Technical Report number SOCS-TR-2007.1, School of Computer Science, McGill University, February 2007.

AO Challenge, Part II: Lessons Learnt from Implementing a Reusable Aspect Framework

Ekwa Duala-Ekoko and Jörg Kienzle

School of Computer Science,
McGill University,
Montreal, QC H3A 2A7, Canada
Ekwa.Duala-Ekoko,Joerg.Kienzle@mcgill.ca

Abstract. Writing a set of reusable aspects that can be used in isolation and in combination is a highly complex task. In this paper we identify a set of key features required for implementing reusable aspects frameworks with complex aspect dependencies and interactions based on our experience implementing AspectOptima, a language independent, aspect-oriented framework that provides runtime support for transactions. By showing details of our implementation in AspectJ, we demonstrate how reusability and configurability, although not directly supported by AspectJ, can be achieved by means of aspect design patterns. Where appropriate, we suggest potential language improvements that address the encountered limitations. Finally, we present performance measures that compare our aspect-oriented solution to a purely object-oriented one, and highlight the impact of language support for reusability and configurability on performance.

1 Introduction

Aspect-orientation [7] has been accepted as a powerful technique for modularizing crosscutting concerns during software development in so-called *aspects*. From the beginning, programmers have also taken advantage of the advanced modularization offered by aspect-oriented programming techniques to improve code reusability. Experience has shown that aspect-oriented programming is successful in modularizing even very application-independent, general concerns such as distribution [23], concurrency [4,16], persistency [23,20] and failures [16]. The implementation of these concerns can be reused in several base applications by binding the reusable aspects to specific application elements. This binding is, however, sometimes tricky. The degree of reusability that can be achieved depends on the expressiveness of the aspect-oriented programming language.

Aspect-oriented frameworks such as [15,18,3] used aspects to modularize well-defined low-level concerns, and then combined them in different ways to implement higher-level concerns. The complexity of writing such aspect-oriented frameworks with individually reusable aspects is exponentially more difficult

because of the fact that aspects are used in combination with each other. Dependencies between aspects raise the question of how to express inter-aspect relationships such as aspect configurations and aspect ordering. Since the correct configuration or ordering of aspects is often application-dependent, mechanisms must be devised that allow an application developer to specify the desired behavior. However, unwanted and undesired interactions among aspects have to be prevented, and functional dependencies between aspects have to be respected.

This paper aims at investigating the properties of aspect-oriented programming languages that enable the development of reusable aspect-oriented frameworks. To illustrate our ideas, we present our experience in implementing the aspect-oriented framework AspectOptima – a language independent framework consisting of a set of ten base aspects (each one providing a well-defined reusable functionality) that can be assembled in different configurations to provide runtime support for transactions. Section 2 briefly describes transactional systems, and then presents a summary of the design of AspectOptima. We then identify eight key language features necessary for implementing aspect-oriented frameworks in section 3. We analyze how these features are supported in the programming language *AspectJ* in section 4, and illustrate the use of these features by presenting some details of our implementation of AspectOptima in section *5*. Section 6 presents an in-depth discussion of the encountered limitations of *AspectJ* and suggests potential language improvements. We show performance measures that illustrate the performance of our aspect implementation compared to a purely object-oriented implementation, and the performance impact of different aspect-oriented language constructs for achieving reusability in section 7. Section 8 comments on related work, and the last section presents a conclusion and future work.

2 Design Summary of AspectOptima

2.1 Transactions and Transactional Objects

A *transaction* groups together an arbitrary number of operations on *transactional objects*, guaranteeing the so-called *ACID properties*: Atomicity (either all operations are executed, or none is), Consistency(guarantees that the execution of a transaction will not erroneously corrupt the application state), Isolation (concurrently running transactions don't interfere with each other) and Durability [12](guarantees that the results of a committed transaction can be re-established in the event of failures).

When a process or thread that works inside a transaction calls a method on a transactional object, the transaction support middleware must take control and perform certain actions to ensure that the ACID properties can be guaranteed. Traditionally, this is done by applying concurrency control and recovery strategies. Concurrency control can be *pessimistic* (for instance, using locks) or *optimistic* (for instance, using time-stamps). Recovery can be done *in-place* (i.e., by checkpointing – saving the state of an object – and, in case of a roll-

back, restoring the state of transactional objects), or using *deferred-update* (i.e., by creating local copies of the state of transactional objects for each transaction).

2.2 The Ten Low-Level Aspects of AspectOptima

[15] describes the design of AspectOptima, an aspect-oriented framework that provides the ACID properties for transactional objects. The framework defines ten low-level aspects that can each be applied to an object to provide a well-defined reusable functionality. The aspects are briefly described below:

- **AccessClassified**: The *AccessClassified* aspect classifies each method that an object defines according to how its execution affects the object's state: each method is classified as a *read*, *write* or *update* method.
- **Named**: The *Named* aspect associates a name with an object that can be used as a unique means of identification.
- **Shared**: The *Shared* aspect ensures multiple readers/single writer access to objects – all modifications made to the state of a shared object are performed in mutual exclusion.
- **Copyable**: The *Copyable* aspect provides functionality to duplicate an object, or replace an object's state with the state from another object.
- **Serializable**: A serializable object knows how to read its state from and write its state to different devices requiring varying data representation formats, e.g. a file or a network connection. The *Serializable* aspect is an incarnation of the *Serializer* pattern described in [21].
- **Versioned**: A *Versioned* object can encapsulate multiple copies – *versions* – of its state. Versions are linked to *views*, one of which is designated the *main view*. A thread can subscribe to a view, and any method call made subsequently by the thread is directed to the associated version.
- **Tracked**: The *Tracked* aspect provides the functionality to monitor object access in a generic way. It allows a thread to define a region (using *begin* and *end* operations) in which object accesses are monitored. At any given time, the thread can obtain all read or modified objects for the current region.
- **Recoverable**: The *Recoverable* aspect makes it possible to store the state of an object at a given time, and later restore it, if needed. This functionality is sometimes also called "establishing a checkpoint".
- **AutoRecoverable**: The *AutoRecoverable* aspect provides region-based recovery. It allows a thread to define a region within which recoverable objects are automatically checkpointed before any modifications are made to their state.
- **Persistent**: *Persistent* objects are objects whose state survives program termination. To achieve this, persistent objects know how to write their state to a non-volatile storage device.

2.3 Aspect Dependencies and Interference in AspectOptima

The ten low-level aspects presented in the previous subsection exhibit complex dependencies. For instance, *AutoRecoverable* depends on the functionality provided by *Recoverable* to establish the checkpoints, which in turn depends on *Versioned* to keep a backup copy of an object's state, which in turn uses *Copyable* to duplicate an object. The left hand side of the UML diagram shown in Fig. 1 illustrates the dependency relationships among the low-level aspects.

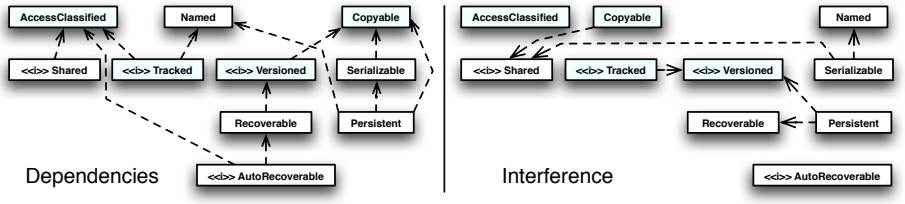

Fig. 1. Aspect Dependencies and Interferences

Some of the aspects also interfere with each other, i.e. they have to modify their behavior in the presence of the interfering aspect in order to provide correct service. For instance, *Serializable* should not serialize an object's state if it is shared and another thread is currently modifying its state. *Persistent*, for instance, must make sure that, when saving a *Recoverable* object, it also saves the backup version, if there is one. The interference dependencies between aspects are depicted on the right side of Fig. 1.

Some aspects only provide functionality when explicitly asked to. The functionality of others is triggered implicitly when the object to which they have been applied to is used. The aspects that have to intercept calls to their objects are marked with the stereotype `<<i>>` (for *interceptors*) in Fig. 1.

2.4 Composing Concurrency Control and Recovery Strategies

[15] describes the composition of several concurrency control and recovery strategies using different combination of the low-level aspects. For space reasons we only present one particular strategy here: *Pessimistic Lock-Based Concurrency Control with In-Place Update*.

It is implemented in the aspect *LockBased*, which relies on the fact that all lockbased transactional objects are also *AccessClassified*, *Named, Copyable*, *Serializable, Shared, Versioned, Tracked, Recoverable*, *AutoRecoverable* and *Persistent*[1]. The aspects also assume that the transaction runtime creates a tracked

[1] The functionality provided by *Persistent* is not used in this section. Persistency is mostly required at commit time of a transaction as shown in section 5.5.

zone, a recoverable zone and a new view when a transaction begins, and ends the zones and the view when a transaction commits or aborts. Fig. 2 illustrates the interaction between these aspects. The sequence diagram depicts how a call to a transactional object – *TAObject* – is intercepted, and how the individual aspects collaborate to provide the desired functionality.

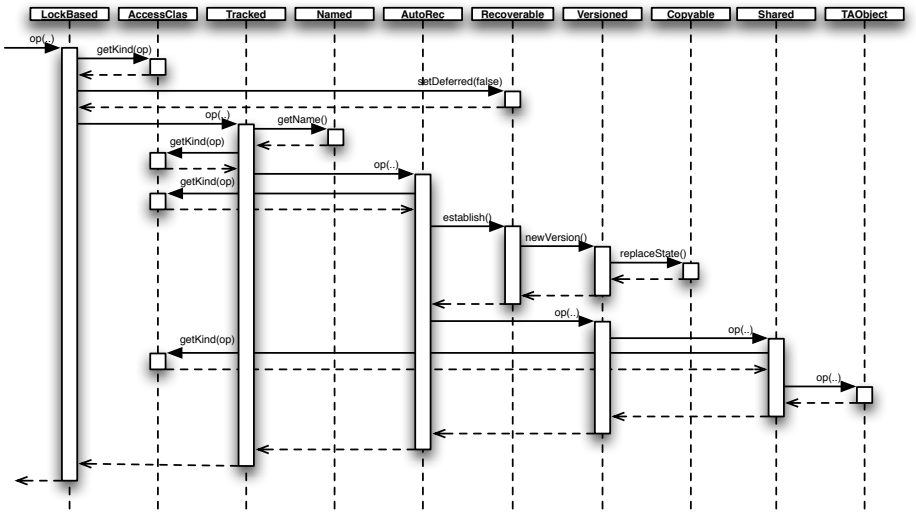

Fig. 2. Aspect Interactions for *LockBased* Objects

Lock-based protocols use locks to implement permissions to perform operations. When a thread invokes an operation (i.e., *op(..)* in Fig. 2) on a transactional object on behalf of a transaction, *LockBased* intercepts the call, forcing the thread to obtain the lock associated with the operation. The kind of lock – *read*, *write* or *update* – is chosen based on the information provided by *AccessClassified*. Before granting the lock, *LockBased* verifies that this new lock does not conflict with a lock held by a different transactions in progress. If a conflict is detected, the thread requesting the lock is blocked and has to wait for the release of the conflicting lock. Otherwise, the lock is granted. *LockBased* then makes sure that *in-place* update has been selected for this object by calling *Recoverable*, and allows the call to proceed.

The order in which locks are granted to transactions imposes an execution ordering on the transactions with respect to their conflicting operations. Two-phase locking [8] ensures serializability by not allowing transactions to acquire any lock after a lock has been released. This implies in practice that a transaction acquires locks during its execution (1st phase), and releases them at the end once the outcome of the transaction has been determined (2nd phase).

To release all acquired locks when a transaction ends, all transactional objects that are accessed during a transaction have to be monitored. To this end,

LockBased depends on *Tracked* to intercept the call and record the access. Obviously, an object should be tracked only after a lock has been granted.

Next, *LockBased* depends on *AutoRecoverable* to intercept the call and to checkpoint the state of the transactional object, if necessary, before it is modified. Since we are using in-place update, *Versioned* then directs the operation call to the main copy of the object. Finally, *Shared* intercepts the call and makes sure that no two threads running in the same transaction are modifying the object's state concurrently. After the method has been executed, *Shared* releases the mutual exclusion lock. The transactional lock, however, is held until the outcome of the transaction is known.

3 Language Requirements for Implementing Aspect Frameworks

Based on our experience implementing AspectOPTIMA, we identified eight key features that an aspect-oriented language has to provide in order to support the implementation of reusable aspect-oriented frameworks:

- **Aspect Packaging**:
  In order to support clear structuring and modularization, and to improve readability and maintainability, it should be possible to package the implementation of a aspect in such a way that the module contains all the code needed to implement the functionality. Proper packaging ensures that an aspect is used as a whole, which simplifies aspect binding and configuration, and hence provides a basis for safe reuse.
- **Separate Aspect Binding**:
  In order to support reusability, aspects should not contain explicit bindings to application elements. The aspect binding is written at a later time by the application developer that wants to apply the functionality offered by a reusable aspect to parts of his application.
- **Inter-Aspect Configurability**:
  In order to support safe reuse, a developer that needs a functionality offered by one aspect should not have to explicitly deal with aspect dependencies, i.e. when deploying an aspect A, all aspects that A depends on should be automatically deployed as well. Aspects have to be able to express their dependence on other aspects. In AspectOPTIMA, for example, *Versioned* can only be applied to objects that are also *Copyable*.
- **Inter-Aspect Incompatibility**:
  In order to support safe reuse, a developer of an aspect framework should be able to specify that certain combinations of aspects are illegal. Some aspects should never be applied to the same object. For example, *LockBased* can not be applied to objects that use some other form of concurrency control, e.g. *MultiVersion*.
- **Inter-Aspect Ordering**:
  In order to support reuse, a developer of an aspect A should be able to specify the order in which the aspects that A depends on are applied.

In AspectOPTIMA, for example, the aspect *LockBased* has to make sure that *Tracked* records the object access only *after* a lock has been acquired. Likewise, a developer that wants to use functionality provided by an aspect framework should be able to specify the ordering of functionally independent aspects when they are applied to the same part of the application.

- **Per-Object (Per-Instance) Aspects**:
  In order to support fine-grained reuse, support for per-object aspects is required. In AspectOPTIMA, for instance, an application programmer might want to use different implementations of ACID for different objects of the same class. It should therefore be possible to associate *LockBased*, *MultiVersion* and *Optimistic* to *objects*, not to classes.
- **Dynamic Aspects**:
  In order to support flexible reuse, support for dynamic aspects is required, i.e., it should be possible to apply aspects to and remove aspects from objects at runtime. In AspectOPTIMA, for example, in multi-version concurrency control, the *Shared* aspect should be removed from a version of a transactional object when it becomes read-only.
- **Thread-Aware Aspects**
  In order to support flexible reuse in multi-threaded applications, it should be possible to activate aspects on a per-thread basis. In AspectOPTIMA, several aspects provide functionality based on the context of the current thread. For instance, *Tracked* only tracks object accesses if the current thread has previously started a tracked zone. *AutoRecoverable* only checkpoints objects if the current thread is within an auto-recoverable zone.

4 Analysis of Language Support in AspectJ

This section describes how some of the features required to implement reusable aspect-oriented frameworks, although not directly supported, were achieved in our *AspectJ*[14] implementation of AspectOPTIMA.

AspectJ is an aspect-oriented extension of Java [11]. It encapsulates cross-cutting behaviors in a class-like construct called an *aspect*. Similar to a Java class, an aspect can contain both data members and method declarations, but it cannot be explicitly instantiated. *AspectJ* introduces four new concepts relevant to this work: *join points*, *pointcuts*, *advice*, and *inter-type declarations*.

Join points are well-defined points in the execution of a program. These include method and constructor calls or executions, field accesses, object and class initialization, and others. Our implementation of AspectOPTIMA uses only method call and execution join points.

A *pointcut* is a construct used to designate a set of join points of interest and to expose to the programmer the context in which they occur, such as the current executing object (*this(ObjectIdentifier)*), the target object of a call or execution (*target(ObjectIdentifier)*) and the arguments of the a method call (*args(..)*).

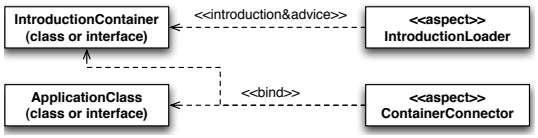

Fig. 3. Abstract Introduction Idiom

An *advice* defines the actions to be taken at the join point(s) captured by a pointcut. It consists of standard Java code. *AspectJ* supports three types of advice: *before*, *after* and *around*. The *before* advice runs just before the captured join point; the *after* advice runs immediately after the captured join point; the *around* advice surrounds the captured join point and has the ability to augment, bypass or allow its execution.

Finally, *inter-type declarations* allow an aspect to define methods and fields for other classes and interfaces, and for modifying the inheritance hierarchy.

4.1 Aspect Packaging, Separate Aspect Binding and Inter-Aspect Configurability

In *AspectJ*, the abstract introduction idiom (also known as indirect introduction) [13] can be used to achieve aspect packaging, separate aspect binding and inter-aspect configurability. The abstract introduction idiom allows us to "collect several extrinsic properties from different perspectives within one unit and defers the binding to existing objects". In order words, the target classes of the static and dynamic crosscutting behavior of an aspect can remain are unknown until weave-time. The strategy has three participants (see also Fig. 3):

- *Introduction container*: a construct used as the target for the inter-type member declarations.
- *Introduction loader*: the aspect that introduces crosscutting behaviors and ancestors to the introduction container.
- *Container connector*: the aspect used for connecting the introduction container to the base application classes.

The introduction container serves a dual purpose in the context of our implementation. First, it enables the aspects to be reused in different contexts; second, it helps in identifying the classes to which the crosscutting behavior of an aspect should be applied.

The introduction container can either be a class or an interface in *AspectJ*. Since multiple inheritance is not supported in Java, our implementation can not use a class as introduction container: it would prohibit several aspects to be applied to the same application object. Consequently, dummy interfaces are used as the introduction container for each of the aspects. For instance, the interface *IShared* is associated with the aspect *Shared*, *IAutoRecoverable* is associated with *AutoRecoverable,* and so on. Each of the AspectOPTIMA aspects, playing the role of the introduction loader aspect, is then implemented to apply its functionality

to all the classes that implement its associated interface (e.g., the *Shared* aspect is applied to all classes that implement the *IShared* interface).

In order to achieve aspect packaging, the interface, the aspect and the utility classes, if needed, are bundled together in the same Java package.

Since all AspectOPTIMA aspects declare dummy interfaces, individual aspect binding can be achieved using the *declare parents* construct of *AspectJ*. The first aspect in Fig. 4 brands the *Account* class as *IShared*; hence, the crosscutting behavior of the *Shared* aspect is applied to all instances of the *Account* class.

```
public aspect Binding {
  declare parents: Account implements IShared; }
public aspect AutoRecoverable{
  declare parents: IAutoRecoverable implements IRecoverable, IAccessClassified;}
```

Fig. 4. Separate Aspect Binding and Inter-Aspect Configurability in *AspectJ*

Inter-aspect configurability is achieved by having the associated interface of an aspect implement the interfaces of the aspects it depends on. For instance, the *AutoRecoverable* aspect declares *IAutoRecoverable* to implement *IAccessClassified* and *IRecoverable* as illustrated in the second aspect of Fig. 4. Hence, an *AutoRecoverable* object is by default *Recoverable* and *AccessClassified*. This technique makes reuse very easy and safe. Application developers do not have to modify their base classes to apply aspects to them and can not forget to deploy low-level aspects whose functionality is needed by a higher level aspect.

4.2 Inter-Aspect Ordering

Inter-aspect ordering is supported in *AspectJ* by the *declare precedence* construct. Fig. 5 illustrates how the *LockBased* aspect specifies its execution order relative to that of the aspects it depends on.

```
public aspect LockBased {
  declare precedence: LockBased,AutoRecoverable,Tracked,Versioned,Shared; }
```

Fig. 5. Inter-Aspect Ordering in *AspectJ*

4.3 Inter-Aspect Incompatibility

Inter-aspect incompatibility can be specified in *AspectJ* by writing two contradicting precedence declarations. Fig. 6 illustrates how to declare that the *LockBased* aspect should never be applied to the same join points as the *MultiVersion* or *Optimistic* aspects.

```
public aspect Incompatibility {
 declare precedence: LockBased, MultiVersion, Optimistic;
 declare precedence: Optimistic, MultiVersion, LockBased; }
```

Fig. 6. Inter-Aspect Incompatibility Specification in *AspectJ*

4.4 Per-Object Aspects, Dynamic Aspects and Thread-Aware Aspects

AspectJ does not support per-object aspects, dynamic weaving or thread-aware aspects. However, per-object aspect can be simulated by introducing a boolean field into each advised object. At each pointcut occurrence, the field is checked to verify that the aspect is actually enabled (see subsection 5.3).

Likewise, thread-aware aspects can be simulated by instantiating the Java class `ThreadLocal`, which allows a programmer to associate any data structure with a thread. Each aspect that needs to be thread-aware can associate a boolean field with a thread, which is checked at runtime to determine if the aspect should execute its functionality or not (see subsection 5.4).

5 *AspectJ* Implementation of AspectOPTIMA

In this section, we present a detailed description of the implementation of some of the AspectOPTIMA aspects in *AspectJ*. Due to space constraints, only the aspects necessary to discuss the encountered *AspectJ* limitations, namely *AccessClassified*, *Copyable*, *Shared*, *Tracked* and *LockBased*, are presented. The interested reader is referred to [5] for a complete description of the implementation.

5.1 Implementation of AccessClassified

It is currently not possible in *AspectJ* to statically determine if a method potentially reads, writes or updates the fields of an object. Therefore, our implementation of *AccessClassified* relies on the application developer to tag every method of an object with marker annotations, such as the *Read* annotation defined in Fig. 7. The annotation has a runtime retention policy (retained by the virtual machine so that it can be read reflectively at runtime), can be inherited (annotations on superclasses are automatically inherited by subclasses) and has to be applied to methods.

The *AccessClassified* implementation aspect shown in Fig. 7 introduces a method `getKind(String)` to every *IAccessClassified* object that examines these annotations by reflection at runtime and classifies each operation accordingly. Non-annotated methods are treated as modifier operations to guarantee system consistency.

A different implementation strategy that *automatically* determines the access kind at runtime by tentatively executing the method and by intercepting all field modifications is presented in [5].

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@Inherited public @interface Read {}

public aspect AccessClassified {
  public enum Kind {READ, WRITE, UPDATE);
  boolean found = false;
  public Kind IAccessClassified.getKind(String methodName)
    throws MethodNotAnnotatedException, MethodNotFoundException{
    for (Method m : this.getClass().getMethods()) {
      if((methodName.trim()).equalsIgnoreCase(m.getName())) {
        found = true;
        if (m.isAnnotationPresent(Read.class)) return READ;
        else if (m.isAnnotationPresent(Write.class)) return WRITE;
        else return UPDATE;
      }
    }
    if (!found) throw new MethodNotFoundException(".."); } }
```

Fig. 7. *AspectJ* Implementation of *AccessClassified*

```
public aspect Copyable {
  public void ICopyable.replaceState(Object source) {
    try {
      // field-by-field deep copy of the state (inherited, declared and introduced)
    } catch (SourceClassNotEqualDestinationClass e) {}
  }
  public Object ICopyable.clone(){
    // deep cloning using Java reflection
    return deepCopyOfOriginalObject; } }
```

Fig. 8. *AspectJ* Implementation of *Copyable*

5.2 Implementation of Copyable

The *Copyable* aspect (see Fig. 8) introduces state replacement and cloning functionality to all classes that implement the `ICopyable` interface. The method `replaceState` enables an object to replace its state with that of another object of the same class. The `clone` method duplicates an object. The actual copying/cloning heavily uses the Java reflection API. The lengthy code is not shown for space reasons.

5.3 Implementation of Shared

The implementation of the *Shared* aspect is presented in Fig. 9. *Shared* depends on the method classification provided by *AccessClassified* to determine the kind of lock to be acquired. The *declare parents* construct used in line 2 illustrates inter-aspect configuration by declaring all *Shared* objects to be *AccessClassified* as well.

```
1 public aspect Shared {
2   declare parents: IShared implements IAccessClassified;
3   public boolean IShared.Enabled = true;
4   private boolean IShared.getEnabled() { return Enabled; }
5   static boolean isEnabled(IShared object) { return object.getEnabled(); }
6   private Lock IShared.threadLock = new Lock();
7   private Lock IShared.getSharedLock() { return threadLock; }
8   pointcut methodExecution(IShared ishared): target(ishared) &&
      execution(public * IShared+.*(..));
9   Object around(IShared shared) : methodExecution(shared) &&
      if(isEnabled(shared)) {
10    Kind accessType = shared.getKind(getMethodName(thisJoinPoint));
11    if (accessType == READ) shared.getSharedLock().getReadLock();
12    else if (accessType == WRITE) shared.getSharedLock().getWriteLock();
13    else shared.getSharedLock().getUpdateLock();
14    Object obj = proceed(shared);
     // release previously acquired lock...
20    return obj; } }
```

Fig. 9. *AspectJ* Implementation of *Shared*

Lines 3-5 define a boolean field and two methods for supporting runtime disabling and enabling of advice on a per-object basis. The `if(isEnabled(shared))` pointcut modifier on line 9 checks the field before executing the functionality provided by *Shared*.

Lines 6-7 allocate a lock for each *Shared* object. The pointcut on line 8 makes sure that all public method calls to a *Shared* object are intercepted. The *around* advice on line 9 obligates every thread executing a method on a *Shared* object to acquire the appropriate lock before proceeding. On line 10, the kind of operation is determined by calling the functionality offered by *AccessClassified*. Line 11 to 13 obtain the corresponding lock. After the operation is executed (line 14), the lock is released again (code not shown for space reasons).

5.4 Implementation of Tracked

Fig. 10 presents an implementation of the *Tracked* aspect. It depends on the *AccessClassified* aspect to distinguish between *read*, *write* and *update* operations, and on the *Named* aspect to avoid tracking different copies of the same transactional object (see line 2). *InheritableThreadLocal,* a class provided by the standard Java API, is used to associate a thread with a zone. The *Zone* class is a simple helper class that maintains three hash tables to keep track of read, written and updated objects. The code of the *Zone* class is not shown due to space constraints. Tracked zones are requested by executing the aspect method *beginTrackedZone(),* and terminated by executing the aspect method *endTrackedZone()* (see lines 9 - 12)[2].

[2] For space reasons, the code dealing with joining and leaving, as well as nested zones has been omitted.

```
1 public aspect Tracked {
2   declare parents: ITracked implements INamed, IAccessClassified;
3   private static InheritableThreadLocal myZone = new InheritableThreadLocal();
4   pointcut methodCall(ITracked track) : target(track) &&
      call(public * ITracked+.*(..));
5   before(ITracked track) : methodCall(track) && if(myZone.get() != null) {
6     Kind type = track.getKind(getMethodName(thisJoinPoint.toShortString()));
7     String myName = ((ITracked)thisJoinPoint.getTarget()).getName();
8     ((Zone)myZone.get()).recordAccess(track, type, myName);
    }
9   public static synchronized void beginTrackedZone(){
10    if (myZone.get() == null) myZone.set(new Zone());
    }
11  public static synchronized void endTrackedZone() {
12    myZone.set(null);
    } }
```

Fig. 10. Implementation of *Tracked*

The pointcut at line 4 makes sure that all public method calls to tracked objects are intercepted. The *before* advice (lines 5 - 10) only executes if the call is made from within a tracked zone thanks to the *if* pointcut modifier. Line 6 shows how *Tracked* calls `getKind`, a functionality provided by *AccessClassified*. Likewise, line 7 calls `getName`, a functionality provided by *Named*, to obtain the object's identity. Finally, line 8 records the access in the zone object.

5.5 Implementation of LockBased

The *LockBased* aspect provides support for pessimistic lock-based concurrency control with in-place update (Fig. 11). To accomplish this, it depends on the following aspects: *AccessClassified* (to determine the appropriate transactional lock to acquire for a given transaction), *Shared* (to prevent threads within a transaction from concurrently modifying an object's state), *AutoRecoverable* (to gather undo information in case a transaction aborts), *Tracked* (to keep track of the transactional objects that participate in a transaction) and *Persistent* (to store the state of the object on stable storage when a transaction commits). The inter-aspect configuration is done using the *declare parents* statement of line 2.

The execution order of these aspects is crucial. An unspecified ordering could result in bad performance, deadlock or in the worst case even break the ACID properties. The desired execution order is: *LockBased*, *AutoRecoverable*, *Tracked*, *Versioned* and *Shared*. *LockBased* first has to acquire the transactional lock and set the update strategy to in-place before *AutoRecoverable* executes, the object is then *Tracked*, the operation directed to the main version by *Versioned*, and mutual exclusion to the state of the object ensured by *Shared* as shown in Fig. 2. This ordering is configured using the *declare precedence* statement in line 3.

Lines 4 and 5 allocate an instance of *TransactionalLock* for each lockbased object. The *TransactionalLock* class is a helper class that implements transaction-

aware read/write locks. The `acquire` method suspends the calling thread if some other transaction is already holding the lock in a conflicting mode.

The pointcut in line 6 makes sure that all public method calls to a *LockBased* object are intercepted. The *before* advice first queries the current transaction in line 8 (details on transaction life cycle management are out of the scope of this paper). In line 10, the functionality of *AccessClassified* is used to classify the operation that is to be invoked. Line 11 attempts to acquire the transactional lock for the current transaction in the corresponding mode. If successful, line 12 sets the update strategy by using functionality provided by *Recoverable*.

Unlike *Shared*, *LockBased* follows the two-phase locking protocol, and therefore holds on to the transactional locks until the outcome of the transaction is known. In case of transaction commit, *LockBased* performs the two-phase commit protocol. The first phase is done by the *before* advice on lines 13 - 15. It obtains all modified objects of the transaction by using the functionality provided by *Tracked*, and saves all pre- and post-states to stable storage using the functionality provided by *Persistent*. The second phase is handled by the *after* advice in lines 16 - 22. It discards the checkpoints of all modified objects using the functionality provided by *Recoverable*, saves their final states to stable storage using the functionality of *Persistent*, and then releases the transactional locks of all accessed objects.

The *after* advice on lines 22 - 26 handles transaction abort. It first rolls back all changes made to modified objects using the functionality provided by *Recoverable* and then releases the transactional locks.

5.6 Using AspectOptima

Line 1 of Fig. 12 shows how a programmer can declare an application class, in this case the class `Account`, and apply the *LockBased* aspect to it by simply declaring the class to implement *ILockBased*. The `getBalance` and `credit` methods are classified as *read* or *update* operations using the marker annotations of *AccessClassified* in line 3 and 4.

6 Encountered AspectJ Limitations

This section provides an in-depth discussion of the encountered *AspectJ* limitations, possible work-around solutions, and suggestions for improvements to the *AspectJ* language features, where appropriate.

6.1 Weak Aspect-to-Class Binding

An object in an *AspectJ* environment has several types of methods: those inherited from super classes and super interfaces, those declared by the class, and those introduced by aspects through direct or indirect introductions. As explained in section 4.1, our implementation achieves aspect reusability, separate

```
1 public aspect LockBased {
2  declare parents: ILockBased implements
     IAccessClassified, IShared, IAutoRecoverable, ITracked, IPersistent;
3  declare precedence: LockBased, AutoRecoverable, Tracked, Versioned, Shared;
4  private TransactionalLock ILockBased.lock = new TransactionalLock();
5  private TransactionalLock ILockBased.getLock() { return lock; }
6  pointcut methodCall(ILockBased lb) : target(lb) &&
     call(public * ILockBased+.*(..));
7  before (ILockBased lb) : methodCall(lb) {
8    Transaction t = getCurrentTransaction();
9    if (t != null) {
10    Kind accessType = lb.getKind(getMethodName(thisJoinPoint.toShortString()));
11      lb.getLock().acquire(t, accessType);
12      lb.setDeferred(false);
   } }
13 before (Transaction t) : call(public void Transaction.commit()) && target(t) {
14   for (ILockBased lb : Tracked.getModifiedObjects()) {
15     lb.saveState();
   } }
16 after (Transaction t) : call(public void Transaction.commit()) && target(t) {
17   for (ILockBased lb : Tracked.getModifiedObjects()) {
18     lb.discardCheckpoint(); lb.saveState(); }
20   for (TransactionalLock l : Tracked.getAccessedObjects()) {
21     l.releaseLock(t);
   } }
22 after (Transaction t) : call(public void Transaction.abort()) && target(t) {
23   for (ILockBased lb : Tracked.getModifiedObjects()) {
24     lb.restoreCheckpoint(); }
25   for (TransactionalLock l : Tracked.getAccessedObjects()) {
26     l.releaseLock(t);
   } } }
```

Fig. 11. *AspectJ* Implementation of *LockBased*

aspect binding and inter-aspect configurability by using the abstract introduction idiom [13]. Extrinsic static crosscutting behavior is collected in dummy interfaces and these interfaces are later bound to application classes using the *declare parents* construct. For instance, declaring an *Account* class as implementing *ICopyable* introduces two additional public operations: `replaceState` and `clone` into every *Account* object.

Unfortunately, *Copyable* interferes with *Shared*, in the sense that it should not be possible to copy or clone an object while it is being modified. Assuming that the previous *Account* class also implements *IShared* (such as, for instance, required by the *LockBased* aspect), it seems logical to assume that the call and execution of `Account.replaceState(..)` will be captured by the pointcuts *call(public * IShared+.*(..))* and *execution(public * IShared+.*(..))* of the *Shared* aspect, since the method `replaceState` is defined for the *Account* class. This is unfortunately not the case. The actual call and execution join points are

```
1 public class Account implements ILockBased {
2  private float balance;
3  @Read public float getBalance() { return balance; }
4  @Update public void credit(float amount) {balance += amount; } }
```

Fig. 12. A Lockbased Account

```
1 placeholder PCopyable {
2   public void clone() {...}
3   public void replaceState(Object o) {...}
  }
4 aspect Copyable {
5   apply PCopyable to Account; }
```

Fig. 13. Proposed "placeholder" Construct

call(ICopyable.replaceState(..)) and *execution(ICopyable.replaceState(..))*, respectively. *AspectJ* performs *weak aspect-to-class binding*, i.e. it associates the call and the execution join points of indirectly introduced methods with the *introduction container,* and not with the application class. As a result, *Shared* does not intercept calls to `replaceState`, which may lead to state inconsistencies if a thread executes a write or update operation while a different thread tries to copy the state of the object. This deficiency is not unique to AspectOPTIMA – any two aspects that interfere and work at the granularity of methods could suffer from the weak aspect-to-class binding problem.

In our case, a possible work-around is to declare the *ICopyable* interface as implementing *IShared*. In this case, the `replaceState` and `clone` method calls are intercepted by *Shared* as desired. An unfortunate side effect though is that *Copyable* is not individually reusable anymore: all *Copyable* objects are now also *Shared*, even if the application is single-threaded. This proposed work-around can not solve the problem for circularly interfering aspects.

Language Improvement Suggestion: The weak aspect-to-class binding problem could be overcome by adding a new class-like construct to *AspectJ* that we called a *placeholder*. A placeholder can define fields and methods, but these members should not be structurally bound to the placeholder. Its functionality should exclusively be to hold static crosscutting behavior that, at weave time, is bound to the target class it is applied to. A *placeholder* should not be instantiable, should never have a superclass, superinterface or be part of an inheritance hierarchy.

Fig. 13 shows a potential declaration of *PCopyable*, a *placeholder* to be used in the implementation of the *Copyable* aspect. Lines 1-3 define the *placeholder* and the `replaceState` and `clone` methods. Line 5 suggests a new construct for binding the fields and methods of a *placeholder* to the target class, in this case *Account*. As opposed to indirect introduction, this *direct introduction* associates the call and execution join points of fields and methods with the target class. As a result, the use of an interface as an introduction container is no longer

necessary. However, in order to use polymorphic calls, an interface declaration for *Copyable* is still needed.

The *placeholder* concept may sound much like mixins [22], but it is fundamentally different. In mixins, the call and execution of a mixin method is delegated to the mixin class, not the target class, and hence the weak aspect-to-class binding problem can occur.

6.2 Reflection/Superclass Method Execution Dilemma

To guarantee mutual exclusion, *Shared* must intercept *every* method invocation on a shared object. *AspectJ* provides two pointcuts for intercepting the call and execution of a method: *call(MethodPattern)* and *execution(MethodPattern)*.

The *method call* pointcut can intercept non-reflective calls to *declared* and *inherited* methods of an object, but not reflective calls, i.e. calls using the Java reflection API. For instance, the pointcut *call(public * SavingAccount.*(..))* would intercept the method call `SavingAccount.debit(..)` but not `debit.invoke (SavingAccountObject, ..)` - a conscious design decision made by the *AspectJ* team not to "delve into the Java reflection library to implement call semantics" [24].

The *method execution* pointcut is typically used to address this deficiency. This pointcut can intercept the execution (both reflective and non-reflective) of declared and "overridden-inherited" methods of an object, but unfortunately not the execution (both reflective and non-reflective) of "non-overridden-inherited" methods, because in this case the execution join point occurs in the super class. For instance, the pointcut *execution(public * SavingAccount.*(..))* intercepts both the reflective and non-reflective execution of `SavingAccount.debit(..)`, but not `SavingAccount.getBalance()`, assuming that the `getBalance` method is defined in *Account* and not overridden in the child class *SavingAccount*.

Composing the call and execution pointcuts with an *or* operator is not a feasible solution either, because reflective invocations of `getBalance` can still not be intercepted.

One possible work-around is to require the application programmer to manually override all the inherited methods from a super class in the subclass, in which case the execution pointcut can be used to capture all calls. This solution is however undesirable: the code reuse benefits of inheritance are diminished, methods introduced by aspects can not be handled without introducing explicit dependencies of the base on the aspect, and there is always the danger that an application programmer forgets to override some of the methods.

Another work-around is to use a pointcut that explicitly names the super class: *target(SavingAccount) && execution(public * Account+.*(..))*. This pointcut intercepts the execution of the methods of an *Account* object when the target is *SavingAccount*. It intercepts both reflective and non-reflective executions of `SavingAccount.getBalance()` and `SavingAccount.debit(..)`. It also correctly excludes the execution of operations on other subclasses of account, e.g. *CheckingAccount*. Unfortunately this solution is application specific and cannot

be reused in a generic context. In order to write the pointcut, the exact superclass and target subclass have to be known.

The only fully generic and reusable solution for the aspect *Shared* would be to write: *target(IShared) && execution(public * *.*(..))*. This pointcut always works, but can result in a significant performance overhead as illustrated in section 7.3.

Language Improvement Suggestion: We propose the addition of an inheritance conscious method execution pointcut: *superexecution(MethodPattern)*. Given a class with no superclasses, this pointcut behaves exactly as the *execution(MethodPattern)* pointcut (i.e., it intercepts both reflective and non-reflective execution of declared methods). When used on a class with superclasses, it automatically overrides all non-overridden inherited methods, in our case `getBalance()`, within the body of the target class, in our case *SavingAccount*, with dummy methods that simply call the method in the superclass. It then applies the standard *execution(MethodPattern)* pointcut to the class. This ensures that the execution join points of non-overridden inherited methods occur in the target subclass, eliminating the reflection/superclass method execution dilemma problem.

6.3 Lack of Support for Explicit Inter-Aspect Configurability and Incompatibility

The aspects in AspectOPTIMA exhibit complex aspect dependencies and interferences. *AspectJ* has no construct that enables developers to express inter-aspect configurations. Ideally, an aspect should be able to express the need for functionality offered by other aspects, or adjust its functionality if interfering aspects are applied to the same joinpoint. Also, it should be possible to specify incompatible aspect configurations.

Our *AspectJ* implementation achieves rudimentary inter-aspect configurability by declaring dummy interfaces for each aspect. Aspects express the dependency on other aspects by having their associated interface implement the interfaces of the aspects they depend on using the *declare parents* construct (see, for example, line 2 of Fig. 11). However, this does not guarantee that the aspects are applied to the same join points.

Language Improvement Suggestion: We propose the addition of a new *declare dependencies* construct to *AspectJ*, which would allow inter-aspect configurability to be expressed as proposed in Fig. 14. The desired effect of this line of code is that *AccessClassified*, *Shared*, *AutoRecoverable* and *Tracked* should be applied to all the join points picked out by *LockBased*. However, general applications might require more fine-grained control over join points in case of complex aspect configurations. Ideally, an aspect should be able to selectively decide to what pointcuts each of the aspects it depends on is to be applied, and on the order in which the advice are to be executed.

declare dependencies:
LockBased **requires** AccessClassified, Shared, AutoRecoverable, Tracked;

Fig. 14. Proposed "declare dependencies" Construct

6.4 Lack of Support for Per-Object Aspects

In systems with many objects, such as in transactional systems, the ability to selectively apply different aspects to different objects of the same class is crucial. For instance, one might want to use pessimistic concurrency control for heavily used *Account* objects, and use optimistic concurrency control for less frequently used instances of the *Account* class. Unfortunately, *AspectJ* does not permit a developer to selectively decide to which instances of a class an aspect should be applied to.

However, the *if(BooleanExpression)* pointcut of *AspectJ* can be used to simulate per-object aspects. An aspect can introduce a field into the target class, and then test for specific values of that field in the pointcut. For example, an enumeration field *usage* could be introduced into the *Account* class, with possible values of *heavy* and *normal*. The *if* pointcut could inspect the value of the *usage* field to decide if an advice is to be applied to the object or not.

6.5 Lack of Support for Runtime Disabling and Enabling of Pointcuts

Aspects are statically deployed in *AspectJ*; i.e., the crosscutting behavior specified in an aspect is applied to the base application at weave-time and cannot be undone at runtime. This is a limitation for multi-version concurrency control strategies that store histories of old states of objects. After an object's state has been committed to history, it does not need to be *AutoRecoverable* and *Shared* anymore, since only *read* transactions are going to access the object's state in the future. To maximize system performance, it should be possible to disable the *AutoRecoverable* and *Shared* aspect for this object.

As shown in the implementation of *Shared*, the *if(BooleanExpression)* pointcut of *AspectJ* can be used to simulate runtime disabling and enabling of aspects, but results in a loss of performance (see section 7.2).

Language Improvement Suggestion: There are already *AspectJ*-like programming languages, e.g. JBossAOP, that support dynamic weaving of aspects as a whole. One could imagine an even more fine-grained language feature that would allow enabling and disabling of pointcuts. For instance, each *AspectJ* aspect could define two static methods `enablePointcut(PointcutPattern)` and `disablePointcut(PointcutPattern)` that would support runtime enabling and disabling of named pointcuts that match the pattern *PointcutPattern*. For instance, the call `Shared.aspectOf(obj).disablePointcut(methodExecution)`

	OO-read	OO-update	AO-read	AO-update
Time (seconds)	382.897	447.284	1871.734	1945.0231
Overhead (factor)	1	1	4.88	4.35

Table 1. Comparing Object-Oriented and Aspect-Oriented Performance

would disable the method execution interception specified by the *Shared* aspect for the object *obj* - eliminating/reducing the performance overhead.

7 Performance Analysis

We conducted several performance measurements on our implementation of AspectOptima in order to determine the performance of aspect-oriented frameworks, and the performance impact that the lack of support of the key language features presented in section 3 can have. All our experiments were run on a 3GHz Intel-based laptop with 512MB of RAM running Windows XP home edition, Eclipse 3.2.0, Java 1.5 and AspectJ 1.5.2. The measurements were obtained using the Eclipse Test and Performance Tools Platform [6].

All measurements execute operations on a simple bank account class that encapsulates a `balance` field and provides the methods `int getBalance()` and `deposit(int)`.

7.1 Aspect-Oriented Implementation vs. Object-Oriented Implementation

This subsection compares the performance of a purely object-oriented implementation of lock-based concurrency control with our aspect-oriented implementation *LockBased*. To perform the object-oriented measurements, we wrote a wrapper class for the bank account class that overrides `getBalance` and `deposit`, executing the same functionality as *LockBased* and the ten low-level aspects before forwarding the call to the actual account.

The performance measurements are given in Table 1. We performed 50,000 `getBalance` (read) operations, and 50,000 `deposit` (update) operations. The overall slowdown of the aspect-oriented implementation is around 1490 seconds, which represents 30 ms per operation, i.e., a slowdown for *read* operations by a factor of 4.88 and for *update* operations by a factor of 4.35.

The fact that the aspect-oriented implementation is slower is not surprising. Each of the low-level aspects is individually reusable and does not know about the specific context in which it is used. This independence makes it impossible to share runtime information among aspects. For instance, *LockBased* has to query the access kind of the method to be called from *AccessClassified*. But so does *AutoRecoverable*, *Tracked* and *Shared* (see Fig. 2). The object-oriented implementation however can optimize and call *AccessClassified* only once.

Although a certain slowdown due to reusability can not be avoided, it is foreseeable that the slowdown will become less significant thanks to advances

	not shared	shared & not enabled	shared	shared & enabled
Time (seconds)	0.586430	8.755212	54.023821	63.328594
Overhead (factor)	1	15	92	108

Table 2. Performance Overhead due to Lack of Dynamic Aspects

in compiler and weaving technology. For instance, *LockBased*, *AutoRecoverable*, *Tracked* and *Shared* all apply to the same joinpoint. An advanced weaver might be able to detect this situation and perform context-dependent optimizations.

7.2 Performance Impact of Simulating Per-Object Aspects

The need for per-object aspects and dynamic aspects, i.e. runtime disabling and re-enabling of aspects, is motivated by the multi-version concurrency control example. Once an object's state is committed, it is inserted into the history, and is subsequently only ever accessed by read-only transactions. Hence, the functionality provided by the *Shared* aspect is not needed anymore, since no transaction will ever modify that particular version of the object's state in the future. In *AspectJ* it is not possible to disable the pointcut defined in the *Shared* aspect at runtime. An *if(BooleanExpression)* pointcut modifier has to be used to simulate the disabling as shown in lines 3-5 and 9 of Fig. 9. Since the *AspectJ* rules forbid the use of non-static function calls within the boolean expression, an additional static version of the `getEnabled()` method that simply forwards the call to the target object had to be created.

To measure the performance overhead incurred, we performed three experiments, in which the read-only operation `getBalance` was called 1,000,000 times. The results of the experiment are presented in Table 2.

The first column shows the time spent inside `getBalance` for a standard bank account object. The third column shows the time spent inside `getBalance` in case the bank account object is shared. This includes the call to *AccessClassified* and the acquisition of the read lock. Obviously, the time spent in the method is considerably bigger – in our case by a factor of 92. The overhead of the `if` pointcut modifier is apparent in the second and the last column. They show the time it takes to check if the shared aspect is enabled for a particular bank account object. Our experiments show a slowdown of 8.2 seconds (a factor of 15!) when shared is disabled, and a slowdown of 9.3 seconds when it is enabled.

An aspect-oriented environment that supports dynamic aspects can therefore achieve significantly better performance. Of course, the actual activation / deactivation of aspects at runtime might also be costly. However, very often activation and deactivation are rare events, and their overhead can be safely ignored. In the case of multi-version concurrency control, the *Shared* aspect is deactivated once and for all when the object's state is inserted into the history of states.

	targeted read	targeted update	generic read	generic update
Time (seconds)	40.210723	29.900585	65.503984	52.767678
Overhead (factor)	1	1	1.63	1.76

Table 3. Comparing Application-Specific and Reusable Pointcuts

7.3 Performance Impact of Writing Reusable Pointcuts

The last experience we conducted aimed at evaluating the performance loss incurred in *AspectJ* due to having to work around the reflection/super class execution dilemma. In section 6.2 we described that with a targeted *call* pointcut we can not handle reflective calls, whereas with a targeted *execution* pointcut we can not handle executions of methods defined in the super class. The only way to achieve full functionality and reusability is to write a generic pointcut that intercepts *all* public method executions occurring in the application and dynamically check for the specific target at runtime.

To evaluate the performance loss we again ran 1,000,000 `getBalance` and `deposit` operations on a shared bank account object, once using the targeted, application-specific execution pointcut *target(SavingAccount) && execution(public * Account+.*(..)*, and once with the generic, reusable execution pointcut *target(IShared) && execution(public * *.*(..))*. The results are presented in Table 3.

The table shows that *read* operations are slower than *update* operations. This results from the fact that acquiring a read lock takes in general more time than acquiring a write lock.

The results also show that the slowdown resulting from a generic pointcut is not too significant: less than a factor of 2. This result must however be interpreted carefully. The performance loss measured here is the loss that is incurred due to the generic pointcut when calling a *Shared* object. But the generic pointcut will slow down *every public method execution* in the system, regardless of whether the object is shared or not, and therefore results in huge runtime overhead for an application with many calls to methods of non-shared objects.

8 Related Work

In [17], Mezini et al. identified several deficiencies of *AspectJ*'s join point interception model, namely:

- *Lack of support for sophisticated mappings*: the authors demonstrate with examples that the mapping from aspect abstractions to base classes via the *declare parents* construct is effective only when each aspect abstraction has a corresponding base class.
- *Lack of support for reusable aspect bindings*: the authors argue that the aspect-to-class binding achieved via the *declare parents* construct strongly binds an aspect to a particular base class; hence, such bindings cannot be effectively reused.

- *Lack of support for aspectual polymorphism*: this limitation is comparable to the lack of support for per-object association of aspects identified in this paper. The paper argued that it is not possible in *AspectJ* to determine at runtime whether an aspect should be applied or not, or which implementation of the aspect to apply.

The authors then proposed a new aspect-oriented programming tool called *CaesarJ* [1] to address these deficiencies. CaesarJ is based on *Aspect Collaboration Interfaces* (ACI). In ACIs, the aspect implementation is decoupled from the aspect binding in independent, indirectly connected modules. CaesarJ relies on a new type called a *weavelet* to compose the implementation and the binding of the aspect to form the final system. Different *weavelets* can combine an aspect binding with different aspect implementations, or a particular aspect implementation with different aspect bindings; making both the aspect bindings and implementations independently reusable. As opposed to *AspectJ*, compiling these *weavelets* with the base application does not have any effect on the execution of the application. This is because the *weavelets* must be explicitly deployed to activate their pointcuts and advice. The *weavelets* can be deployed statically or dynamically; hence, the support for runtime deployment of aspects on a per-object basis.

JBossAOP [3] is an aspect-oriented programming environment similar to *AspectJ*. It supports both per-object aspects and dynamic aspects, i.e., the ability to unregister existing advice bindings – pointcuts – and deploy new bindings at runtime. This dynamism is accomplished using the "*prepare*" statement of JBossAOP, which instruments target join points so that pointcuts and advice can later be applied at runtime. Initial experiments show that JBossAOP also suffers from the weak aspect-to-class binding problem. This is not surprising, since the implementation of reusable static crosscutting behavior in JBossAOP is achieved using mixins. Hence, the call and execution join points of introduced methods are associated with the mixin class, not the target class.

Cunha et al. [4] explore the possibility of implementing reusable aspects for concurrent programming in *AspectJ*. The authors illustrate how abstract pointcut interfaces and annotations can be used to implement one-way calls, synchronization barriers, reader/writer locks, schedulers, active objects and futures. The paper also compares the performance overhead and reusability of conventional object-oriented implementations with their own aspect-oriented implementations. The authors conclude that the *AspectJ* implementation is more reusable and pluggable, but incurs a noticeable performance overhead. The authors also highlight that *AspectJ* has a limitation in acquiring local join point information in concrete aspects: when a superaspect defines an abstract pointcut, the subaspects can not change the pointcut's signature.

The work of Cunha et al. differs from ours in the amount of effort required to reuse aspects in different contexts. In their case, developers must provide concrete pointcuts for each of the abstract pointcuts, which can be error-prone if not done correctly. Conversely, the *declare parents* construct used by our AspectOptima implementation to bind the aspects to application classes is safe: the correct pointcuts are hardcoded in the aspects.

Rashid et al. [19] have worked extensively on techniques which apply AOP concepts to database systems. In [20] the authors explore the possibility of aspectizing and implementing a reusable aspect-oriented framework for persistence in *AspectJ*.

The work of Fabry et al. [9,10] applies AOP concepts to advanced transaction models, e.g. nested and long running transactions. They proposed a general-purpose aspect language called KALA for modularizing the concerns of advanced transaction models into aspects.

9 Conclusions and Future Work

Based on the experience gained while implementing AspectOptima, we identified in this paper a set of eight important key features required for implementing reusable aspect-oriented frameworks with complex aspect dependencies and interactions: aspect packaging, separate aspect binding, inter-aspect configurability, inter-aspect incompatibility, inter-aspect ordering, per-object aspects, dynamic aspects and thread-aware aspects. We then showed how the implementation of AspectOptima in *AspectJ* revealed several limitations of the language, discussed possible work-around solutions, and suggested language improvements where appropriate. Finally we analyzed the performance overhead of aspect-oriented frameworks compared to a purely object-oriented implementation, and highlighted the impact that appropriate language support for reuse can have on performance.

We believe that studies such as this one are essential to discover key language features of aspect-oriented programming languages, and evolve aspect-oriented languages to even better modularize crosscutting concerns, improve maintainability and, most importantly, provide powerful and elegant ways of reusing crosscutting concerns.

We intend to extend our evaluation of AOP languages using AspectOptima in two areas. First, we intend to conduct a similar evaluation of CaesarJ and JBossAOP using AspectOptima. Once completed, we shall provide a comparative study of the language support provided by these three AOP languages necessary for implementing the key features identified by AspectOptima. Secondly, we intend to explore the feasibility of the new language features suggested in this work, and where possible implement them as an extension in the AspectBench Compiler[2].

10 Acknowledgments

The authors would like to thank Samuel Gélineau and Güven Bolükbasi, as well as the software engineering students and the members of the Sable research lab at McGill for their feedback on the implementation of AspectOptima. This research was partially supported by the Natural Sciences and Engineering Research Council of Canada.

References

1. Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of caesarJ. *Transactions on Aspect-Oriented Software Development*, 3880:135–173, 2006.
2. Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible aspectj compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2005. ACM Press.
3. Bill Burke, Austin Chau, Marc Fleury, Adrian Brock, Andy Godwin, and Harald Gliebe. JBoss aspect-oriented programming, February 2004.
4. Carlos A. Cunha, Joao L. Sobral, and Miguel P. Monteiro. Reusable aspect-oriented implementations of concurrency control patterns and mechanisms. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development - AOSD 2006, March 20 - 24, 2006*, pages 134 – 145. ACM Press, March 2006.
5. Ekwa Duala-Ekoko. Evaluating the Expressivity of AspectJ in Implementing a Reusable Framework for the ACID Properties of Transactional Objects - Master Thesis, School of Computer Science, McGill‘ University, August 2006.
6. Eclipse Development Team. Test and Performance Tools Platform. http://www.eclipse.org/tptp/, December 2006.
7. Tzilla Elrad, Mehmet Aksit, Gregor Kiczales, Karl Lieberherr, and Harold Ossher. Discussing aspects of aop. *Communications of the ACM*, 44(10):33–38, October 2001.
8. K. P. Eswaran, Jim Gray, R. A. Lorie, and I. L. Traiger. The notion of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624 – 633, November 1976.
9. Johan Fabry and Thomas Cleenewerck. Aspect-oriented domain specific languages for advanced transaction management. In *International Conference on Enterprise Information Systems 2005 (ICEIS 2005) proceedings*, pages 428–432. Springer-Verlag, 2005.
10. Johan Fabry and Theo D'Hondt. KALA: Kernel Aspect Language for Advanced Transactions. In *SAC '06: Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1615–1620. ACM Press, 2006.
11. James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. The Java Series. Addison Wesley, Reading, MA, USA, 1996.
12. Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, California, 1993.
13. Stefan Hanenberg and Rainer Unland. Parametric introductions. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development - AOSD'2003*.
14. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersen, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *15th European Conference on Object–Oriented Programming (ECOOP'2001)*, pages 327 – 357, June 18–22, 2001, Budapest, Hungary, 2001.
15. Jörg Kienzle and Samuel Gélineau. AO Challenge: Implementing the ACID Properties for Transactional Objects. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development - AOSD 2006, March 20 - 24, 2006*, pages 202 – 213. ACM Press, March 2006.

16. Jörg Kienzle and Rachid Guerraoui. AOP - Does It Make Sense? The Case of Concurrency and Failures. In *16th European Conference on Object–Oriented Programming (ECOOP'2002)*.
17. Mira Mezini and Klaus Ostermann. Conquering aspects with caesar. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development - AOSD'2003*.
18. Renaud Pawlak, Lionel Seinturier, Laurence Duchien, Gerard Florin, Fabrice Legond-Aubry, and Laurent Martelli. Jac: an aspect-based distributed dynamic framework. *Software Practice and Experience*, 34(12):1119–1148, 2004.
19. Awais Rashid. *Aspect-Oriented Database Systems*. Springer-Verlag, 2004.
20. Awais Rashid and Ruzanna Chitchyan. Persistence as an aspect. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development - AOSD'2003*, March 2003.
21. Dirk Riehle, Wolf Siberski, Dirk Bäumer, Daniel Megert, and Heinz Züllighoven. Serializer. In *Pattern Languages of Program Design 3*, pages 293–312. Addison-Wesley, 1998.
22. Arno Schmidmeier, Stefan Hanenberg, and Rainer Unland. Known concepts implemented in AspectJ. In Boris Bachmendo, Stefan Hanenberg, Stephan Herrmann, and Günter Kniesel, editors, *3rd Workshop on Aspect-Oriented Software Development (AOSD-GI) of the SIG Object-Oriented Software Development, German Informatics Society*, March 2003.
23. Sérgio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 174–190. ACM Press, 2002.
24. Xerox Corporation. Frequently Asked Questions about AspectJ. Available at http://www.eclipse.org/aspectj/doc/released/faq.html, May 2006.