AspectOptima: A Case Study on Aspect Dependencies and Interactions

Jörg Kienzle, Ekwa Duala-Ekoko, and Samuel Gélineau

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

Abstract. This paper presents AspectOptima, a language independent, aspect-oriented framework consisting of a set of ten base aspects—each one providing a well-defined reusable functionality—that can be configured to ensure the ACID properties (Atomicity, Consistency, Isolation, and Durability) for transactional objects. The overall goal of AspectOptima is to serve as a case study for aspect-oriented software development, particularly for evaluating the expressivity of AOP languages and how they address complex aspect interactions and dependencies. The ten base aspects of AspectOptima are simple, yet have complex dependencies and interactions among each other. To implement different concurrency control and recovery strategies, these aspects can be composed and assembled into different configurations; some aspects conflict with each other, others have to adapt their run time behavior according to the presence or absence of other aspects. The design of AspectOptima highlights the need for a set of key language features required for implementing reusable aspect-oriented frameworks. To illustrate the usefulness of AspectOptima as a means for evaluating programming language features, an implementation of AspectOptima in *AspectJ* is presented. The experiment reveals that *AspectJ*'s language features do not directly support implementation of reusable aspect-oriented frameworks with complex dependencies and interactions. The encountered *AspectJ* language limitations are discussed, workaround solutions are shown, potential language improvements are proposed where appropriate, and some preliminary measurements are presented that highlight the performance impact of certain language features.

Keywords: aspect dependencies, aspect collaboration, aspect interference, reusability, aspect-oriented language features, aspect binding, inter-aspect ordering, inter-aspect configurability, per-object aspects, dynamic aspects.

1 Introduction

Aspect orientation [1] has been accepted as a powerful technique for modularizing crosscutting concerns during software development in so-called *aspects*. Research

A. Rashid and H. Ossher (Eds.): Transactions on AOSD V, LNCS 5490, pp. 187–234, 2009.
© Springer-Verlag Berlin Heidelberg 2009

has shown that aspect-oriented programming (AOP) is successful in modularizing even very application-independent, general concerns such as distribution [2], concurrency [3, 4], persistency [2, 5], and failures [4].

However, the complexity of aspect-oriented software development increases exponentially when aspects are used in combination with each other. Dependencies between aspects raise the question of how to express aspect configurations, aspect interactions, conflicts among aspects, and aspect ordering. One way to find answers to these questions is to investigate non-trivial and realistic case studies, in which an application with many different concerns is implemented using many interacting aspects.

To this intent we designed AspectOptima [6], a language independent, aspect-oriented 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 ensure the ACID properties (Atomicity, Consistency, Isolation, and Durability) for transactional objects. The primary objective of AspectOptima is to provide the aspect-oriented research community with a language independent framework that can serve as a case study for evaluating the expressivity of AOP languages, the performance of AOP environments, and the suitability of aspect-oriented modeling notations, aspect-oriented testing techniques and aspect-oriented software development processes. The aspects that make up AspectOptima are simple, yet have complex dependencies and interactions among each other. The aspects can be composed and assembled into different configurations to achieve various concurrency control and recovery strategies. This composition is non-trivial; some aspects conflict with each other, others cannot function without the support of other aspects, and others have to adapt their run time behavior according to the presence or absence of other aspects.

We believe that studies such as this one are essential to discover key language features for dealing with aspect dependencies and interactions. The experience gained allows researchers to evolve aspect-oriented languages to even better modularize crosscutting concerns, reason about composition, and, most importantly, provide powerful and elegant ways of reuse. The second part of the paper demonstrates this by showing an implementation of AspectOptima in *AspectJ*. The goal of this effort was not to implement AspectOptima in an elegant way using the most appropriate AOP language. The idea was rather to show how the exercise of implementing AspectOptima can highlight the elegance or the lack of programming language features that can appropriately address aspect dependencies and interference in a reusable way. We have chosen to perform our implementation in *AspectJ* since it is currently one of the most popular and stable AOP languages, and *not* because of the features it provides.

The paper is structured as follows: Section 2 describes the context of the case study, namely transactional systems, the ACID properties of transactional objects, concurrency control, and recovery strategies. We present the design of AspectOptima in Sect. 3; it details the design of the 10 well-defined, reusable base aspects, and provides a description of three aspects that implement various concurrency control and recovery strategies by composing the base aspects in

different ways. This design highlights the need for a set of key language features required for implementing reusable aspect-oriented frameworks. In Sect. 4, we describe how we used AspectOptima to evaluate the expressiveness of the language features offered by the AOP language *AspectJ*. We present parts of our implementation to demonstrate that *AspectJ* provides sufficient (but certainly not ideal or elegant) support for dealing with mutually dependent and interfering aspects, discuss the encountered language limitations, suggest possible language improvements where appropriate, and present some preliminary performance measurements. Section 5 comments on related work and the last two sections present the conclusions and future work.

2 Background on Transactional Systems

We present the context of our case study in this section of the paper. The concepts of transactional systems, concurrency control, and recovery strategies relevant to this work are presented in Sects. 2.1, 2.2, and 2.3, respectively.

2.1 Transactional Objects

A *transaction* [7] groups together an arbitrary number of operations on *transactional objects*, which encapsulate application data, ensuring that the effects of the operations appear indivisible with respect to other concurrent transactions. The transaction scheme relies on three standard operations: *begin*, *commit*, and *abort*, which mark the boundaries of a transaction. After beginning a new transaction, all update operations on transactional objects are done on behalf of that transaction. At any time during the execution of the transaction it can *abort*, which means that the state of all accessed transactional objects is restored to the state at the beginning of the transaction (also called *rollback*). Once a transaction has completed successfully (is *committed*), the effects become permanent and visible to the outside.

Classic transaction models typically assume that each transaction is executed by a single thread of control. In recent years, however, advanced multithreaded transaction models have been proposed. Our case study implements the *Open Multithreaded Transaction* model [8, 9], which allows several *participants* (threads or processes) to enter the same transaction in order to perform a joint activity.

The ACID Properties. Transactions focus on preserving and guaranteeing important properties of application data, assuming that all the information is properly encapsulated inside transactional objects. These properties are referred to as the *ACID properties*: Atomicity, Consistency, Isolation, and Durability [7].

Atomicity ensures that from the outside of a transaction, the execution of the transaction appears to jump from the initial state to the result state, without any observable intermediate state. Alternatively, if the transaction cannot be completed for some reason, it appears as though it had never left the initial state.

This *all-or-nothing* property is unconditional, i.e., it holds whether the transaction, the entire application, the operating system, or any other components function normally, function abnormally, or crash.

The *consistency* property states that the application data always fulfill the validity constraints of the application. To achieve this property, transaction systems rely on the application programmer to write *consistency preserving* transactions. In this case, a transaction starts with a consistent state, and recreates that consistency after making its modifications, provided it runs to completion. The transaction system guarantees only that the execution of a transaction will not erroneously corrupt the application state.

The *isolation* property states that transactions that execute concurrently do not affect each other. In other words, no information is allowed to cross the boundary of a transaction until the transaction completes successfully.

Durability guarantees that the results of a committed transaction remain available in the future: the system must be able to re-establish a transaction's results after any type of subsequent failure.

Atomicity and isolation together result in transaction *serializability* [10], guaranteeing that any result produced by a concurrent execution of a set of transactions could have been produced by executing the same set of transactions serially, i.e., one after the other, in some arbitrary order.

When a participant calls a method on a transactional object, the underlying transaction support must take control and perform certain actions to ensure that the ACID properties can be guaranteed. Traditionally, this activity has been divided into concurrency control and recovery activities, which are described in the next two sections.

2.2 Concurrency Control

Concurrency control is that part of the transaction runtime that guarantees the isolation property for concurrently executing transactions, while preserving data consistency. In order to perform concurrency control, conflicting operations, i.e., operations that could jeopardize transaction serializability when executed by different transactions, have to be identified.

The simplest form of concurrency control among operations of a transactional object is *strict concurrency control*. It distinguishes *read*, *write*, and *update* operations. Reading a value from a data structure does not modify its contents, writing a value to the data structure does. Reading and subsequently writing a data structure is called updating. The conflict table of read, write, and update operations is shown in Table 1. There exist more advanced techniques of constructing a conflict table that take into account the semantics of operations [11], but they are out of the scope of this paper.

At run time, concurrency control is performed by associating a concurrency manager with each transactional object. The manager uses the conflict table to isolate transactions from each other. This can be done in a *pessimistic* (*conservative*) or *optimistic* (*aggressive*) way, both having advantages and disadvantages.

Table 1. Strict Concurrency Control Conflict Table

	read	***write/update***
read	No	Yes
write/update	Yes	Yes

Pessimistic Concurrency Control. The principle underlying *pessimistic concurrency control* schemes [7] is that, before attempting to perform an operation on any transactional object, a transaction has to get permission to do so from the concurrency control manager. The manager then checks if the operation would create a conflict with other uncommitted operations already executed on the object on behalf of other transactions. If so, the calling transaction is blocked or aborted.

Optimistic Concurrency Control. With *optimistic concurrency control* schemes [12], transactions are allowed to perform conflicting operations on objects without being blocked, but when they attempt to commit, the transactions are validated to ensure that they preserve serializability. If a transaction passes validation successfully, it means that it has not executed operations that conflict with operations of other concurrent transactions. It can then commit safely. A distinction can be made between optimistic concurrency control schemes based on the validation technique used in determining conflicts. *Forward validation* ensures that committing transactions do not invalidate the results of other transactions still in progress. *Backward validation* ensures that the result of a committing transaction has not been invalidated by recently committed transactions.

Concurrency Control and Versioning. In order to avoid rejecting operations that arrive out of order, several concurrency protocols have been proposed that maintain multiple *versions* (copies of the state) of objects [13–15]. For each update or write operation on an object, a new version of the object is produced. Read operations are performed on an appropriate, old version of the object, thereby minimizing the interactions between read-only transactions and write/update transactions. Versions are transparent to transactions: objects appear to them as only having a single state.

Concurrency Control and Multithreaded Transactions. Advanced transaction models such as Open Multithreaded Transactions allow several threads to perform work within the same transaction. These threads should not be isolated from each other. On the contrary, they should be allowed to see each other's state changes. However, concurrency control must still guarantee data consistency by ensuring that all modifications are performed in mutual exclusion.

2.3 Recovery

Recovery takes care of atomicity and durability of state changes made to transactional objects by transactions in spite of sophisticated caching techniques and

system failures. In other words, recovery must make sure that either all modifications made on behalf of a committing transaction are reflected in the state of the accessed transactional objects or none is, which means that any partial execution of state modifying operations has to be undone.

Recovery actions have to be taken in two situations: on *transaction abort* and in case of a *system failure*. Recovery strategies are either based on *undo* operations or *redo* operations, or both. Supporting both operations allows the cache management to be more flexible.

In order to be able to recover from a system failure, the recovery support must keep track of the status of all running transactions and of the modifications that the participants of a transaction have made to the state of transactional objects. Depending upon the meta-data available about the performed operations, whether a simple read/write classification or a more thorough semantic description, different kind of information will need to be kept. This so-called *transaction trace* must be stored in a log, i.e., on a stable storage device [16] that is not affected by system failures. Once the system restarts, the recovery support can consult the log and perform the cleanup actions necessary to restore the system to a consistent state.

In-place Update and Deferred Update. There exist two different strategies for performing updates and recovery for transactional objects, namely *in-place update* and *deferred update*.

When using *in-place update*, all operations are executed on *one main copy* of a transactional object. The effects of the operation invocation are therefore potentially visible to all following operation invocations, even those made on behalf of other transactions. In order to be able to undo changes in case of transaction abort, a backup copy, *snapshot* or *checkpoint* of an object is made before a transaction modifies the object's state.

When using *deferred update*, each modifying transaction executes operations on a *different copy* of the state of a transactional object. Therefore, until it commits, a transaction's changes are not visible to other transactions. When it does commit, the effects of its operations are applied to the original object either by explicitly copying the state or by reapplying the transaction's operations on the main copy.

2.4 Putting Things Together

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*: The concurrency control associated with the transactional object has to be notified of the method invocation to come. Pessimistic concurrency control schemes will use this opportunity to block or abort the calling transaction in case of conflicts.
2. *Recovery Prologue*: The recovery manager has to be notified in order to collect information for undoing the method call in case the transaction aborts later on.

3. *Method Execution*: The actual method call is executed.
4. *Recovery Epilogue*: The recovery manager has to be notified to gather redo information, if necessary.
5. *Concurrency Control Epilogue*: The concurrency control has to be notified that the method execution is now over.

Every transaction has to also remember all accessed transactional objects. When a transaction commits or aborts, it has to notify the concurrency control and recovery managers of each object of the transaction outcome.

3 The AspectOptima Framework

As shown in [4], transactions cannot be transparently added to an application. Yet, once the programmer has decided to use transactions in his application, and decided upon transaction boundaries, and determined the state that has to be encapsulated in transactional objects, it is possible to provide a middleware/framework/library that provides the run time support for transactions. Optima (OPen Transaction Integration for Multi-threaded Applications) [9, 17] is an object-oriented framework providing such support.

In this paper, we present the design of AspectOptima, a purely aspect-oriented framework ensuring the ACID properties for transactional objects. We present the design rationale of our framework in Sect. 3.1; Section 3.2 describes each of the ten base aspects of AspectOptima and Sect. 3.3 describes how these aspects can be combined to implement different concurrency control and recovery strategies for transactional objects.

3.1 Design Rationale

At a higher level, concurrency control and recovery can be seen as two completely separate concerns. As explained in the previous section, there are different ways of performing concurrency control and recovery, and, depending on the application, a developer might want to choose one technique over the other to maximize performance. Based on our experience of implementing the object-oriented Optima, we know that, at the *implementation* level, concurrency control and recovery cannot be separated completely from each other. There may be conflicts between the two, since not all combinations of concurrency control and recovery techniques mentioned before successfully provide the ACID properties. For example, most optimistic concurrency control techniques do not work with in-place update. There is also overlap between the two, since both techniques have to gather similar run time information in order to correctly perform concurrency control and recovery. For example, they both need to distinguish read from write/update operations.

Motivated by this incomplete separation of concerns, we applied aspect-oriented reasoning to decompose concurrency control and recovery further, and identified a set of ten aspects, each providing a specific sub-functionality. We

did not follow any particular aspect-oriented design technique to determine the functionality and scope of each aspect. Instead we relied on our object-oriented experience in implementing the ACID properties to identify and modularize reusable crosscutting functionality within aspects. Each of these aspects provides a well-defined common functionality, which, as it turns out, is often needed in non-transactional applications as well. In order to allow the aspects to be used in other contexts, they have been carefully designed to be reusable.

3.2 Ten Individually Reusable Aspects

This section describes each of the ten base aspects of AspectOptima. The motivation for each aspect is given, together with a brief summary of its functionality. Then the dependencies of the aspect are listed, i.e., what other base aspects the current aspect depends on, and to what other aspects the current aspect provides essential functionality. Situations of aspect interference, i.e., aspects that have to modify their behavior in order to continue to provide their functionality in the presence of other aspects, are pointed out. Finally, at the end of each aspect description, we sketch examples of how the aspect could be used in a stand-alone way to convince the reader of its reuse potential.

AccessClassified

Motivation. Concurrency control and recovery can be done more efficiently if the operations of transactional objects are classified according to how they affect the object's state: *read operations* (observers)—operations that do not modify the state of an object, *write operations,* and *update operations* (modifiers)—operations that read and write the state of an object.

The *AccessClassified* aspect provides this classification for methods of an object. Ideally, the classification of the operations should be done automatically. However, some implementations might require assistance from the developer. In such a case, the aspect should detect obvious misclassifications and output warnings.

Summary of Functionality of AccessClassified

- Every method of the object must be classified as either a *read*, *write*, or *update* operation. This functionality can, for instance, be provided by an operation `Kind getKind(Method m)`.

Dependencies of AccessClassified

- *Depends on*: –
- *Interferes with*: –
- *Is used by*: *Shared, Tracked, AutoRecoverable,* Concurrency Control, Recovery

Reusability of AccessClassified. The information provided by *AccessClassified* is very useful in many contexts. It could be exploited in multi-processor systems to implement intelligent caching strategies, or help to choose among different communication algorithms and replication strategies in distributed systems.

Named

Motivation. One of the fundamental properties of an object is its *identity*. Identity is the property that distinguishes an object from all other objects. It makes the object unique. At run time, a reference pointing to the memory location at which the state of an object is stored is often used to uniquely identify an object.

The lifetime of transactional objects, however, is not linked to the lifetime of an application, even less to a specific memory location. Since the state of a transactional object survives program termination, there must be a unique way of identifying a transactional object that remains valid during several executions of the same program. This identification means must allow the run time support to retrieve the object's state from a storage device or database. Also, depending on the chosen concurrency control and recovery techniques, there might exist multiple copies of the state of a transactional object in memory at a given time. These multiple copies, however, represent in fact one application object.

Previous work [18] has shown that an object *name* in the form of a string is an elegant solution for uniform object identification. We can therefore define a *Named* property or aspect for objects. A named object has a name associated with its identity. A name must be given to the object at creation time. The name remains valid throughout the entire lifetime of the object. At any time it should be possible to obtain the name of a given object, or retrieve the object based on its name.

Summary of Functionality of Named

- All creator operations must associate a unique name with the object that is created.
- `Object getObject(String s)` and `String getName(Object o)` operations map from an object to a name and vice versa.

Dependencies of Named

- *Depends on*: –
- *Interferes with*: –
- *Is used by*: *Tracked*, *Persistent*

Reusability of Named. Named can be used as a stand-alone aspect whenever the logical lifetime of an object does not coincide with the actual lifetime of its pointer into memory. For instance, one might like to destroy large, referenced but empirically unused objects and recompute them if needed. A name can also be used as a key to retrieve the state of an object from a database.

Shared

Motivation. Transactional objects are shared data structures. To conserve data consistency, it is important to prevent threads from concurrently modifying an object's state. Such a situation could arise when participants *of the same transaction* want to simultaneously access the same object.[1]

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.

Summary of Functionality of Shared

- Before executing the method body of a shared object, a read lock or write lock has to be acquired.
- When returning from the method, the previously acquired lock has to be released.

Dependencies of Shared

- *Depends on*: *AccessClassified*
- *Interferes with*: –
- *Is used by*: Concurrency Control, Recovery

In order to maximize throughput and hence optimize performance, *Shared* needs the semantic information of each method of the object provided by *AccessClassified*. If this information is not available, *Shared* must make worst case assumptions, i.e., assume that all methods of the object are in fact write or update operations.

All concurrency control schemes rely on *Shared* to provide consistency of data updates in spite of simultaneous accesses made by participants of the same transaction.

Reusability of Shared. Shared can be used as a stand-alone aspect in any concurrent application to guarantee data consistency of shared objects.

Copyable

Motivation. An object encapsulates state. The state of an object is initialized at creation time and can be altered by each method invocation. In a sense, the state of an object is its memory.

Sometimes it is necessary to duplicate an object, or replace an object's state with the state from another object. This functionality is offered by *Copyable*.

[1] The functionality offered by *Shared* is *not* to provide isolation among threads running in *different* transactions, but mutual exclusion among threads *of the same transaction*. This functionality is transaction specific and can be implemented in many different ways. We will show later how isolation can be achieved by combining several of the ten base aspects, such as demonstrated in *LockBased*, *MultiVersion*, and *Optimistic* (see Sect. 3.3).

Summary of Functionality of Copyable

- `Object clone()` creates an identical copy of the object.
- `replaceState(source)` copies the state of `source` over the state of the current object.

There is no obvious answer to the question whether to perform a *deep copy* or a *shallow copy* of the state of an object. When an object A stores in its state a reference to an object B, deep copy also clones/replaces the state of B when cloning/replacing the state of A. Recursively, if B refers to other objects, they are cloned/replaced as well. Shallow copy, on the other hand, only clones/replaces the state of A. Which technique is ideal depends on the application. Sometimes an application might even want to handle different classes/objects in different ways. A flexible implementation of *Copyable* should therefore provide a default technique, but allow the user to override the default behavior if needed.

Dependencies of Copyable

- *Depends on*: –
- *Interferes with*: *Shared*
- *Is used by*: *Serializable*, *Versioned*

Copyable should detect the presence of *Shared*. In a multi-threaded environment, the state of a shared object can only be copied when no other thread is modifying it.

Reusability of Copyable. *Copyable* is used whenever an object's state must be copied into another object of the same class. This situation arises whenever an object needs to be duplicated, e.g. for caching or replication. It is so often encountered that most programming languages provide the functionality of *Copyable* within the language (e.g. the Java `Object clone()` method that can be invoked on all objects that implement the `Cloneable` interface).

Serializable

Motivation. When an object is created, its state is in general stored in main memory. Whenever the object's state has to be moved to a different location, e.g. to a file, a database, or over the network to a different machine, the in-memory representation of the state of the object has to be transformed to suit the representation required by the destination location.

Serializable provides this functionality. A serializable object knows how to read its state from and write its state to backends requiring varying data representation formats. It is an incarnation of the *Serializer* pattern described in [19]. Just like with *Copyable*, serialization can be deep or shallow. Again, the ideal way of performing serialization is application-dependent. *Serializable* should therefore be customizable to specific application needs.

Summary of Functionality of Serializable

- `readFrom(reader)` restores the state of the object by reading it from a back end.
- `createFrom(reader)` creates a new object and initializes it with the state read from a back end.
- `writeTo(writer)` saves the state of the object to a back end.

Dependencies of Serializable

- *Depends on*: *Copyable*
- *Interferes with*: *Shared, Named*
- *Is used by*: *Persistent*

Serializable should detect the presence of *Shared*. In a multi-threaded environment, a shared object should only be serialized when no other thread is modifying it. *Serializable* should also detect the presence of *Named*, and serialize the name together with the object's state.

Reusability of Serializable. Serializable can be used in many situations, e.g., for writing an object's state to a file, sending the state over the network, or displaying an object's state. Serialization is so handy that modern programming languages usually provide a default serialization implementation for objects. The default serialization can usually be overridden with customized serialization, if needed.

Versioned

Motivation. During execution, each transaction must have its own view of the set of objects it accesses. Every thread participating in a transaction should see updates made by other participants, but not updates made from within other transactions. This is why multi-version concurrency control strategies, as well as snapshot-based recovery techniques, have to create multiple copies of the state of a transactional object in order to isolate state changes made by different transactions from each other.

This functionality is provided by *Versioned*. A *Versioned* object can encapsulate multiple copies—*versions*—of its state. Versions are linked to *views*, one of which is the default *main view*. If a *main view* is not explicitly designated, the original state of the object when it was created becomes the default view. A thread can subscribe to a view, and any method call made subsequently by the thread is directed to the associated version. A call coming from a thread that is not part of a specific view is forwarded to the main view.

Summary of Functionality of Versioned

- `Version newVersion()` creates a new version. The returned *Version* object is a "handle" to the newly created version.
- `deleteVersion(Version v)` deletes the version *v*.

- `View newView()`, `joinView(View v)`, `leaveView()`, and `deleteView(View v)` allow a thread to create, join, leave, or delete a view. Views are *nestable*, meaning that if a thread joins view A and then later creates a new view B, and then leaves view B, it should end up back in view A.
- `Version getCurrentVersion()` queries the current version assigned to the view of the calling thread.
- `setCurrentVersion(Version v)` assigns the version v to the view of the calling thread.
- `setMainView(View v)` designates a view to be considered the main one.
- Any method invocation on the transactional object are directed either to the version of a particular view, if the calling thread has joined a specific view, or else by default to the main view.

Dependencies of Versioned

- *Depends on*: *Copyable*
- *Interferes with*: –
- *Is used by*: *Recoverable,* Concurrency Control

Versioned requires the presence of *Copyable* in order to duplicate the object's state when a version is created. *Versioned* interferes with *Shared*: only when no other thread is currently modifying the object's state a new version can be created. Fortunately, *Copyable* should take care of this interference already, and therefore the interference between *Versioned* and *Shared* is only indirect.

Versioned is used by multi-version and optimistic concurrency control schemes, as well as by *Recoverable* for snapshot-based recovery. The optimistic concurrency control presented in Sect. 3.3, for instance, updates the main version of a *Versioned* object after successful validation of a committing transaction and deletes all non-main versions created by this transaction.

Reusability of Versioned. Versioned can be used as a stand-alone aspect in any application requiring transparent handling of multiple instances of an object's state. For instance, if an editor is extended to support different views for a single document, *Versioned* can be used to make different instances of the toolbar act on the correct view even though the toolbar source code refers to a single global variable.

Tracked

Motivation. To guarantee the ACID properties, the transaction runtime has to keep track of state access. The *Tracked* aspect provides the functionality to monitor object access in a generic way. It allows a thread to define a region in which object accesses are monitored. The region is delimited by *begin* and *end operations.* At any given time, the thread can obtain all read or modified objects for the current region.

Summary of Functionality of Tracked

- `TrackedZone beginTrackedZone()`, `joinTrackedZone(TrackedZone z)`, `leaveTrackedZone()`, and `endTrackedZone()` operations are provided that delimit the regions in which object access is to be monitored. Just like views, zones should be *nestable*. When a thread joins zone A, and then later zone B, and then leaves B, it should end up back in A.
- All accesses to a *Tracked* object from within a zone is monitored.
- `Object[] getAccessedObjects()`, `Object[] getReadObjects()`, and `Object[] getModifiedObjects()` operations that return the set of accessed, read, or modified objects of the current zone. Note that if nesting of zones is supported, then read or modified objects of a child zone have to be included when returning the set of read or modified objects of the parent.

Dependencies of Tracked

- *Depends on*: *AccessClassified, Named*
- *Interferes with*: *Versioned*
- *Is used by*: Concurrency Control, Recovery

In order to distinguish observer and modifier methods, *Tracked* depends on the presence of *AccessClassified*. Since it is not necessary to track accesses to different copies of the same application object, *Tracked* should detect the presence of *Versioned*. Using *Named* it is possible to compare the object names instead of object references to avoid duplicate counting.

Tracked is used by the transaction support run time to notify the concurrency controls of all accessed objects when a transaction commits or aborts. The recovery manager uses *Tracked* to identify all objects whose state has been modified and therefore must be made persistent (or rolled back in case of an abort). In this case, the tracked zone begins when the transaction begins, and ends when the transaction ends.

Reusability of Tracked. *Tracked* can be used in a stand-alone way to monitor object access made by arbitrary pieces of code, for instance, for the sake of logging and debugging. As another example, an implementation of the model-view-controller pattern [20] could use *Tracked* to maintain a dirty list of the parts of the model which changed since the view was last redrawn.

Recoverable

Motivation. The transaction runtime must be able to undo state changes made on behalf of a transaction when it aborts. *Recoverable* provides that functionality.

A recoverable object [21] is an object whose state can be saved and later on restored, if needed. Saving is sometimes also referred to as establishing a checkpoint; it is usually performed when the object is in a consistent state. Once saved, the state of the object can be restored at any time. It is possible to establish multiple checkpoints of the state of an object.

To implement checkpointing, this version of *Recoverable* takes a snapshot of the state of an object. In the future we might want to support logical recovery as well (see Sect. 7).

Recoverable should support both in-place and deferred update strategies (see Sect. 2.3). To support nested transactions it must be possible to establish multiple checkpoints for a single object. In case of in-place update, this creates a "sequence" of checkpoints, i.e., each checkpoint has at a given time at most one predecessor and one follower. In case of deferred update, a "tree" of checkpoints is created.

Summary of Functionality of Recoverable

- `Checkpoint establishCheckpoint()` creates a checkpoint. Depending on the chosen update strategy, *Recoverable* either makes sure that all views continue to point to the original copy of the object (in-place update) or creates a new version of the object and assigns it only to the view associated with the calling thread (deferred-update).
- `restoreCheckpoint(Checkpoint c)` restores the object's state to the state of a previously established checkpoint *c*. If no checkpoint is given as a parameter, then strict nesting of checkpoints is assumed, and the latest checkpoint is discarded.
- `discardCheckpoint(Checkpoint c)` discards a checkpoint *c*. If no checkpoint is given as a parameter, then strict nesting of checkpoints is assumed, and the latest checkpoint is discarded.
- `setDeferred(boolean On)` switches between in-place and deferred-update mode.

Dependencies of Recoverable

- *Depends on*: *Versioned*
- *Interferes with*: –
- *Is used by*: *AutoRecoverable,* Recovery

Recoverable depends on *Versioned* to provide snapshot-based checkpointing. It indirectly interferes with *Shared*: only when no other thread is currently modifying the object's state, a checkpoint should be established. Fortunately, *Versioned* should take care of this interference already.

Reusability of Recoverable. *Recoverable* can be used in any application that wants to be able to recover a previous state of an object. For instance, it can be used to implement a simple undo functionality.

AutoRecoverable

Motivation. In order to be able to rollback the state of an application when a transaction aborts, all accesses to transactional objects have to be monitored, and when an object is going to be modified for the first time from within the transaction, a checkpoint has to be established.

The *AutoRecoverable* aspect provides such region-based recovery. It allows a thread to define a region in which object access is monitored. The region is delimited by *begin* and *end operations*. Within a region, before any modifications are made to an object's state, a checkpoint is established automatically.

Summary of Functionality of AutoRecoverable

- `beginRecoverableZone()`, `joinRecoverableZone(RecoverableZone z)`, `leaveRecoverableZone()`, and `endRecoverableZone()` operations that delimit the regions in which object access is to be monitored for the current thread. Zones should be nestable.
- Whenever an auto-recoverable object is modified for the first time from within a zone, a checkpoint of the object has to be established.

Dependencies of AutoRecoverable

- *Depends on*: *Recoverable, AccessClassified*
- *Interferes with*: –
- *Is used by*: Recovery

AutoRecoverable depends on *Recoverable* to provide undo functionality for the object. It also depends on *AccessClassified* to determine if an operation is modifying the object's state or not.

Reusability of AutoRecoverable. *AutoRecoverable* can be used in any application that wants to be able to recover state changes made by arbitrary pieces of code. For instance, it can be used to undo the operations of a user-defined command manipulating an unknown subset of the application objects.

Persistent

Motivation. *Persistent* objects are objects whose state survives program termination. To achieve this, persistent objects know how to write their state to stable storage [16], i.e., a reliable secondary storage such as a mirrored hard disk or a database. Subsequently, it is possible to reinitialize the object's state based on the content of the storage device.

Summary of Functionality of Persistent

- All creator operations (constructors) of the object must associate a well-defined location on a storage device with the object.
- Operations to load/save the state of the object from/to the associated storage device.
- An operation to destroy a persistent object. When the object ceases to exist, the space on the associated storage device has to be freed as well.

Dependencies of Persistent

- *Depends on*: *Serializable*, *Copyable*, *Named*
- *Interferes with*: *Versioned, Recoverable*
- *Is used by*: Recovery

Persistent requires the presence of *Serializable* in order to transform the object's state into a flat stream of bytes. *Persistent* requires the presence of *Named*. It assumes that the object's name designates a valid location on a secondary storage device. *Persistent* should know how to handle the presence of *Versioned*, since, in general, there is a "main" version that contains the state of the object that is currently considered the correct one. *Persistent* should know how to handle the presence of *Recoverable*. When a recoverable object is made persistent, all checkpoints have to be made persistent as well. *Persistent* indirectly interferes with *Shared*. Only when no other thread is modifying the state of the object, *Persistent* should load or save the state of the object. Fortunately, *Serializable* should take care of this interference already.

Persistent is used by the recovery manager to write the old and new state of a transactional object to stable storage before the transaction commits. It is also used by the recovery manager to write information used to achieve tolerance to crash failures to the system log.

Reusability of Persistent. *Persistent* can be used in any application where an object's state has to survive program termination and hence has to be stored on some non-volatile storage device. To support many different storage devices, *Persistent* should be used in combination with a persistence framework such as [18].

Comments on Persistent. The *Persistent* aspect on its own only supports *explicit persistence*, i.e., the *Persistent* aspect has to be explicitly applied to every object that is to be made persistent. Also, loading and saving of the state of the object has to be done explicitly by invoking the corresponding method.

In a programming language providing *orthogonal persistence* [22], persistent data are created and used in the same way as non-persistent data. Loading and saving of values does not alter their semantics, and the process is transparent to the application program. Whether or not data should be made persistent is often determined using a technique called *persistence by reachability*. The persistence support designates an object as a persistent root and provides applications with a built-in function for locating it. Any object that is "reachable" from the persistent root, for instance by following pointers, is automatically made persistent. Providing orthogonal persistence and persistence by reachability is out of the scope of the AspectOptima framework.

Dependency and Interaction Summary. Figure 1 shows a UML class diagram that depicts the different relationships among the ten base aspects. Dependencies between aspects are shown on the left and interference between aspects on the right. The aspects that have to intercept calls to objects are stereotyped `<<i>>` (for *interceptors*). They all apply to the same joinpoints, i.e., they have to intercept *all* public method calls to the object they apply to. As mentioned before, the order in which they intercept is important.

The right-hand side of the diagram also shows two UML notes labeled *Concurrency Control* and *Recovery*. These show how the ten base aspects relate to

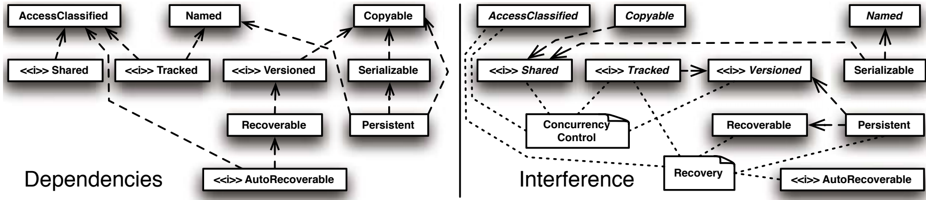

Fig. 1. Aspect Dependencies and Interferences

concurrency control and recovery strategies. The aspects that have italic names are those aspects that are used directly or indirectly by *both* concurrency control and recovery. They represent the implementation overlap between the two high-level concerns mentioned in Sect. 3.1.

3.3 Aspect Compositions

This section describes how the aforementioned base aspects can be combined to implement different concurrency control and recovery strategies for transactional objects. Each of the following aspects requires that all transactional objects are *AccessClassified*, *Named, Copyable*, *Serializable, Shared, Versioned, Tracked, Recoverable*, *AutoRecoverable*, and *Persistent.*[2] The aspects also assume that the transaction runtime creates a tracked zone, a recoverable zone, and a new view when a transaction begins, and ends the tracked zone, the recoverable zone, and the view when a transaction commits or aborts.

Pessimistic Lock-Based Concurrency Control with In-Place Update. This section describes the design of the *LockBased* aspect, which implements pessimistic lock-based concurrency control. Lock-based protocols use locks to implement permissions to perform operations. When a thread invokes an operation 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 transaction 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 [23] ensures serializability by not allowing transactions to acquire any lock after a lock has been released. This implies in practice that a

[2] The functionality provided by *Persistent* is not used in the examples shown in this section. Actually, persistency is mostly required at commit time of a transaction as shown in Sect. 4.3.

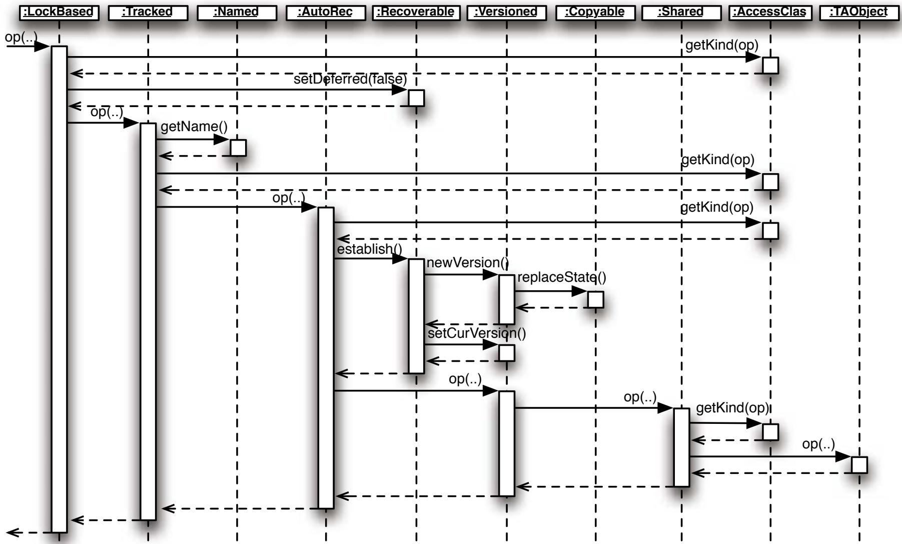

Fig. 2. Aspect Interactions for *LockBased* Objects

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. Figure 2 illustrates this interaction; the sequence diagram depicts how a call to a transactional object is intercepted, and how the individual aspects collaborate to provide the desired functionality.

Comments on LockBased. The design of LockBased is currently very simple. It does not support upgrading or downgrading of locks. Also, *LockBased* currently does not detect deadlock. Deadlock situations can happen with any blocking pessimistic concurrency control in case there are circular dependencies between transactions. Deadlock detection can therefore be seen as a crosscutting functionality, and could therefore be added as yet another base aspect to AspectOptima. Starvation is prevented in *LockBased* if the locks are granted in a strict FIFO ordering.

Pessimistic Multi-version Lock-Based Concurrency Control with In-Place Update. One drawback of standard lock-based concurrency control is that *read-only* transactions, i.e., transactions that only invoke observer methods on transactional objects, can be blocked by *update* transactions. This is especially annoying in applications where there are many short-lived read-only transactions, but only a few long-lived update transactions.

The *MultiVersion* aspect addresses this problem by implementing multi-version lock-based concurrency control with in-place update. *MultiVersion* relies on the fact that the transaction runtime knows how to classify transactions into *read-only* transactions and *update* transactions, i.e., transactions that write (and potentially read) the state of an object. *MultiVersion* also assumes that it is possible to assign timestamps with transactions.

MultiVersion keeps multiple versions of the state of a transactional object in memory—the "history" of *committed* states of an object, so to speak. Each version is annotated with a logical time interval during which that state was valid.

Update transactions are handled just as in *LockBased*. First, *MultiVersion* tries to acquire a write lock on the object. If no other transaction is currently modifying the object's state, then the lock is granted; otherwise, the calling thread is suspended. Once the lock is granted, *MultiVersion* relies on *Tracked* to record the access. If this is the first write performed on behalf of the transaction, *AutoRecoverable* checkpoints the object's state using in-place update, creating a new version. *Versioned* then directs the call to the new copy of the object, and finally *Shared* intercepts the call and makes sure that no two threads are modifying the object's state concurrently.

After the call has been executed, *Shared* releases the mutual exclusion lock. Future updates performed by the same transaction are automatically directed by *Versioned* to this version.

When an update transaction commits, *MultiVersion* assigns it a new logical timestamp, and adds the new committed state to the history of states, annotated with the new timestamp.

Read-only transactions are handled differently. They are assigned logical timestamps at *creation time*. They do not have to acquire any locks. *MultiVersion* nevertheless has to intercept the call and look at the transaction timestamp. It then finds the version with the highest timestamp that is lower than the transaction timestamp and assigns this version to the view of the transaction using *Versioned*. The call then proceeds to *Tracked*, where the read access is recorded. Then *Versioned* directs the call to the selected version. There is no need for *AutoRecoverable* or *Shared*, since only read requests are directed to this version.

Figure 3 illustrates the control flow through the aspects when a read or update operation is invoked on a transactional object. The versions *old1*, *old2*, *old3*, and *old4* represent previously committed states of the transactional object. Since they are only accessed by read operations, the *Shared* aspect is not needed anymore. To optimize performance, the *Shared* aspect should be removed from the main version as soon as an update transaction commits.

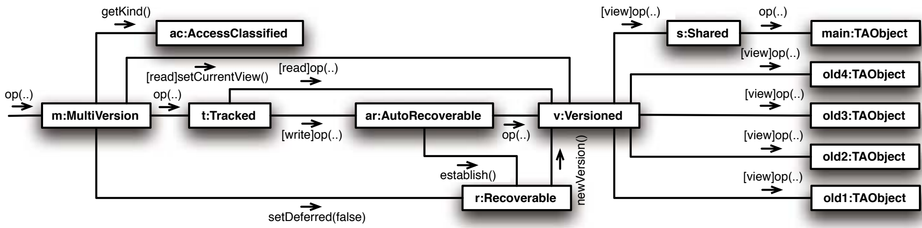

Fig. 3. Control Flow for Multi-Version Concurrency Control

Optimistic Concurrency Control with Deferred Update and Backward Validation. The aspect *Optimistic* implements optimistic concurrency control with deferred update and backward validation. When using optimistic concurrency control, the execution of each transaction is split into a *read phase*, a *validation phase*, and a *write phase*. When a transaction starts, it remembers the timestamp of the most recently committed transaction (*Tstart*). If and only if a transaction passes the validation phase, it receives a timestamp of its own and commits.

During the read phase, the transaction always reads the most recently committed states of transactional objects. *Optimistic* intercepts method calls to transactional objects and queries *AccessClassified* to classify the call.

In case of a read, *Optimistic* passes the call along to *Tracked* to record the read access. *Versioned* forwards the call to the current main version that contains the most recently committed state. There is no need for *AutoRecoverable* to do any work, nor is the presence of *Shared* required, since the call is read-only.

In case of a write or update operation,[3] *Optimistic* makes sure that deferred update is selected by calling *Recoverable*, and then passes the call to *Tracked*. Next, *AutoRecoverable* creates a new version of the transactional object, but this time using deferred update. This ensures that subsequent reads made by other transactions are still forwarded to the most recently committed version. *Versioned* forwards the call to the newly created version, and *Shared* takes care of ensuring mutual exclusion.

In case of a concurrent write made by a different transaction, *AutoRecoverable* creates yet another version. Therefore, at a given time, there might exist *multiple uncommitted versions* of a transactional object, each one belonging to a different transaction.

The UML 2.0 communication diagram shown in Fig. 4 illustrates the control flow through the aspects for read and update operations. In the depicted situation, there are currently four active update transactions, each one having its own local version of the transactional object's state. Read access to the main version does not flow through *AutoRecoverable* or *Shared*.

In optimistic concurrency control schemes, a transaction has to pass *validation* before it can commit. During validation, *Optimistic* looks at the timestamp of the most recently committed transaction (*Tend*). *Optimistic* then calculates the

[3] Note that we are still in the read phase of the transaction!

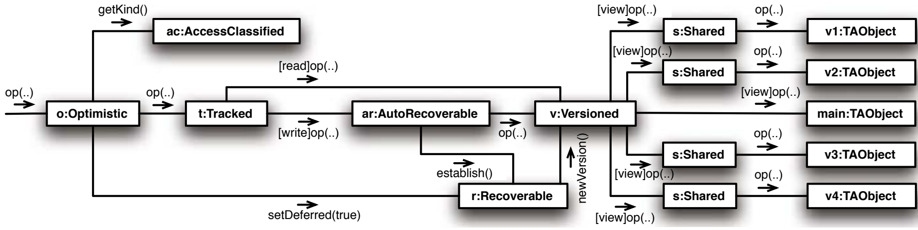

Fig. 4. Control Flow for Optimistic Concurrency Control

union of all transactional objects updated by transactions *Tstart+1* to *Tend* using *Tracked* and intersects it with the set of objects read by the validating transaction (backward validation).

If the intersection is non-empty, validation fails and the transaction has to abort. *Optimistic* tells *Recoverable* to restore the checkpoints of all modified objects, which results in deleting the local versions of the transaction.

If the intersection is empty, validation is successful. The transaction receives a timestamp and proceeds to the write phase, in which *Optimistic* tells *Recoverable* to discard the checkpoints of all modified objects. This results in committing the local versions of the transaction and discarding the previous one.

3.4 Summary

Transaction systems in general implement the ACID properties by performing concurrency control and recovery, each of which can be done using different techniques. This separation of concerns is, however, not very clean. Concurrency control and recovery can benefit from sharing parts of their implementation, and certain combinations of concurrency control techniques conflict with certain recovery techniques.

AspectOptima shows how aspect-oriented techniques can help to decompose the implementation of the ACID properties into a set of fine-grained aspects. The decomposition exhibits the following properties:

- *Clear Separation of Concerns:* Each aspect provides a well-defined functionality. For example, *Shared* takes care of ensuring mutual exclusion of state updates.
- *High Reusability:* Each aspect can be used in other applications in a stand-alone way to implement similar functionalities. For example, *Recoverable* can be used to implement "undo" functionality.
- *Complex Aspect Dependencies:* Some aspects cannot function properly without the functionality offered by other aspects. For example, *Persistent* depends on the presence of *Named*. It uses the object's name to designate a valid location on a secondary storage device.
- *Complex Aspect Interference:* Some aspects have to adapt their functionality in the presence of other aspects. For example, *Copyable* has to detect the presence of *Shared*, and make sure that it only makes a copy of an object when no other thread is modifying the object's state.

4 Evaluating the Expressiveness of AO Languages with AspectOptima

In this section, we demonstrate how AspectOptima can be used to evaluate the expressiveness of the language features of an AOP language, particularly those features concerned with aspect reuse, aspect dependencies and interference. The language under study in our case is *AspectJ* [24]. The goal of this effort was not to implement AspectOptima in an elegant way using the most appropriate AOP language. The idea was rather to show how the exercise of implementing AspectOptima can highlight the elegance or the lack of programming language features that can appropriately address aspect dependencies and interference in a reusable way. We have chosen to perform our implementation in *AspectJ* since it is currently one of the most used languages with a mature and reliable compiler, and *not* because of the features it provides.

This section is structured as follows. We outline the language requirements necessary for implementing AspectOptima in Sect. 4.1; a brief overview of the *AspectJ* is presented in Sect. 4.2. We present an *AspectJ* implementation of AspectOptima in Sect. 4.3, discuss the encountered language limitations of *AspectJ*, present some suggestions for language improvements in Sect. 4.4, and finally show some preliminary measurements that highlight the performance impact of certain language features in Sect. 4.5.

4.1 AspectOptima Implementation Language Requirements

One of the goals of AspectOptima is to define a framework that can be used to evaluate and compare the expressiveness of language features of AOP languages, particularly with respect to how they deal with complex aspect dependencies and interactions in a reusable way. The key questions an implementation has to address are:

- Can each of the aspects be implemented in a *modular*, stand-alone way? To be considered modular, the implementation of an aspect should be packaged in such a way that the package contains all the code needed to implement the functionality. This simplifies adding and removing of an aspect for application developers, and improves readability and maintainability for aspect developers.
- Can each of the aspects be implemented in a *reusable* way? To be considered reusable, a developer that needs a functionality offered by one of the aspects in his application should be able to integrate the functionality with minimal effort into his implementation by simply deploying the aspect.
- Does the AOP language allow to specify different aspect configurations and compositions of the ten base aspects? Can *different* combinations be used within the *same* application? In aspect frameworks it is likely that different aspect combinations are possible, and could be useful at different places in the application.

- Is aspect configuration *safe*? Aspect deployment should not be error-prone, i.e., it should not happen that an application developer can make mistakes when deploying the aspect in his application.
- Can *dependencies* between aspects be handled in a *transparent* way? To be considered transparent, 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.
- Can *interferences* between aspects be dealt with in such a way that the aspect implementations are still individually reusable, i.e., there are no direct dependencies among the aspect implementations due to aspect interference? When two aspects interfere and additional behavior is required to address the interference, can this be dealt with in a transparent way without bothering the application developer at configuration time?

Based on our experience, the following list summarizes what features an AOP language and environment has to offer to make the implementation of AspectOptima possible:

- *Separate Aspect Binding:* In order to support reusability, reusable aspect implementations should not contain explicit bindings to application elements. An application developer has to be able to specify where an aspect is to be applied when he composes his application.
- *Inter-Aspect Configurability:* Some aspects have to be able to express their dependence on other aspects. For example, *Versioned* can only be applied to objects that are also *Copyable*.
- *Inter-Aspect Ordering:* Some aspects need to specify the order in which other aspects get applied. For example, the aspect *LockBased* has to make sure that *Tracked* records the object access only after a lock has been acquired.
- *Per-Object (Per-Instance) Aspects:* An application programmer might want to use different concurrency control or recovery implementations for different objects of the same class. It should therefore be possible to associate *LockBased*, *MultiVersion*, and *Optimistic* to *objects*, not to classes. As a consequence, *LockBased*, *MultiVersion*, and *Optimistic* have to be able to apply the aspects they depend on to their *object*, not to the class.
- *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 run time. In AspectOptima, for example, in multiversion 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.

Most of these requirements have been mentioned in the aspect-oriented literature before (the interested reader is referred to the proceedings of the SPLAT (Software Engineering Properties of Languages and Aspect Technologies) [25] and FOAL (Foundations of Aspect-Oriented Languages) [26] workshops). The main contribution here is that AspectOptima requires *all* of the features in order to be implemented in an elegant reusable way.

4.2 AspectJ

We decided to validate the design of AspectOptima and test its effectiveness in evaluating AOP language features by an implementation in *AspectJ* [24]. *AspectJ* is an aspect-oriented extension of the Java [27] programming language. It was conceived by a team of researchers at Xerox Parc and is probably currently the most popular AOP language. The version used for our experiments is version 1.5.2.

In *AspectJ*, crosscutting behavior is encapsulated 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. Four new concepts introduced in *AspectJ* are relevant to this work: *joinpoints*, *pointcuts, advice*, and *inter-type declarations.*

Joinpoints are well-defined points in the execution of a program. These include method and constructor calls, their executions, field accesses, object and class initializations, and others. Only call and execution joinpoints were essential in our current implementation of AspectOptima.

A *pointcut* is a construct used to designate a set of joinpoints 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(..)*).

An *advice* defines the actions to be taken at the joinpoint(s) captured by a pointcut. It consists of standard Java code. *AspectJ* supports three types of advice: the *before*, the *after*, and the *around* advice. The *before* advice runs just before the captured joinpoint; the *after* advice runs immediately after the captured joinpoint; the *around* advice surrounds the captured joinpoint 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.

The following paragraphs of this subsection present techniques and workarounds that can be used in *AspectJ* to achieve some of the requirements presented in Sect. 4.1. It should be noted here that we did not choose *AspectJ* because we expected it to be the ideal language for implementing AspectOptima. To the contrary, there exist many other AOP languages that provide more advanced features and hence are probably more suitable. *CaesarJ* [28], for instance, defines *Aspect Collaboration Interfaces*, which among many other benefits nicely decouple aspect implementations from aspect bindings. Initial experiments with *CaesarJ* however showed that the current compiler is not stable enough to build a complex aspect framework such as AspectOptima.

Separate Aspect Binding and Inter-aspect Configurability. In *AspectJ*, the abstract introduction idiom (also known as indirect introduction) [29, 30] can be used to achieve 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" [29]. In order words, the target classes of the static and dynamic crosscutting behavior are unknown until weave-time. This strategy has three participants (see Fig. 5):

- *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.

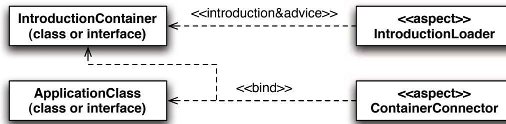

Fig. 5. Abstract Introduction Idiom

The introduction container serves a dual purpose in the context of our implementation. First, it enables the aspects (i.e., both static and dynamic crosscutting behaviors) to be reused in different contexts; second, it helps in identifying the classes to which the crosscutting behaviors 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 cannot 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).

Since all AspectOptima aspects declare dummy interfaces, separate aspect binding can be achieved using the *declare parents* construct of *AspectJ*. The first aspect in Fig. 6 brands the *Account* class as *IShared*; hence, the crosscutting behavior of the *Shared* aspect is applied to all instances of the *Account* class.

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. 6. Hence, an *AutoRecoverable* object is by default *Recoverable* and *AccessClassified*. This technique to achieve separate aspect binding and inter-aspect configurability

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

Fig. 6. Separate Aspect Binding and Inter-aspect Configurability in *AspectJ*

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

Fig. 7. Inter-aspect Ordering in *AspectJ*

makes reuse very easy. Application developers do not have to modify their base classes to apply aspects to them.

Inter-Aspect Ordering. Inter-aspect ordering is supported in *AspectJ* by the *declare precedence* construct. Figure 7 illustrates how the *LockBased* aspect specifies its execution order relative to that of the aspects it depends on.

AspectJ precedence declarations are application-wide. It is hence not possible to declare, for instance, two different orderings of the same set of aspects for two different pointcuts. In the current version of AspectOptima, however, such a functionality is not necessary since *MultiVersion* and *Optimistic* depend on the exact same ordering as *LockBased*. However, it is not guaranteed that this would also be the case if the ten base aspects are reused within other applications.

Per-Object Aspects, Dynamic Aspects, and Per-Thread Aspects. *AspectJ* does not support per-object aspects or dynamic weaving. run time enabling and disabling of aspects (i.e., advice within an 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 Sect. 4.3 for example code using this technique).

Per-thread aspects can be simulated by using the `ThreadLocal` class provided by the Java standard library. Using `ThreadLocal`, it is possible to associate an object with each thread instance. Within this object, boolean attributes can be stored that can be consulted by the advice of an aspect in order to determine if the aspect is enabled for the currently executing thread or not.

4.3 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 [31] for a complete description of the implementation.

```
@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. 8. *AspectJ* implementation of *AccessClassified*

AccessClassified Implementation. In our *AspectJ* environment (based on the `ajc` compiler), it is not possible to statically determine if a method potentially reads, writes, or updates the fields of an object. Therefore, our implementation of *AccessClassified* (see Sect. 3.2) relies on the application developer to tag every method of an object with marker annotations, such as the *Read* annotation defined in the top lines of Fig. 8. The annotations have a run time retention policy (i.e., they are retained by the virtual machine so that they can be read reflectively at run time), can be inherited (i.e., annotations on superclasses are automatically inherited by subclasses), and have to be applied to methods.

The *AccessClassified* implementation aspect shown in Fig. 8 introduces a method (`getKind(String methodName)`) to every *IAccessClassified* object that examines these annotations by reflection at run time and classifies each operation accordingly. Non-annotated methods are treated as modifier operations to guarantee system consistency. For an example of how a developer can classify the operations of a class, see Sect. 4.3.

Another interesting possibility is to determine the access kind automatically at run time by tentatively executing the method and by intercepting all field modifications (see [31] for details). It might also be possible to extend the flexible *AspectJ* compiler `abc` [32] to perform an automatic classification based on static code analysis.

Copyable Implementation. The *Copyable* aspect (Fig. 9) introduces state replacement and cloning functionality to all classes that implement the *ICopyable* interface. Java already provides a default `clone()` method for objects that implement the *Cloneable* interface. However, this default method only implements shallow cloning. To provide deep cloning, some additional work is needed. The `replaceState(Object src)` method enables an object to swap its state with

```
public aspect Copyable {
  declare parents: ICopyable implements Cloneable;
  public void ICopyable.replaceState(Object src) {
    try {
      copyFields(this,src);
    } catch (SourceClassNotEqualDestinationClass e) {}
  }
  public Object ICopyable.clone() {
    Object deepCopyOfOriginalObject = super.clone();
    deepCopyOfOriginalObject.replaceState(this);
    return deepCopyOfOriginalObject; } }
```

Fig. 9. *AspectJ* implementation of *Copyable*

that of another object of the same class. The `copyFields(this, src)` helper method performs a field-by-field deep-copy of the state (inherited, declared, and introduced) of the source object to the invoking object (code not shown here for space reasons).

Shared Implementation. The implementation of the *Shared* aspect is presented in Fig. 10. This aspect depends on the method classification provided by the *AccessClassified* aspect to determine the appropriate lock to be acquired for a given operation. The *declare parents* construct used in line 2 illustrates inter-aspect configuration by declaring all *Shared* objects to be *AccessClassified* as well.

Lines 3–5 define a boolean field and two methods for supporting run time 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 and 7 allocate a lock for each *Shared* object. The pointcut on line 8 makes sure that all public or protected method executions of 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. After the operation is executed, the lock is released again.

Tracked Implementation. Figure 11 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 11–14).[4]

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

```
 1 public aspect Shared {
 2   declare parents: IShared implements IAccessClassified;
     //Introduce variable and methods for run time disabling/re-enabling of advice
 3   private boolean IShared.Enabled = true;
 4   private boolean IShared.getEnabled() { return Enabled; }
 5   static boolean isEnabled(IShared object) { return object.getEnabled(); }
     //Introduce the variable and method for enforcing synchronization
 6   private Lock IShared.threadLock = new Lock();
 7   private Lock IShared.getSharedLock() { return threadLock; }
 8   pointcut methodExecution(IShared ishared): target(ishared) &&
      (execution(public * IShared+.*(..))||execution(protected * IShared+.*(..)));
 9   Object around(IShared shared) : methodExecution(shared) &&
       if(isEnabled(shared)) {
10     Object obj;
11     Kind accessType = shared.getKind(getMethodName(thisJoinPoint));
12     // Get the appropriate lock
13     if (accessType == READ) shared.getSharedLock().getReadLock();
14     else if (accessType == WRITE) shared.getSharedLock().getWriteLock();
15     else shared.getSharedLock().getUpdateLock();
16     obj = proceed(shared);
   // Release previously acquired lock
...
20     return obj; } }
```

Fig. 10. *AspectJ* Implementation of *Shared*

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 8 calls `getName`, a functionality provided by *Named*, to obtain the object's identity. If the object has not been associated with the zone yet, the advice records the access according to its category (lines 9 and 10). Finally, lines 15–20 implement operations that provide the set of objects read or modified from within a zone.

LockBased Implementation. The *LockBased* aspect provides support for pessimistic lock-based concurrency control with in-place update (Fig. 12). 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

```
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    Zone z = (Zone)myZone.get();
8    String myName = ((ITracked)thisJoinPoint.getTarget()).getName();
     //Place the target object in the appropriate category
9    if (type == READ && !z.readObjects().containsKey(myName))
       z.addReadObject(thisJoinPoint.getTarget());
10   else if (!z.writeObjects().containsKey(myName))
       z.addWriteObject(thisJoinPoint.getTarget());
   }
11 public static synchronized void beginTrackedZone(){
12   if (myZone.get() == null) myZone.set(new Zone());
   }
13 public static synchronized void endTrackedZone() {
14   myZone.set(null);
   }
15 public static Vector getReadObjects() throws NoZoneFoundException{
16   if (myZone.get() == null) throw new NoZoneFoundException("..");
17   return ((Zone)myZone.get()).getReadObjects();
   }
18 public static Vector getModifiedObjects() throws NoZoneFoundException{
19   if (myZone.get() == null) throw new NoZoneFoundException("..");
20   return ((Zone)myZone.get()).getModifiedObjects(); } }
```

Fig. 11. Implementation of *Tracked*

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 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

```
 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();
19      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. 12. *AspectJ* Implementation of *LockBased*

sets the update strategy by using functionality provided by *Recoverable* (which is configured to apply to the target object by *AutoRecoverable*).

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.

```
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. 13. A Lockbased Account

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.

Using AspectOptima. Line 1 of Fig. 13 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* and *update* operations respectively using the marker annotations of *AccessClassified* in lines 3 and 4.

4.4 Encountered AspectJ Limitations and Possible Improvements

We provide a discussion of the encountered *AspectJ* limitations, possible workaround solutions, and suggestions for improvements to the *AspectJ* language features in this section.

Weak Aspect-to-Class Binding. An object in an *AspectJ* environment could have three types of methods: those inherited from superclasses and superinterfaces, those declared by the class, and those introduced by aspects through direct or indirect introductions. As explained in Sect. 4.2, our implementation achieves aspect reusability, separate aspect binding, and inter-aspect configurability by using the abstract introduction idiom [29, 30]. Extrinsic static crosscutting behaviors are collected in dummy interfaces (via the inter-type member introduction) 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(SrcObj)* and *clone()* into every *Account* object.

As described in Sect. 3.2, *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(SrcObj)` will be captured by the pointcuts *call(public * IShared+.*(..))* and *execution(public * IShared+.*(..))* of the *Shared* aspect, since the method *replaceState(SrcObj)* is defined for the *Account* class. This is not the case; the actual call and execution joinpoints are `call(ICopyable.replaceState(..))` and `execution(ICopyable.replaceState(..))`, respectively; i.e, *AspectJ* associates the call and the execution joinpoints of indirectly introduced methods with the *introduction container* not the application class. As a result, *Shared* does not intercept

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

Fig. 14. Proposed *"placeholder"* Construct

calls to *replaceState(SrcObj)*, 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 cannot 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.

Figure 14 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 joinpoints 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 [33], 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.

Reflection/Superclass Method Execution Dilemma. The enforcement of the ACID properties of transactional objects occurs at the level of method invocations. To achieve this, the AspectOPTIMA aspects, for instance *Shared*, must rigorously intercept *every* method invocation on a transactional object to perform the appropriate pre- and post-actions before allowing the call to proceed.

AspectJ provides two pointcut designators 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" [34].

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 joinpoint 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 cannot 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, since a dynamic check has to be performed at *every* public method execution of any class.

Language Improvement Suggestion: We propose the addition of an inheritance-conscious method execution pointcut to *AspectJ*: *superexecution(MethodPattern)*. Given a class with no superclasses, this pointcut behaves exactly as the *execution(MethodPattern)* pointcut (i.e., it intercepts both

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

Fig. 15. Proposed "declare dependencies" Construct

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 joinpoints of non-overridden inherited methods occur in the target subclass, eliminating the reflection/superclass method execution dilemma problem.

Lack of Support for Explicit Inter-Aspect Configurability. 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 of functionality offered by other aspects, or adjust its functionality if interfering aspects are applied to the same pointcuts. 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. 12). However, this does not guarantee that the aspects are applied to the same joinpoints.

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. 15. The desired effect of this line of code is that *AccessClassified*, *Shared*, *AutoRecoverable*, and *Tracked* should be applied to all the joinpoints picked out by *LockBased*. However, general applications might require more fine-grained control over joinpoints 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.

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.

Lack of Support for Run time Disabling and Enabling of Pointcuts. Aspects are statically deployed in *AspectJ*; i.e., the crosscutting behaviors specified in the aspects become effective in the base applications once they are woven together and these crosscutting behaviors cannot be altered at run time. This limitation is encountered, for instance, in multi-version concurrency control (see Sect. 3.3). 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 the *Shared* aspect in Sect. 4.3, the *if(BooleanExpression)* pointcut of *AspectJ* can be used to simulate run time disabling and enabling of aspects. Unfortunately, this only disables the advice associated with a joinpoint. This implies that the target-joinpoint will always be intercepted but the execution of its associated advice is conditional on the value of *BooleanExpression*—resulting in performance loss, since operations of read-only transactions are still unnecessarily intercepted, and a run time check has to be performed on every operation invocation. Reference [35] reports that the *if(BooleanExpression)* pointcut (where *BooleanExpression* is a single static method call) introduces a 22% performance overhead.

Language Improvement Suggestion: Some AOP languages, e.g. JBossAOP [36], already support dynamic weaving of aspects as a whole. One could imagine an even more fine-grained feature that would allow enabling and disabling of pointcuts. For instance, aspects could define static methods `enablePointcut(Pattern)` and `disablePointcut(Pattern)` that would support run time disabling and reenabling of named pointcuts whose name matches *Pattern*. For instance, the call `Shared.aspectOf(obj).disablePointcut(methodExecution)` would disable the method execution interception specified by the *Shared* aspect for the object *obj*, eliminating/reducing the performance overhead.

4.5 Initial Performance Evaluation

We conducted several preliminary performance measurements on our implementation of AspectOptima in order to determine the performance of the aspect-oriented framework, and the performance impact that the lack of support of some of the key language features presented in Sect. 4.1 can have. The measurements are preliminary in the sense that far more measurements would be needed in order to accurately determine the performance impact of aspect-oriented frameworks and language features. In order to compare the performance of different aspect-oriented execution environments, thorough benchmarks should be defined. This is, however, out of the scope of the paper and left for future work.

Table 2. Comparing Object-Oriented and Aspect-Oriented Performance

	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

All our experiments were run on a 3 GHz Intel-based laptop with 512 MB 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 [37]. All measurements execute operations on a simple bank account class that encapsulates a `balance` field and provides the methods `int getBalance()` and `deposit(int)`.

Aspect-Oriented Implementation vs. Object-Oriented Implementation. This section 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`, and then calls OPTIMA [9, 17] (the object-oriented version of our framework) to execute 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 2. We performed 50,000 `getBalance` (read) operations and 50,000 `deposit` (update) operations. The overall slowdown of the aspect-oriented implementation is around 1490 s for both read and update operations, which represents 30 ms per operation.

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 run time 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: it calls *AccessClassified* only once, and then passes the access kind as a parameter to the different components implementing 2-phase locking, recovery, and mutual exclusion.

This is of course not a problem of aspect-orientation as such, but rather a problem of separation of concerns in general. Since each aspect should be individually reusable, it cannot depend on other aspects to classify the operation. It is foreseeable, however, that this slowdown in the future will become less significant thanks to advances in compiler and weaving technology. For instance, *LockBased*, *AutoRecoverable*, *Tracked* and *Shared* all apply to the same joinpoint. An advanced weaver, such as found in the `abc` [32] compiler or the *Steamloom* environment [38], might be able to detect this situation and perform context-dependent optimizations. To make this possible, the compiler would have to detect that the result returned by `getKind(String methodName)` of *AccessClassified* is constant for a given method name. It always returns the same meta-information.

Table 3. Performance Overhead due to Lack of Dynamic Aspects

	not shared	shared & not enabled	shared (no `if`)	shared & enabled
Time (seconds)	0.586430	8.755212	54.023821	63.328594
Overhead factor	1	15	92	108

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. 10. 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 3.

The first column shows the time spent inside `getBalance` for a standard bank account object. The third column shows the time spent inside `getBalance` when *Shared* has been applied to the bank account object (but in this case without an `if` pointcut modifier in the pointcut). 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 s (a factor of 15!) when shared is disabled, and a slowdown of 9.3 s 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.

Performance Impact of Writing Reusable Pointcuts. The last experiment we conducted aimed at evaluating the performance loss incurred in *AspectJ* due to having to work around the reflection/super class execution dilemma. In Sect. 4.4, we described that with a targeted *call* pointcut we cannot handle reflective calls, whereas with a targeted *execution* pointcut we cannot handle

Table 4. Comparing Application-Specific and Reusable Pointcuts

	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

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 4.

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.

5 Related Work

The ideas and techniques investigated in this paper intersect with a broad spectrum of research projects on transactional systems and aspects, reusable aspect-oriented frameworks, and aspect dependencies and interactions. We present the most relevant related work in this section.

5.1 Aspects Implementing General Application Concerns

Aspects for Concurrent Programming. Cunha et al. [3] investigated techniques for implementing reusable aspects for high-level concurrency mechanisms in *AspectJ*. The authors illustrated how abstract pointcut interfaces and annotations can be used to implement one-way calls, synchronization barriers, reader/writer locks, and schedulers. The performance overhead and reusability of an object-oriented implementation of these mechanisms was compared to their aspect-oriented implementation. They concluded that the *AspectJ* implementation is more reusable and pluggable, but incurs a noticeable performance overhead. However, *AspectJ* was found to have a limitation in acquiring local

joinpoint information in concrete aspects: when a superaspect defines an abstract pointcut, the subaspects cannot change the pointcut's signature.

Similar to our work, Cunha et al. used annotations to denote methods that require special processing at run time. However, their work did not address the issue of aspect dependencies and interactions that may occur when these concurrency mechanisms are applied to a common method. In addition, their technique used in supporting aspect reusability is different and requires additional code. In their case, developers must provide concrete pointcuts and advice for each of the abstract pointcuts, which can be error-prone if done incorrectly. 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.

Persistence Aspects. Rashid et al. [5, 39] have worked extensively on techniques which apply AOP concepts to database systems. In [5], the authors explored three issues in the context of AOP and data persistence: the possibility of using AOP techniques in aspectizing persistence, the reusability of persistence aspects, and whether persistence aspects could be developed independently of an application. Using a relational database application as an example, they demonstrated incrementally how reusable aspects for database connections, data storage and updates, data retrieval, and data deletion can be implemented in *AspectJ*. It was concluded that persistence can indeed be aspectized in a reusable way, but can only be partially developed independently of an application since operations such as data retrieval and deletion must be explicitly considered.

Rashid et al. achieved reusability by requiring all classes whose instances are to be made persistent extend a common base class. Since Java does not support multiple inheritance, classes that already extend other classes cannot be made persistent without some degree of code restructuring. In contrast, reuse in AspectOptima is achieved through marker interfaces—aspects are applied to classes that implement their associated interfaces—eliminating concerns about multiple inheritance support. In addition, our *Persistence* aspect is built upon reusable aspects that can be useful in non-transactional contexts too, whereas their implementation uses database specific code that cannot be reused in other contexts. On the other hand, our *Persistence* aspect currently does not support databases.

Aspects for Distributed Applications. In [40], the authors proposed JAC, an AOP-based middleware for building distributed Java applications. JAC separates aspect binding and crosscutting code into two different modules, respectively aspect components and dynamic wrappers, facilitating aspect reuse. Support for run time aspect deployment is achieved through load-time transformations, whereas our aspects must all be weaved in at compile-time and then selectively enabled or disabled. To achieve aspect distribution, JAC offers a container mechanism that hosts both application objects and aspect component instances and makes them remotely accessible using either CORBA or RMI.

Similar to our transaction aspect, developers can reuse the aspects that come with the JAC framework (e.g., tracing, persistence, authentication, session, load-balancing, to name a few) easily within their application. Configuration is achieved through a separate configuration file. JAC differs from AspectOptima because the functionality that JAC provides is not achieved by composing individually reusable base aspects, and hence aspect dependencies and interferences also are not addressed in a general and reusable way. The JAC documentation does not explicitly mention aspect dependencies and interferences, but it is safe to assume that they are handled internally for every possible aspect combination.

5.2 Aspects and Transactions

The work of Fabry et al. [41, 42] applies AOP concepts to advanced transaction models, e.g. nested and long running transactions. The authors argued that the high complexity and inadequate separation of concerns in these models impedes their use by application programmers. In order to encourage the use of these models, they proposed a domain-specific aspect language called KALA for modularizing the concepts of advanced transaction models into aspects. KALA is based on the ACTA formalism [43].

The main goal of our work is to define an aspect framework with many individually reusable aspects that can be combined in several ways according to the application developers needs. AspectOptima is not meant to be used in a high-performance transactional application, but rather serves as a real-world aspect framework for experimenting with intricate aspect dependencies and interactions. While KALA aims at synthesizing new transaction models and at providing an elegant interface for defining transaction boundaries to an application programmer, our framework simply aims at implementing several concurrency control and recovery strategies by combining individually reusable aspects in different ways.

5.3 Reusable Aspect Design Frameworks

Our ten aspects can be combined in different ways to implement different concurrency control and recovery strategies. A few design frameworks promise this kind of customizable composability, and we present some of them later. This version of our design did not follow any particular methodology, so it would be interesting future work to compare it with similar designs obtained through some of the following frameworks.

FODA. Feature-Oriented Domain Analysis (FODA) [44] is a domain analysis method for product line development, i.e., a family of systems in a domain, rather than a single system. Domain products, representing the common functionality and architecture of applications in a domain, are produced from domain analysis. Specific applications in the domain may be developed as refinements of the domain products.

As part of the process, FODA prescribes the creation of a feature model. Features are defined as attributes, properties, functions, capabilities, or services

of a system that directly affect end-users. The feature interaction model allows the developer to specify dependencies among features, such as *specialization*, *optional*, *requires*, *mutually exclusive with*. When building a concrete application, the developer has to specify which features the final application should contain.

FODA is a domain analysis method, and hence very different from aspect frameworks such as AspectOptima. An interesting experiment would be to apply FODA to the transaction domain and compare the *user-oriented* features identified in the FODA feature model with the AspectOptima aspects.

Framed Aspects. In [45] the authors propose *framed aspects*, an approach that uses AOP to modularize crosscutting and tangled concerns and frame technology [46] to allow aspect parameterization, configuration, and customization. In framed aspects, the identification of features (here called feature aspects), and detection of dependencies and interferences, is performed following the high-level feature interaction approach promoted by FODA. Once this is done, the features are modularized within *framed aspects*, together with their dependencies.

Framed aspects are made up of three distinct modules: the framed aspect code (normal and parameterized aspect code), composition rules (aspect dependencies, acceptable and incompatible aspect configurations), and specifications (user-specific customization). These modules are composed to generate customized aspect code using a frame processor. Framed aspects achieve separate aspect bindings and aspect dependencies through parameterization and composition rules respectively. Composition rules can also be used for specifying acceptable and incompatible aspect configurations. The above mentioned constructs enable framed aspects to be reused in contexts other than that for which they were implemented.

It would be very interesting to attempt an implementation of AspectOptima using framed aspects in order to investigate how well this approach supports the language features presented in Sect. 4.1.

Ahead. Reference [47] describes a generic mechanism to generate a group of artifacts (code, documentation, makefiles, uml diagrams...) which, together, describe a program implementing a given subset of the available features. To do so, they first choose a labelled-tree representation for each artifact, and they interpret the resulting structure as a hierarchy of objects containing other objects. They then label the containment relationships according to the features they implement, extracting similarly purposed relationships into independently injectable mixins, and grouping the mixins into "layers" representing the features.

This mixin-based strategy is strikingly similar to CaesarJ's collaboration composition mechanism [28], and quite different from AspectJ's strategy. Attempting an implementation with this kind of language should be especially insightful.

6 Conclusions

Designing and implementing the ACID properties of transactional objects is a simple, but non-trivial, real-world example to which aspect-oriented techniques can be applied. The first part of the paper presents AspectOptima, a

language-independent, aspect-oriented framework that ensures the ACID properties for transactional objects. The framework consists of ten base aspects at the lowest level, each one providing a well-defined reusable functionality. The base aspects are simple, yet have complex dependencies among each other. We demonstrated how the base aspects can be configured and composed in different ways to implement different concurrency control and recovery strategies. This composition is delicate: some aspects conflict with each other or require a specific invocation ordering, others have to be reconfigured dynamically at run time.

All of the above, and the fact that AspectOptima has not been invented to promote a particular aspect-oriented system, makes it an ideal challenge case study for the aspect-oriented community. In particular, we believe that it can be used to evaluate:

- *Aspect-Oriented Software Development Processes:* We performed our decomposition into aspects based on our previous implementation experience. How does an AOSD process perform when applied to this case study? Is the resulting decomposition equally simple, modular, and reusable? Can the process identify aspect dependencies and conflicts?
- *Aspect-Oriented Modeling Notations:* Can an AOM notation capture the complex structural and behavioral dependencies of the aspects in this case study? Are the resulting models easy to understand? Are the models reusable? Can aspect dependencies and conflicts be expressed?
- *Aspect-Oriented Validation and Verification:* Can AO formal methods and AO testing techniques be used to individually validate and verify each aspect in a stand-alone way? Can these methods detect conflicts between aspects?
- *AOP Implementations:* How do different AOP implementations compare with respect to performance and memory footprint? How do they compare to standard OO implementations? Is run time weaving to implement dynamic AOP faster than static weaving with run time checks?
- *Aspect-Oriented Language Features:* What are the features that an AO programming language must provide in order to implement this case study? What features can promote good software engineering properties such as encapsulation, modularization, testability, maintainability, and reusability? What features are required to support aspect dependencies and resolution of conflicts in a reusable way?

In order to give an idea on how AspectOptima can be used to evaluate the expressiveness of aspect-oriented languages, we presented in the second part of this paper an implementation of AspectOptima in *AspectJ*. We identified six key language features that an aspect-oriented language must provide in order to implement AspectOptima in a satisfactory way: separate aspect binding, inter-aspect configurability, inter-aspect ordering, per-object aspects, dynamic aspects, and per-thread aspects. We then showed parts of our implementation to demonstrate that *AspectJ* provides sufficient, but certainly not ideal or elegant, support for implementing reusable aspect frameworks and dealing with mutually dependent and interfering aspects. We discussed the encountered

language limitations, suggested possible language improvements where appropriate, and presented some preliminary measurements that highlight the performance impact of certain language features.

7 Future Work

We have demonstrated how AspectOptima can be used to evaluate the language features of *AspectJ*, and intend to do the same for other AOP languages in the immediate future. We intend to define different benchmarks that can be used to compare the performance of these AOP environments, and run these benchmarks on our implementations to obtain reference measurements. We also plan to run the benchmarks on the object-oriented implementation of Optima [9, 17] and compare the results.

We are also working on extending AspectOptima. First of all, concurrency control and recovery can be further enhanced when more information about the semantics of operations is available. We intend to define a *SemanticClassified* aspect that defines forward and backward commutativity for all operations of an object, and maps every operation to a corresponding inverse operation. Such semantic information opens the door to semantic-based concurrency control [11] and logical recovery based on intention lists.

The work described in this paper has focused on the identification and implementation of aspects that crosscut objects. However, the *Versioned*, *Tracked*, and *AutoRecoverable* aspects share a common need for a well-defined *region of computation* (zone/view) that threads can be associated with, and during which certain actions (such as object accesses) are to be monitored. This is a crosscutting concern of *threads of computation*, rather than a crosscutting concern of objects. We have already started to extend AspectOptima beyond object-centered aspects, implementing transaction life-cycle management with aspects. Initial results can be found in [48].

Acknowledgments

This research has been partially funded by the Natural Sciences and Engineering Research Council of Canada (NSERC). The authors would also like to thank Jean-Sebastien Légaré and Isaac Yuen for their work on the implementation of the AspectOptima prototype, and the anonymous reviewers.

References

[1] Elrad, T., Aksit, M., Kiczales, G., Lieberherr, K., Ossher, H.: Discussing Aspects of AOP. Communications of the ACM 44, 33–38 (2001)
[2] Soares, S., Laureano, E., Borba, P.: Implementing Distribution and Persistence Aspects with AspectJ. In: Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 174–190. ACM Press, New York (2002)

[3] Cunha, C.A., Sobral, J.L., Monteiro, M.P.: Reusable Aspect-Oriented Implementations of Concurrency Control Patterns and Mechanisms. In: Proceedings of the 5th International Conference on Aspect-Oriented Software Development - AOSD 2006, pp. 134–145. ACM Press, New York (2006)
[4] Kienzle, J., Guerraoui, R.: AOP - Does It Make Sense? The Case of Concurrency and Failures. In: Magnusson, B. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 37–61. Springer, Heidelberg (2002)
[5] Rashid, A., Chitchyan, R.: Persistence as an Aspect. In: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development - AOSD 2003, pp. 120–129. ACM Press, New York (2003)
[6] Kienzle, J., Gélineau, S.: 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, pp. 202–213. ACM Press, New York (2006)
[7] Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers, San Mateo (1993)
[8] 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, Roma, Italy, 2001, pp. 209–217. IEEE Computer Society Press, Los Alamitos (2001)
[9] Kienzle, J.: Open Multithreaded Transactions — A Transaction Model for Concurrent Object-Oriented Programming. Kluwer Academic Publishers, Dordrecht (2003)
[10] Papadimitriou, C.: The Serializability of Concurrent Database Updates. Journal of the ACM 26, 631–653 (1979)
[11] Ramamritham, K., Chrysanthis, P.K.: Advances in concurrency control and transaction processing, Los Alamitos, California (1997)
[12] Kung, H.T., Robinson, J.T.: On Optimistic Methods for Concurrency Control. ACM Transactions on Database Systems 6, 213–226 (1981)
[13] Bernstein, P.A., Goodman, N.: Concurrency Control in Distributed Database Systems. ACM Computing Surveys 13, 185–221 (1981)
[14] Papadimitriou, C.H., Kanellakis, P.C.: On Concurrency Control by Multiple Versions. ACM Transactions on Database Systems 9, 89–99 (1984)
[15] Agrawal, D., Sengupta, S.: Modular Synchronization in Multiversion Databases: Version Control and Concurrency Control. In: Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data, Portland, Oregon, pp. 408–417. ACM Press, New York (1989)
[16] Lampson, B., Sturgis, H.: Crash Recovery in a Distributed Data Storage System. Technical report, XEROX Research Center, Palo Alto (1979)
[17] Kienzle, J., Jiménez-Peris, R., Romanovsky, A., Patiño-Martinez, M.: Transaction Support for Ada. In: Strohmeier, A., Craeynest, D. (eds.) Ada-Europe 2001. LNCS, vol. 2043, pp. 290–304. Springer, Heidelberg (2001)
[18] Kienzle, J., Romanovsky, A.: A framework based on design patterns for providing persistence in object-oriented programming languages. IEEE Proceedings of Software Engineering 149, 77–85 (2002)
[19] Riehle, D., Siberski, W., Bäumer, D., Megert, D., Züllighoven, H.: Serializer. In: Pattern Languages of Program Design, vol. 3, pp. 293–312. Addison-Wesley, Reading (1998)

[20] Krasner, G., Pope, S.: A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. Journal of Object-Oriented Programming 1, 26–49 (1988)

[21] Kienzle, J., Strohmeier, A.: Shared Recoverable Objects. In: Harbour, M.G., la de Puente, J.A. (eds.) Ada-Europe 1999. LNCS, vol. 1622, pp. 397–411. Springer, Heidelberg (1999)

[22] Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P., Morrison, R.: An Approach to Persistent Programming. Computer Journal 26, 360–365 (1983)

[23] Eswaran, K.P., Gray, J., Lorie, R.A., Traiger, I.L.: The Notion of Consistency and Predicate Locks in a Database System. Communications of the ACM 19, 624–633 (1976)

[24] Kiczales, G., Hilsdale, E., Hugunin, J., Kersen, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–357. Springer, Heidelberg (2001)

[25] Workshop on Software Engineering Properties of Languages and Aspect Technologies – SPLAT (2003 - 2007)

[26] Workshop on Foundations of Aspect-Oriented Languages – FOAL (2002 - 2007)

[27] Gosling, J., Joy, B., Steele, G.L.: The Java Language Specification. The Java Series. Addison Wesley, Reading (1996)

[28] Aracic, I., Gasiunas, V., Mezini, M., Ostermann, K.: An overview of caesarJ. In: Rashid, A., Aksit, M. (eds.) Transactions on Aspect-Oriented Software Development I. LNCS, vol. 3880, pp. 135–173. Springer, Heidelberg (2006)

[29] Hanenberg, S., Costanza, P.: Connecting Aspects in AspectJ: Strategies vs. Patterns. In: First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (2002)

[30] Hanenberg, S., Unland, R.: Parametric Introductions. In: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development - AOSD 2003, pp. 80–89. ACM Press, New York (2003)

[31] Duala-Ekoko, E.: 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 (2006)

[32] Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: abc: an Extensible AspectJ Compiler. In: AOSD 2005: Proceedings of the 4th international conference on Aspect-oriented software development, pp. 87–98. ACM Press, New York (2005)

[33] Schmidmeier, A., Hanenberg, S., Unland, R.: Known Concepts Implemented in AspectJ. In: 3rd Workshop on Aspect-Oriented Software Development (AOSD-GI) of the SIG Object-Oriented Software Development, German Informatics Society (2003)

[34] Xerox Corporation: Frequently Asked Questions about AspectJ (2006), `http://www.eclipse.org/aspectj/doc/released/faq.html`

[35] Hilsdale, E., Hugunin, J.: Advice Weaving in AspectJ. In: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development - AOSD 2004, pp. 26–35. ACM Press, New York (2004)

[36] Burke, B., Chau, A., Fleury, M., Brock, A., Godwin, A., Gliebe, H.: JBoss Aspect-Oriented Programming (2004)

[37] The Eclipse Project: Eclipse Test and Performance Tools Platform (2007), `http://www.eclipse.org/tptp/`

[38] Bockisch, C., Arnold, M., Dinkelaker, T., Mezini, M.: Adapting Virtual Machine Techniques for Seamless Aspect Support. In: ACM Sigplan International Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 109–124 (2006)

[39] Rashid, A.: Aspect-Oriented Database Systems. Springer, Heidelberg (2004)

[40] Pawlak, R., Seinturier, L., Duchien, L., Florin, G., Legond-Aubry, F., Martelli, L.: JAC: an Aspect-Based Distributed Dynamic Framework. Software Practice and Experience 34, 1119–1148 (2004)

[41] Fabry, J., Cleenewerck, T.: Aspect-Oriented Domain Specific Languages for Advanced Transaction Management. In: International Conference on Enterprise Information Systems 2005 (ICEIS 2005) proceedings, pp. 428–432. Springer, Heidelberg (2005)

[42] Fabry, J., D'Hondt, T.: KALA: Kernel Aspect Language for Advanced Transactions. In: SAC 2006: Proceedings of the ACM Symposium on Applied Computing, pp. 1615–1620. ACM Press, New York (2006)

[43] Chrysanthis, P.K., Ramamritham, K.: Synthesis of Extended Transaction Models using ACTA. ACM Transactions on Database Systems 19, 450–491 (1994)

[44] Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (1990)

[45] Loughran, N., Rashid, A.: Framed Aspects: Supporting Variability and Configurability for AOP. In: International Conference on Software Reuse (ICSR-8), pp. 127–140. Springer, Berlin (2004)

[46] Bassett, P.: Framing Software Reuse: Lessons from the Real World. Prentice-Hall, Inc., Upper Saddle River (1997)

[47] Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Transactions on Software Engineering 30, 355–371 (2004)

[48] Bölükbaşi, G.: Aspectual Decomposition of Transactions. Master's thesis, School of Computer Science, McGill University, Montreal, Canada (2007)