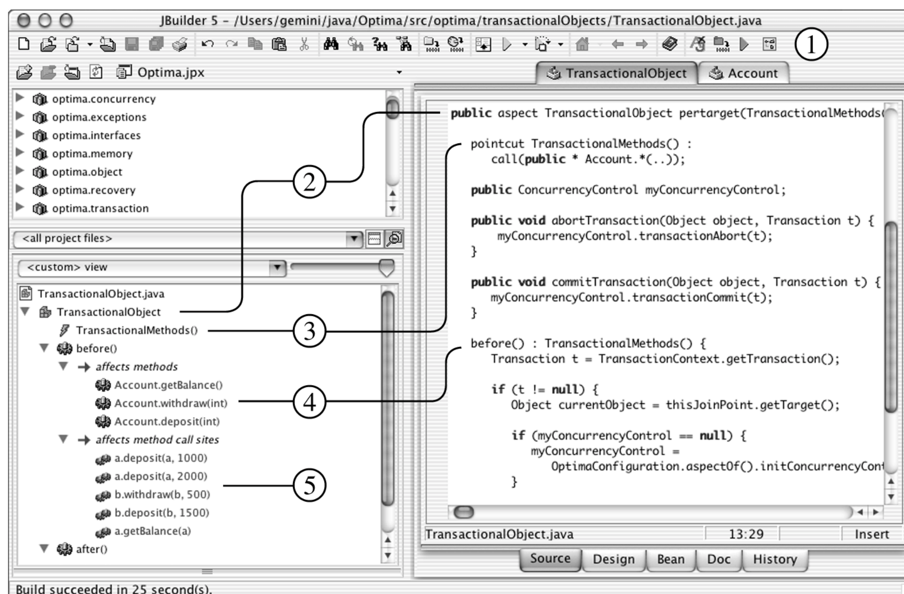

Fig. 1: AspectJ Integration with JBuilder 6 under Mac OS X

Figure 1 illustrates the integration of `AspectJ` with Borland JBuilder 6. The `AspectJ` plug-in adds buttons for compiling, running, and setting `AspectJ` preferences to JBuilder's toolbar ①. When the `AspectJ` environment is activated, the structure view of JBuilder is replaced with `AspectJ`'s structure view. It contains all elements of

JBuilder's structure view, but additionally allows a programmer to visualize aspect-specific constructs, e.g. pointcuts and advice.

In Figure 1, the structure of the `TransactionalObject` aspect is shown in the structure view ②: the definition of the `TransactionalMethods` pointcut ③, and a *before* advice. The next tab presents a list of all methods statically affected by the *before* advice, namely all public methods of the class `Account` ④. The following tab shows what actual calls are affected. In our test application, several calls are executed on `Account` objects `a` and `b` ⑤. Clicking on one of the method calls in the structure view makes the editor open the file that declares the call and jump to the corresponding line.

3.2 OPTIMA

Transactions require considerable run-time support. Our experiments make use of OPTIMA [6, 9], a highly configurable, object-oriented framework that provides support for open multithreaded transactions and guarantees the ACID properties for transactional objects.

Since transactions are nowadays used in different software domains, the requirements of applications using transactions vary tremendously. It is therefore important that the application programmer can configure a transaction support or middleware to fit the application's needs.

The OPTIMA framework has been designed along these lines. Hierarchies with classes implementing standard transactional behavior are provided, but a programmer is free to extent the hierarchies to tailor the framework to the application-specific needs. The framework supports, among other features, optimistic and pessimistic concurrency control, strict read / write or commutativity-based method invocation, different recovery strategies (Undo/Redo, NoUndo/Redo, Undo/NoRedo), different caching techniques, different logging techniques (physical logging and logical logging), different update strategies (in-place and deferred), and different storage devices.

For our experiment, a prototype of the OPTIMA framework has been implemented for Java. It offers a procedural interface that allows an application programmer to start, join, commit, and abort transactions (see Section 2.3). To guarantee the ACID properties, the framework must additionally be called before and after every method invocation on a transactional object.

4 Aspectizing Transactions

In this section, we relate our experience in trying to achieve the most ambitious of the three goals mentioned in the introduction, namely *aspectizing transaction semantics*. In other words, does it make sense to write a program without transactions, and then (somehow automatically) have the program run with transactions? Is it possible to take code written for and used in a non-transactional setting, and run it with transactions?

We discuss several reasons why this is clearly impossible. We point out the issue of local synchronization versus global synchronization, that is, transaction synchronization, then the issue of irreversible actions, and finally the impact of ensuring transactional semantics for all objects.

4.1 Cooperation vs. Competition

Concurrent systems can be classified into *cooperative* systems, where individual components collaborate, share results and work for a common goal, and *competitive* systems, where the individual components are not aware of each other and compete for shared resources [10, 11, 12].

Programming languages address collaboration and competition by providing means for communication and synchronization among threads. This can be done by using *shared objects*, also called *monitors* [13, 14]. Typically, two forms of synchronization are considered: *mutual exclusion* and *condition synchronization*.

- Mutual exclusion is a synchronization mechanism that ensures that while one thread is accessing the state of an object, no other thread can possibly gain access. In Java, this behavior is provided by classes that declare all their methods as being `synchronized`.
- Condition synchronization is necessary when a thread wishes to perform an operation that can only sensibly or safely be performed if another thread has itself taken some action or is in some defined state. For example, if a thread wants to pass some data to some other thread via a shared object, then the receiver thread must make sure that the sender has already stored the data in the shared object before trying to retrieve it. In this case, the receiver wants to synchronize with the sender, but the sender does not need to synchronize with the receiver. If the sender wants to know that the receiver has taken the data, then both threads must synchronize.

In Java, condition synchronization can be achieved by using the `wait()`, `notify()` and `notifyAll()` methods provided by all classes inheriting from `Object`. If a certain condition is not met, a thread executing one of the synchronized methods of a shared object can call `wait()` to suspend itself, thereby releasing the mutual exclusion lock. If some other thread modifies the condition, it should call `notify()`, which results in awakening the suspended thread.

In order to highlight the problems that arise when introducing transactions into previously non-transactional applications, let us consider the following example. Thread T1 wants to transfer money from bank account A to bank account B, whereas thread T2 wants to do the same from B to A. Without transactions, the program works just fine, provided that (1) no failure occurs during its execution, and (2) the `withdraw` and `deposit` methods are synchronized (ACID properties are of course not ensured without transactions).

Solvable Deadlock. To tolerate failures however, each thread must execute the two operations inside a transaction. If in this case the interleaving happens to be the one shown in Situation 1 of Figure 2, then a deadlock occurs. Fortunately, the deadlock can be broken by aborting one of the transactions and restarting it.

Unsolvable Deadlock. In situation 2, the withdraw operation is only performed if there is enough money on the account. In this case, an insufficient balance results in an unbreakable deadlock.

Situation 1 (solvable deadlock)		Situation 2 (unsolvable deadlock)	
T1:	T2:	T1:	T2:
`A.withdraw(Amount)`		`A.deposit(Amount)`	`B.deposit(Amount)`
	`B.withdraw(Amount)`	`while (B.getBalance()`	`while (A.getBalance()`
`B.deposit(Amount)`		`<= Amount) { }`	`<= Amount) { }`
	`A.deposit(Amount)`	`B.withdraw(Amount)`	`A.withdraw(Amount)`

Fig. 2: Possible Deadlock Situations caused by Transactions

Two different execution interleavings must be considered:

1. The first one is similar to the one presented before. T1 deposits the money on A, and T2 deposits the money on B. Both threads then try to execute `getBalance()`, but are blocked by the transaction support to prevent information smuggling (isolation property).
2. In the second scenario, T1 goes ahead, deposits the money on account A, queries the balance of account B, but then remains blocked in the while loop, because the balance of B is insufficient. The transaction support cannot allow T2 to call `deposit()` on B, otherwise the isolation property is violated. This deadlock is due to condition synchronization of T1 on T2.

Generally speaking, Java synchronized classes implement *linearizability* of *every single* operation [15], whereas transactions require *serializability* of *all* operations performed within a transaction [16].

The only way of circumventing this mismatch is to remove the isolation requirement between the two threads by executing them within the same transaction. Situation 2 in Figure 2 actually depicts a loose form of collaboration between the two threads. Either one can not perform its job without the help of the other.

To prevent unsolvable deadlocks, all threads that cooperate in some form must execute inside the same transaction. If transactions are to be introduced automatically and in a transparent manner, all such situations must be detected. For reasonably complex applications, this is clearly infeasible.

4.2 Irreversible Actions: I/O

Some method invocations are irreversible. Such a situation arises, for instance, in software controlled production cells. Invoking a method on an object that controls a forge might irreversibly shape some piece of metal. But even more conventional actions, such as displaying an alert message on the screen, cannot be undone. Admittedly, it is possible to remove the alert message on the screen, but perhaps some person has already read it and taken corresponding actions.

All I/O operations give rise to this kind of problems in transactional systems. Depending on the exact situation, different solutions are possible [17]. One solution is, for instance, to buffer irreversible method invocations, and only executing them on transaction commit[1]. Although it is possible to apply such techniques to objects with

1. This, of course, assumes that the method itself cannot fail.

irreversible methods, it can obviously not be done completely transparently. Irreversible actions are therefore a clear argument against the complete aspectization of transactions.

4.3 Uniformity

A more practical-oriented reason why a complete aspectization is impossible is that, as a result of such aspectization, *all* application objects must be made *transactional*, i.e., provide concurrency control, undo-functionality, durability, etc.

In principle, this can be achieved in `AspectJ` by declaring an aspect as shown in Figure 3. The pointcut `PublicMethodCall()` captures all public method invocations of all objects in the system, except those declared in the OPTIMA framework. This restriction is necessary to prevent that the OPTIMA objects providing support for transactions are made transactional themselves, leading to a clear nonsense recursion.

```
aspect TransactionalObjects pertarget(PublicMethodCall()) {
    pointcut PublicMethodCall() : call(public * *.*(..)) &&
                                  !within(ch.epfl.lglwww.optima..*);
    // introduce fields here that link the object to the transaction support
    // i.e. concurrency control, recovery manager, storage, etc.
    before() : AllPublicMethodCalls() {...}
    after() : AllPublicMethodCalls() {...}
}
```

Fig. 3: Capturing all Public Method Invocations on all Objects

At run-time, an instance of the `TransactionalObjects` aspect is associated with any object outside of OPTIMA that is the target of a public method invocation. The `before()` and `after()` advice make the necessary calls to the OPTIMA framework.

Although this approach looks reasonable, it might not be feasible in a particular setting. In our Java-based experiment, durability of the state of transactional objects is achieved using the Java serialization facility. Unfortunately, not all Java objects implement the `Serializable` interface, and hence making all objects transactional may not be possible.

An additional problem is memory usage. An instance of the aspect shown in Figure 3 is created for every accessed object in the system. Every object also needs, for instance, an associated concurrency control. This might end up adding a significant amount of memory use to applications composed of a large number of objects.

The issues raised in this subsection are fortunately not insurmountable. Some of them have been successfully addressed in [18].

5 Aspectizing Transaction Interfaces

In this section, we discuss the extent to which one can *aspectize transaction interfaces,* that is, completely separate transactional interfaces (*begin, commit, abort, etc.*) from the main (functional) object methods, and have these methods encapsulated within code invoked through specific aspects. We point out that this leads to very intricate programs because of the very nature of transaction terminations, their integration with exception handling mechanisms, and the difficulty in expressing thread collaboration.

The very need to consider worst-case situations for concurrency control and recovery typically also impacts performance.

5.1 Interactions with Transactions

Most of the time, an application programmer wants to commit his (her) transactions. In certain cases however, he (she) might want to abort a running transaction as illustrated by the example presented in Figure 4. The code shows a method `transfer()` that withdraws money from the bank account `source` and deposits it into the bank account `dest`. If there is not enough money on the source account, then the transaction is aborted.

```
void transfer(Account source, Account dest, int amount) {
   beginTransaction();
   try {
      source.withdraw(amount);
      dest.deposit(amount);
      commitTransaction();
   } catch (NotEnoughFundsException e) {
      abortTransaction();
   }
}
```

Fig. 4: Interacting with Transactions

If we separate the calls to the transaction support from the functional code, some other means must be found that allow the programmer to trigger a transaction abort.

One possibility is to use the exception mechanism provided in most modern programming languages. Transactions can be associated with exception handling contexts, typically methods [7, 19]. If the method ends normally, then the transaction is committed. If the method call terminates exceptionally, then the transaction is aborted.

The interaction problem might however also arise in the other direction. Even though a transaction is intended to commit, it may abort due to some failure in the system, i.e., the remote server that hosts the destination bank account is unreachable at commit time. In this case, the application programmer should be notified, for he (she) might want to take corresponding actions, i.e., execute an alternative transaction or retry the original one. Again, exceptions can be used to perform this notification, e.g., by means of a predefined exception `TransactionAbortedException`.

If this kind of integration is chosen, throwing exceptions has an additional meaning. The application programmer must be aware of the fact that an exception that crosses a transaction boundary results in a rollback. It seems clear to us that in this case it is preferable that the application programmer and the person that applies the transaction boundaries be the same person.

5.2 Making Methods Transactional Using AspectJ

When making the interface to transactions transparent, the calls to the transaction support must be completely hidden from the application programmer. They should execute automatically at certain points in the program.

Figure 5 shows an abstract aspect `TransactionalMethods` that wraps a transaction around a method invocation by making calls to the procedural interface of OPTIMA introduced in Section 2.3.

```
public abstract aspect TransactionalMethods {
    abstract public pointcut MethodToBeMadeTransactional();
    void around() : MethodToBeMadeTransactional() {
        ProceduralInterface.beginTransaction(); ①
        boolean aborted = false;
        try {
            proceed(); ②
        } catch (TransactionException e) {
            ProceduralInterface.abortTransaction(); ③
            aborted = true;
            throw e; ④
        } finally {
            if (!aborted) {
                ProceduralInterface.commitTransaction(); ⑤
            }
        }
    }
}
```

Fig. 5: Making Method Calls Transactional

The method call is encapsulated with the help of the `around()` advice. First, the `beginTransaction()` method is called to start a new (nested) transaction ①. The actual method call is placed inside a `try-catch` block and executed using the `proceed()` ② statement. If the original method call terminates with a `Transaction-Exception`, then the transaction is aborted ③, and the exception is thrown again ④. In any other case the transaction is committed ⑤. If the commit is not possible, the `commitTransaction()` method will throw the `TransactionAbortedException` exception.

In order to apply the `TransactionalMethods` aspect to a method, the programmer must extend the aspect and override the `MethodToBeMadeTransactional` pointcut. Figure 6 shows an aspect that makes all method invocations on the `Account` class transactional.

```
aspect MakeAccountMethodsTransactional extends TransactionalMethods {
    public pointcut MethodToBeMadeTransactional() :
        call (public * Account.*(..));
}
```

Fig. 6: Making Account Methods Transactional

As a result, all method invocations on the `Account` class may now throw the `TransactionAbortedException`.

5.3 Java-related Problems

Java has very strict rules for exception handling. Java exceptions are part of a method signature, i.e., a method or constructor must declare all exceptions it might throw during its execution. Forgetting to do so results in a compilation error. This rule applies to all exceptions apart from subclasses of `Error` or `RuntimeException`.[1]

In order to adhere to the Java exception rules, our aspect would have to modify the signature of the method it applies to. This is not possible in the current version of `AspectJ` (version 1.0.3), and we therefore had to declare the `TransactionAbortedException` as a subclass of `RuntimeException` in order to avoid compilation errors. Unfortunately this work-around is not completely satisfying. A Java program-

mer, relying on the fact that important application exceptions are checked, might forget to handle the `TransactionAbortedException`, which results in an incorrect program behavior.

5.4 Collaboration Among Threads

In the example presented in Section 5.2, every invocation of a method that has been specified as being transactional results in the creation of a new transaction. Unfortunately, this precludes any collaboration between threads as explained in Section 4.1.

In order to make collaboration possible, threads must be able to enter the same transaction. This can be achieved using a named transaction as shown in Figure 7.

```
public abstract aspect TransactionallyCollaboratingMethods {
   abstract public pointcut MethodToBeMadeTransactional();
   abstract public String initTransactionName();
   final String transactionName = initTransactionName();
   void around() : MethodToBeMadeTransactional() {
      ProceduralInterface.beginOrJoinTransaction(transactionName);
      // the rest of the code remains the same
   }
}
```

Fig. 7: Using Named Transactions

To apply this aspect to all `Account` objects, the programmer must define a concrete pointcut and provide a transaction name as follows:

```
aspect CollaboratingAccount extends TransactionallyCollaboratingMethods {
    public pointcut MethodToBeMadeTransactional() :
       call (public * Account.*(..));
    public String initTransactionName() {
       return "AccountTransaction";
    }
}
```

Fig. 8: Account Methods Collaborating Inside the Same Transaction

Now, a thread that executes `getBalance()` can proceed, even if some other thread has previously invoked `deposit()`, because they both participate in the same transaction named `"AccountTransaction"`.

5.5 Transactional Objects

In Section 4.3 we argued that it makes no sense to turn *all* application objects into transactional objects. The situation here is different, since we aim only at aspectizing transaction interfaces. The programmer specifies which methods are to be executed

---

1. Exceptions of the class `Error` indicate serious problems, e.g. `VirtualMachineError`. They should never occur and ordinary programs are not expected to recover from them. Subclasses of the class `RuntimeException`, e.g. `ArithmeticException` or `NullPointerException`, are thrown in case a language-defined check fails. These exceptions occur frequently, and hence the language designers decided that it would be cumbersome to force the programmer to declare them everywhere.

transactionally, and therefore also knows which objects are accessed from within a transaction. Only these objects must be made transactional.

In OPTIMA, every transactional object must be associated with a recovery manager and a concurrency control. In order to guarantee the ACID properties, each time a method is invoked on a transactional object the following actions must be taken:

1. **Concurrency Control Prologue** — Call the `preOperation()` method of the concurrency control associated with the object. This allows, for instance, a lock-based concurrency control to suspend the calling thread in case the method to be called conflicts with other method calls made from other transactions.
2. **Recovery Prologue** — Call the recovery manager's `preOperation()` method. This allows the recovery manager to collect information for undoing the method call in case the transaction aborts later on.
3. **Method Execution** — Execute the actual method call.
4. **Recovery Epilogue** — Invoke the recovery manager's `postOperation()` method.
5. **Concurrency Control Epilogue** — Call the `postOperation()` method of the concurrency control associated with the object.

Using `AspectJ`, these actions can be encapsulated inside an aspect as shown in Figure 9. The `TransactionalObject` defines a pointcut `TransactionalMethod`, which in our example specifies that all calls to public methods of the `Account` class are to be intercepted ①, thus making all `Account` objects transactional.

```
aspect TransactionalObject pertarget(TransactionalMethod()) {
   pointcut TransactionalMethod() : call(public * Account.*(..)); ①
   private final RecoveryManager myRecoveryManager = ...;
   private final ConcurrencyControl myConcurrencyControl = new ...; ②
   // other per-object info, e.g. recovery info
   public void abortTransaction(Object object, Transaction t) {
      myConcurrencyControl.transactionAbort(t);
   }
   public void commitTransaction(Object object, Transaction t) {
      myConcurrencyControl.transactionCommit(t);
   }
   before() : TransactionalObjectMethodCall() { ③
      Transaction t = TransactionContext.getTransaction(); ④
      if (t != null) {
         Object currentObject = thisJoinPoint.getTarget(); ⑤
         myConcurrencyControl.preOperation(t);
         myRecoveryManager.preOperation(currentObject, t); ⑥
      }
   }
   after() : TransactionalObjectMethodCall() {
      Transaction t = TransactionContext.getTransaction();
      if (t != null) {
         Object currentObject = thisJoinPoint.getTarget();
         myRecoveryManager.postOperation(currentObject, t);
         myConcurrencyControl.postOperation(t);
      }
   }
}
```

Fig. 9: Making Objects Transactional

The aspect itself is specified to be instantiated `pertarget(TransactionalMethod)`, meaning that an instance of the aspect is created for each account object that receives a public method call. Therefore, a separate copy of the private fields `myRecoveryManager` and `myConcurrencyControl` exist for each object. The fields are initialized when the aspect is instantiated, i.e. before a public method is invoked on the `Account` object for the first time ②.

The `before()` and `after()` advice encapsulate the actual method call. Any invocation of a public method on the `Account` class is intercepted, and the `before` advice is executed ③. First, the current transaction context is obtained from the transaction support ④. A reference to the transactional object itself is obtained by calling the `getTarget()` method of the `thisJoinPoint` object, which is an object offered by the `AspectJ` environment that provides information on the context of the advice's current join point ⑤. Next, the concurrency control prologue, and finally the recovery prologue are executed ⑥. The `after()` advice handles the epilogues in a similar way.

Although this aspect can be used to make any Java class transactional[1], it is nevertheless not a viable solution for an application that heavily relies on transactions. The transaction support does not get any knowledge on the frequency of use of the object, the size of its state and the semantics of its methods. As a result, worst-case assumptions must be made, which yields in poor performance. Exploiting such knowledge makes it possible to increase concurrency and decrease disk access, and therefore considerably improves performance.

6 Aspectizing Transaction Mechanisms

In this section, we discuss the extent to which one can *aspectize transaction mechanisms,* that is, separate the mechanisms needed to ensure the ACID properties of transactions (i.e., concurrency control and failure management) from the main (functional) program (objects) and have these encapsulated within specific aspects. We present how `AspectJ` aspects have been used to provide application-wide, per-class, and per-method customization of transaction mechanisms. We show that, although possible and elegant, this separation should be handled with care, especially whenever the actual functional part does change. In short, the programmer must be aware that the physical separation does not imply a semantic decoupling.

6.1 Configuring Application-Wide Transaction Preferences

Transactional systems guarantee atomicity and durability even in the presence of failures, i.e., crashes. Different techniques for recovering from a crash failure exist with different performance trade-offs.

To achieve durability, the state of transactional objects is stored on so-called stable storage [20]. To boost performance, the state of frequently used transactional objects is kept in a cache. On a system crash however, the content of the cache is lost, and there-

1. Note that, in order to support durability, the class must implement the `Serializable` or `Externalizable` interface.

fore, in general, the state of the stable storage can be inconsistent for the following reasons:

- The storage does not contain updates of committed transactions.
- The storage contains updates of uncommitted transactions.

When recovering from a system crash, the former problem can be solved by *redoing* the changes made by the corresponding transactions, the latter by *undoing* the changes made by the corresponding transactions [21].

The Undo/Redo recovery strategy can handle both situations, and therefore gives the most freedom to the cache manager. However, the time needed for performing recovery is considerable. Other recovery strategies, e.g., NoUndo/Redo or Undo/NoRedo, perform better during recovery [22], but constrain the cache manager and hence may potentially slow down performance during normal processing.

In our OPTIMA framework, one can select the appropriate recovery manager by instantiating the corresponding class from the class hierarchy presented in Figure 10.

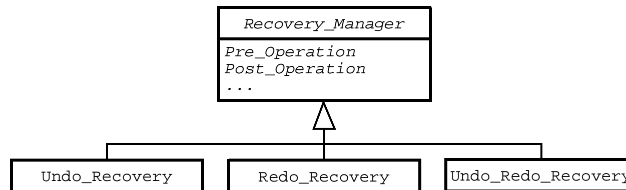

Fig. 10: The Recovery Manager Hierarchy

There must be a single recovery manager for the entire application, and it must be initialized during start-up. Using aspects, this initialization can be achieved by declaring an `OptimaConfiguration` aspect as shown in Figure 11.

```
public aspect OptimaConfiguration issingleton() {
   public RecoveryManager initRecoveryManager() {
      // instantiate your chosen recovery manager here
      // the parameters (omitted here) specify the desired log storage
      return new UndoNoRedoManager(...);
   }
   public final RecoveryManager recoveryManager = initRecoveryManager();
   // more code follows later
}
```

Fig. 11: Selecting a Recovery Manager

By using the modifier `issingleton()`, the aspect has exactly one instance that crosscuts the entire application. That instance is available at any time during execution of the program using the static method `OptimaConfiguration.aspectOf()`.

6.2 Configuring Object Transactional Properties

Objects that are accessed from within a transaction must be capable of handling cooperative and competitive concurrency. In order to address cooperative concurrency, methods that update the state of an object must execute in mutual exclusion.

Competitive concurrency control, which guarantees the isolation property of transactions, can be *pessimistic (conservative)* or *optimistic (aggressive)* [23], both having

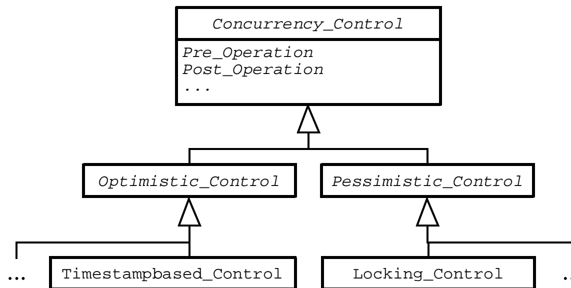

Fig. 12: The Concurrency Control Hierarchy

advantages and disadvantages. Figure 12 depicts an excerpt of the concurrency control class hierarchy of OPTIMA.

Every transactional object must have an associated concurrency control. In order to maximize concurrency, the kind of concurrency control must be configurable on a per-class (or even per-object) basis. However, in order to guarantee the serializability of transactions, the global serialization order must be the same for all concurrency controls used in a system [24].

To do this transparently, we have introduced an interface `CustomizedConcurrencyControl`, shown in Figure 13. An object that wants to specify its preferred concurrency control must implement this interface.

```
interface CustomizedConcurrencyControl {
   public ConcurrencyControl getConcurrencyControl();
}
```

Fig. 13: The `CustomizedConcurrencyControl` Interface

The following aspect does this transparently. It specifies a timestamp-based concurrency control for the `Account` class.

```
public aspect AccountConcurrencyControlAspect {
   declare parents: Account implements CustomizedConcurrencyControl;
   public ConcurrencyControl
      CustomizedConcurrencyControl.getConcurrencyControl() {
      return new TimestampbasedControl();
   }
}
```

Fig. 14: Selecting a Custom Concurrency Control for a Class

The default concurrency control can be set in the `OptimaConfiguration` aspect as shown in Figure 15.

The method `initConcurrencyControl` is called once for each object. In Figure 15, if the object does not implement the `CustomizedConcurrencyControl` interface, a `LockingControl` is instantiated.

6.3 Specifying Transactional Properties on a Per-Method Basis

To optimize concurrency even further, the transaction support needs specific information about the semantics of each method of an object.

```
public aspect OptimaConfiguration issingleton() {
   // code shown in Figure 11
   public ConcurrencyControl initConcurrencyControl(Object o) {
      if (o instanceof CustomizedConcurrencyControl) {
         // get the customized ConcurrencyControl
         return ((CustomizedConcurrencyControl) o).getConcurrencyControl();
      } else {
         // instantiate your default concurrency control here
         return new LockingControl();
      }
   }
}
```

Fig. 15: Selecting a Default Concurrency Control for All Classes

A sophisticated concurrency control can, for instance, allow method invocations on the same object made from different transactions to execute concurrently, if it knows that no information smuggling will occur. For example, multiple `getBalance()` invocations on an `Account` object do not conflict. This is not surprising, since `getBalance()` does not modify the state of an account. However, two `deposit()` operations do not conflict either; they *commute*. Generally speaking, the decision of what methods may cause a conflict depends on the semantics of the method, the method input and output parameters, the structure of the object state, and the object usage [24].

Other parts of the transaction support can also benefit from the knowledge of method semantics. If, for instance, every method has an associated *inverse method*, which undoes the effects of the former one, then the recovery manager can perform logical logging instead of physical logging, if appropriate.

Obviously, such semantic knowledge about methods can not be guessed automatically. It must be provided by the application programmer. In OPTIMA, this information is encapsulated in the abstract `Operation` class. Subclasses of `Operation` must implement methods such as `isCompatible(Operation op)`, which must determine if the current operation conflicts with the operation `op` passed as a parameter.

```
public class GetBalanceOperation extends Operation {
   boolean isCompatible(Operation op) {
      return (op instanceof GetBalanceOperation);
   }
}
```

Fig. 16: The `GetBalanceOperation` Class

Figure 16 depicts parts of the declaration code of the `GetBalanceOperation` class. It specifies that calls to the `getBalance()` method from one transaction are compatible with calls to `getBalance()` from other transactions, but incompatible with all other methods invocations on the `Account` class.

Following the same idea shown in Section 6.2 for customizing concurrency control, classes that want to customize their transactional behavior on a per-method basis must implement the `CustomizedMethods` interface shown in Figure 17.

```
public interface CustomizedMethods {
   public Operation getOperation(String name, JoinPoint jp)
      throws MethodCustomizationException;
}
```

Fig. 17: The `CustomizedMethods` Interface

The aspect shown in Figure 18 adds this functionality to the `Account` class. The implementation of the method `getOperation` may make use of the `JoinPoint` parameter `jp`. In `AspectJ`, `JoinPoint` objects provide access to run-time information, e.g., parameter values. In Figure 18, `jp` is used when the deposit method is invoked to extract the value of the parameter that holds the amount of money to be deposited.

```
public aspect AccountMethodAspect {
   declare parents: Account implements CustomizedMethods;
   public Operation CustomizedMethods.getOperation(String name, JoinPoint jp)
      throws MethodCustomizationException {
      if (name.equals("getBalance")) {
         return new GetBalanceOperation();
      } else if (name.equals("deposit")) {
         return new DepositOperation((Integer)jp.getArgs()[0]);
      } else {
         throw MethodCustomizationException;
      }
   }
}
```

Fig. 18: Customizing All Methods of the `Account` Class

If no operation subclass is defined for a given method, or if the class does not implement the `CustomizedMethods` interface, the default operation class is used as shown in Figure 19 ①. The default operation class assumes the worst: the operation is assumed to modify the state of the object, and is assumed to conflict with any other operation.

```
public aspect OptimaConfiguration issingleton() {
   // code shown in Figure 11 and Figure 15
   public Operation getOperation(Object o, JoinPoint jp) {
      if (o instanceof CustomizedMethods) {
         // get the customized Operation object
         Operation op;
         try {
            op = ((CustomizedMethods)o).getOperation
               (jp.getSignature().getName(), jp);
         } catch (MethodCustomizationException e) {
            return new DefaultOperation();
         }
         return op;
      } else {
         return new DefaultOperation(); ①
      }
   }
}
```

Fig. 19: Default Operation Object for All Methods

6.4 Transactional Objects with Customization

The previous sections have shown how to add customization at the application, object and method level. To encapsulate transactional objects and at the same time provide customization, the aspect presented in Figure 9 has been extended. The result is shown in Figure 20.

The aspect `CustomizedTransactionalObject` is still specified to be instantiated `pertarget(TransactionalMethods)`. At instantiation time, the recovery manager is initialized to the one given in the `OptimaConfiguration` aspect ①. The

concurrency control for the transactional object is initialized when a public method is invoked from within a transaction for the first time by calling `initConcurrency Control()` ② (see Section 6.2).

Before the method is executed, the `Operation` object for the method is obtained by calling `getOperation()` ③. The returned `Operation` object is passed to the concurrency control `preOperation()` method ④.

```
aspect CustomizedTransactionalObject pertarget(TransactionalMethods()) {
   pointcut TransactionalMethods() : call(public * Account.*(..));
   private final RecoveryManager myRecoveryManager =
      OptimaConfiguration.aspectOf().recoveryManager; ①
   private ConcurrencyControl myConcurrencyControl;
   // other per-object info, e.g. recovery info
   public void abortTransaction(Object object, Transaction t) {
      myConcurrencyControl.transactionAbort(t);
   }
   public void commitTransaction(Object object, Transaction t) {
      myConcurrencyControl.transactionCommit(t);
   }
   before() : TransactionalObjectMethodCall() {
      Transaction t = TransactionContext.getTransaction();
      if (t != null) {
         Object currentObject = thisJoinPoint.getTarget();
         if (myConcurrencyControl == null) {
            myConcurrencyControl = OptimaConfiguration.aspectOf()
                .initConcurrencyControl(currentObject); ②
         }
         Operation myOperation = OptimaConfiguration.aspectOf()
            .getOperation(currentObject, thisJoinPoint); ③
         myConcurrencyControl.preOperation(myOperation, t);
         myRecoveryManager.preOperation(currentObject, t); ④
      }
    }
   after() : TransactionalObjectMethodCall() {
      Transaction t = TransactionContext.getTransaction();
      if (t != null) {
         Object currentObject = thisJoinPoint.getTarget();
         myRecoveryManager.postOperation(currentObject, t);
         Operation myOperation = OptimaConfiguration.aspectOf()
            .getOperation(currentObject, thisJoinPoint);
         myConcurrencyControl.postOperation(myOperation, t);
      }
   }
}
```

Fig. 20: Making Objects Transactional with Customization

6.5 Extensibility

Care must be taken when modifying methods of transactional objects with customized transactional behavior, since modifications in the code might also modify the method semantics.

Consider a bank account that offers the usual `deposit` and `withdraw` operations, and an operation that returns the current interest rate `getInterestRate`. In general, bank accounts have fixed interest rates, and therefore invocations of `getInterestRate` do not conflict with `deposit` or `withdraw`. An implementation of such a bank account will exploit this property and implement customized methods to increase concurrency.

But what if a bank decides to offer better interest rates to "good" customers, meaning customers whose account balance exceeds a certain amount of money? In this case, calling `deposit` or `withdraw` might change the interest rate if the new balance passes the threshold. The point we want to make here is that such a modification inside a method of the `Account` class must be accompanied by a corresponding modification in the `AccountMethodAspect`; otherwise the ACID properties will be violated. Hence, although the transaction mechanisms are physically separated from the "functional" part of the account class, they remain semantically coupled. When performing maintenance, both parts must be updated in accordance.

7 Related Work

To the best of our knowledge, there has been no previously published work on providing support for transactions using AOP.

A widely used platform that promises what we called *transaction interface aspectization* is *Enterprise Java Beans* [5]. EJB is a higher-level component-based architecture for distributed business applications, which aims at simplifying the development of complex systems in Java by dividing the overall development process into seven different architecture roles that can be performed by different parties.

One of the architecture roles is the *Enterprise Bean Provider*. Typically performed by an application-domain expert, e.g. from the financial industry, the enterprise bean provider builds a component, called an *enterprise bean*, that implements the business methods without being concerned about the distribution, transaction, security, and other non-business-specific aspects of the application. The *EJB Container Provider* on the other hand is supposed to be an expert in distributed systems, transactions and security. The container provider must deliver tools for the deployment of enterprise beans, and a run-time system that provides the deployed beans with transaction and security management, distribution, management of resources, and other services. The other architecture roles are the *Persistence Manager Provider,* the *EJB Server Provider*, the *System Administrator*, the *Application Assembler*, and finally the *Deployer*.

Entity beans provide an object view of data in a database, and typically access this data from within a transaction. However, the methods of an enterprise bean do not handle transactions directly. Instead, transactional properties are specified in the *deployment descriptor* of a bean. Possible transaction policies are presented in figure 21.

The transaction policies can be set by the bean provider for the entire bean or for each method separately. But, surprisingly, these policies can later on be changed by the application assembler, or even by the deployer. It is also possible to change the isolation level for an entire bean, or even for each method separately. Isolation levels, however, are not standardized. In Visual Age for Java, for instance, possible isolation levels (from strongest to weakest) are TRANSACTION_SERIALIZABLE, TRANSACTION_REPEATABLE_READ, TRANSACTION_READ_COMMITTED, TRANSACTION_READ_UNCOMMITTED.

Based on our experience, changing the transaction policies and isolation levels defined by the bean provider is highly error-prone. Only the implementor of the bean knows the exact semantics of the methods, and is qualified to select the appropriate

Policy	Meaning
Tx_Not_Supported	The method can not be called from inside a transaction.
Tx_Supported	The method can be called from inside a transaction.
Tx_Mandatory	The method must be called from inside a transaction. If this is not the case, an exception is thrown to the caller.
Tx_Required	The method requires to be executed from inside a transaction. If this is not the case, a new transaction is created.
Tx_Requires_New	The container creates a new transaction before executing the method.
Tx_Bean_Managed	Session beans are allowed to manage transactions explicitly by calling `javax.transaction.CurrentTransaction`. This policy is not supported for entity beans.

Fig. 21: Enterprise Java Beans Transaction Policies

policies. Allowing a different person to fiddle with these properties at deployment time will inevitably lead to incorrect programs.

Another obvious drawback of the EJB approach is performance. When writing the entity bean methods, the bean provider does not have to worry about concurrent accesses by multiple transactions. The bean provider may assume that the container will ensure appropriate synchronization for entity objects that are accessed concurrently by multiple transactions.

Unfortunately, the container does not have any knowledge of the semantics of the methods of a bean, and therefore must make a "blind" choice when implementing concurrency control. The EJB specification mentions two different implementation strategies. The container can activate multiple instances of a bean, one for each transaction, and let the underlying database handle proper serialization. Depending on what kind of lock the `ejbLoad` method acquires, this may unnecessarily block read-only transactions, or lead to deadlocks. The other solution is to activate only a single instance of the entity bean, and serialize the accesses by multiple transactions to this instance, which also restricts concurrency among transactions dramatically.

8 Discussion

Our experience was limited to:

1. The use of `AspectJ` as a representative of AOP languages;
2. *Transactions* as a fundamental paradigm to handle concurrency and failures;
3. Our underlying Optima transactional framework to implement concurrency control and failure management.

Hence, in principle, one can hardly draw any conclusion on using AOP to aspectize concurrency and failures in general. Furthermore, the very fact that we could not smoothly aspectize concurrency and failures does in no way mean that other techniques to aspectize those concerns are bound to fail.

We have, however, tried to explore different possibilities, and we considered mainly issues of general importance without focussing on technical issues related, for instance, to the current implementation of `AspectJ`. For example, the current `AspectJ` implementation only advises the parts of an application for which source code is available at compile time, excluding, for instance, code in precompiled libraries such as `java.lang`. This restriction is an additional reason why the *aspectizing transactions* approach presented in Section 4 is impossible. We also ignored technical problems like aliasing of transactional objects, serializing references to transactional objects, and static fields of classes.

Our underlying thesis is, however, that concurrency control and failure management are hard to aspectize in general, and we argue below that this is actually not surprising.

- On one hand, existing transactional languages, e.g. Argus [25], Arjuna [26], KAROS [27], Transactional Drago [28] or PJama [29, 30, 18], provide primitives for expressing transaction boundaries within methods, and not as separate concerns. Furthermore, even if the systems underlying those languages provide default mechanisms for handling concurrency and failures, most work on how to obtain effective mechanisms advocate the tight integration of the mechanisms within the actual methods or objects [21, 31, 32]. The difficulty of providing local concurrency control mechanisms and the strong integration with recovery management is pointed out in [25].
- On the other hand, object-oriented programming is about modeling real-world phenomenon with objects. Each object is supposed to encapsulate the state and the behavior of a real world phenomena, and concurrency and failures are usually parts of that phenomena. For instance, the very fact that a transaction should be aborted if there is not enough money in a bank account is a full part of the semantics of the bank account. Similarly, one would hate to get the actual balance of his bank account during a transfer.

9 Summary

We considered three levels of aspectization in our transactional context. The results of our experiment are summarized below:

Aspectizing transactions: Trying to automatically apply transactions to previously non-transaction code is doomed to failure, because of the incompatibility of the linearizability of method invocations provided by shared objects and transaction serializability, and because of the impossibility to automatically identify irreversible actions.

Aspectizing transaction interfaces: Separating transactional interfaces from the rest of the program can be achieved using aspect-oriented programming techniques. This separation, however, might seem artificial in situations where the “transactional aspect” actually is part of the semantics of the object it applies to. Each object is supposed to encapsulate the state and the behavior of a real world phenomena, and concurrency and failures are usually parts of that phenomena. In such situations, an indirect connection between functional and transactional concerns must be established, for instance by using exceptions. This, however, might lead to rather confusing code.

Another drawback of aspectizing transaction interfaces and not exposing transaction mechanisms is that default choices must be made by the underlying transaction support, which can considerably impact performance.

Aspectizing transaction mechanisms: AOP languages provide interesting features that can simplify the separation, at the syntactic level, of concurrency control and failure management mechanisms from the rest of the objects. Pretty much like an advanced macro language, however, these features should be reserved for smart programmers who have an advanced sense of the risk [4]. Physical separation does not necessarily imply semantic decoupling, and at least in the case of transactions, we believe that the application programmer and the programmer applying transactions using aspects must be the same person. Approaches such as the one taken by Enterprise Java Beans, where the application deployer can set or change transactional attributes for each Java bean, are error-prone, since they can easily lead to the violation of the ACID properties of transactions.

To prevent such problems, and to help aspect-oriented programmers, some guidelines and tool support would be useful, e.g., the ability to display "tightly coupled" aspects applying to an object whenever the implementation of the object changes. It might also be interesting to clearly emphasize that what can possibly be safely aspectized is probably what is not part of the object semantics, i.e., of the phenomena that the object is supposed to simulate, e.g., debugging and display. Drawing that borderline would be another interesting exercise.

In short, although the thesis underlying our experiment is not surprising, we believe nevertheless that the experiment itself provides some material for discussing what can be aspectized and what cannot. Given the growing interest in AOP, such a discussion can be of great value.

10 Acknowledgements

We are very grateful to Andrew Black and Pierre Cointe for many fruitful discussions on AOP. Besides the anonymous reviewers, we would also like to thank Bjorn Freeman-Benson, Gregor Kiczales and Mira Mezini for their comments on earlier drafts of this paper. Jörg Kienzle has been partially supported by the Swiss National Science Foundation project FN 2000-057187.99/1.

References

[1] Madsen, O. L.; Møller-Pederson, B.: "What object-oriented programming may be - and what it does not have to be". In Gjessing, S.; Nygaard, K. (Eds.), *2nd European Conference on Object–Oriented Programming (ECOOP '88)*, pp. 1 – 20, Olso, Norway, August 1988, Lecture Notes in Computer Science **322**, Springer Verlag.

[2] Elrad, T.; Aksit, M.; Kiczales, G.; Lieberherr, K.; Ossher, H.: "Discussing Aspects of AOP". *Communications of the ACM **44**(10)*, pp. 33–38, October 2001.

[3] Gray, J.; Reuter, A.: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, California, 1993.

[4] Guerraoui, R. “AOP = SMP (Structured Macro Programming)”, Panel at the 14th European Conference on Object–Oriented Programming (ECOOP ’2000), Cannes, France, June 2000.

[5] Shannon, B.; Hapner, M.; Matena, V.; Davidson, J.; Pelegri-Llopart, E.; Cable, L.: *Java 2 Platform Enterprise Edition: Platform and Component Specification*. The Java Series, Addison Wesley, Reading, MA, USA, 2000.

[6] Kienzle, J.: *Open Multithreaded Transactions: A Transaction Model for Concurrent Object-Oriented Programming*. Ph.D. Thesis #2393, Swiss Federal Institute of Technology, Lausanne, Switzerland, April 2001.

[7] Kienzle, J.; Romanovsky, A.; Strohmeier, A.: “Open Multithreaded Transactions: Keeping Threads and Exceptions under Control”. In *Proceedings of the 6th International Worshop on Object-Oriented Real-Time Dependable Systems, Universita di Roma La Sapienza, Roma, Italy, January 8th - 10th, 2001*, pp. 209 – 217, IEEE Computer Society Press, 2001.

[8] Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersen, M.; Palm, J.; Griswold, W. G.: “An Overview of AspectJ”. In *15th European Conference on Object–Oriented Programming (ECOOP 2001)*, pp. 327 – 357, June 18–22, 2001, Budapest, Hungary, 2001.

[9] Kienzle, J.; Jiménez-Peris, R.; Romanovsky, A.; Patiño-Martinez, M.: “Transaction Support for Ada”. In *Reliable Software Technologies - Ada-Europe’2001, Leuven, Belgium, May 14-18, 2001*, pp. 290 – 304, Lecture Notes in Computer Science **2043**, Springer Verlag, 2001.

[10] Lee, P. A.; Anderson, T.: “Fault Tolerance - Principles and Practice”. In *Dependable Computing and Fault-Tolerant Systems*, Springer Verlag, 2 ed., 1990.

[11] Hoare, C. A. R.: “Parallel Programming: an Axiomatic Approach”. In Bauer, F. L.; Samelson, K. (Eds.), *Proceedings of the International Summer School on Language Hierarchies and Interfaces*, pp. 11 – 42, Marktoberdorf, Germany, July 1975, Lecture Notes in Computer Science **46**, Springer Verlag.

[12] Horning, J. J.; Randell, B.: “Process Structuring”. *ACM Computing Surveys **5**(1)*, pp. 5 – 30, March 1973.

[13] Brinch Hansen, P.: *Operating System Principles*. Prentice Hall, 1973.

[14] Hoare, C. A. R.: “Monitors: An Operating Systems Structuring Concept”. *Communications of the ACM **17**(10)*, pp. 549 – 557, October 1974.

[15] Herlihy, M.; Wing, J.: “Linearizability: a correctness condition for concurrent objects”. *ACM Transactions on Programming Languages and Systems **12**(3)*, pp. 463 – 492, July 1990.

[16] Papadimitriou, C.: “The serializability of concurrent database updates”. *Journal of the ACM **26**(4)*, pp. 631 – 653, October 1979.

[17] Romanovsky, A. B.; Shturtz, I. V.: “Unplanned recovery for non-program objects”. *Computer Systems Science and Engineering **8**(2)*, pp. 72–79, April 1993.

[18] Daynès, L.: “Implementation of automated fine-granularity locking in a persistent programming language”. *Software — Practice & Experience **30**(4)*, pp. 325 – 361, April 2000.

[19] Romanovksy, A.; Kienzle, J.: "Action-Oriented Exception Handling in Cooperative and Competitive Object-Oriented Systems". In Romanovsky, A.; Dony, C.; Knudsen, J. L.; Tripathi, A. (Eds.), *Advances in Exception Handling Techniques*, pp. 147 – 164, Lecture Notes in Computer Science **2022**, Springer Verlag, 2001.

[20] Lampson, B. W.; Sturgis, H. E.: "Crash Recovery in a Distributed Data Storage System". *Technical report*, XEROX Research, Palo Alto, June 1979.

[21] Bernstein, P. A.; Goodman, N.: "Concurrency Control in Distributed Database Systems". *ACM Computing Surveys* ***13****(2)*, pp. 185 – 221, June 1981.

[22] Bernstein, P. A.; Hadzilacos, V.; Goodman, N.: *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[23] Kung, H. T.; Robinson, J. T.: "On Optimistic Methods for Concurrency Control". *ACM Transactions on Database Systems* ***6****(2)*, pp. 213 – 226, June 1981.

[24] Ramamritham, K.; Chrysanthis, P. K.: "Advances in Concurrency Control and Transaction Processing". Los Alamitos, California, 1997.

[25] Liskov, B.: "Distributed Programming in Argus". *Communications of the ACM* ***31****(3)*, pp. 300 – 312, March 1988.

[26] Shrivastava, S. K.: "Lessons Learned from Building and Using the Arjuna Distributed Programming System". In Birman, K.; Mattern, F.; Schiper, A. (Eds.), *Theory and Practice in Distributed Systems*, pp. 17 – 32, Lecture Notes in Computer Science **938**, 1995.

[27] Guerraoui, R.; Capobianchi, R.; Lanusse, A.; Roux, P.: "Nesting Actions through Asynchronous Message Passing: the ACS Protocol". In Madsen, O. L. (Ed.), *6th European Conference on Object–Oriented Programming (ECOOP '92)*, pp. 170 – 184, Utrecht, The Netherlands, June 1992, Lecture Notes in Computer Science **615**, Springer Verlag.

[28] Patiño-Martinez, M.; Jiménez-Peris, R.; Arévalo, S.: "Integrating Groups and Transactions: A Fault-Tolerant Extension of Ada". In *Reliable Software Technologies - Ada-Europe'98, Uppsala, Sweden, June 8-12, 1998*, pp. 78 – 89, Lecture Notes in Computer Science **1411**, 1998.

[29] Atkinson, M. P.; Daynès, L.; Jordan, M. J.; Printezis, T.; Spence, S.: "An orthogonally persistent Java". *ACM SIGMOD Record* ***25****(4)*, pp. 68 – 75, December 1996.

[30] Daynès, L.: "Extensible Transaction Management in PJava". In *Proceedings of the First International Workshop on Persistence and Java, University of Glasgow, UK*, September 1996.

[31] Weihl, W. E.: "Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types". *ACM Transactions on Programming Languages and Systems* ***11****(2)*, pp. 249 – 283, April 1989.

[32] Guerraoui, R.: "Atomic Object Composition". In Tokoro, M.; Pareschi, R. (Eds.), *8th European Conference on Object–Oriented Programming (ECOOP '94)*, pp. 118 – 138, Bologna, Italy, June 1994, Lecture Notes in Computer Science **821**, Springer Verlag.