

# XL-AOF – Lightweight Aspects for Space-Based Computing

eva Kühn  
eva@complang.tuwien.ac.at

Fabian Schmied  
fabian@complang.tuwien.ac.at

Institute of Computer Languages  
Vienna University of Technology, Austria  
Argentinierstraße 8, A-1040 Wien  
Web: [www.complang.tuwien.ac.at/eva](http://www.complang.tuwien.ac.at/eva)

## ABSTRACT

Space-based computing is a powerful model of abstraction for distributed application development. Although such applications solve a high number of cross-cutting concerns, there is no aspect-oriented environment available at the moment which supports the space-based communication paradigm. This paper describes XL-AOF, an extensible lightweight aspect-oriented framework, whose main focus is to allow for easy development of space-based applications.

## Categories and Subject Descriptors

C.2.4 [Computer-Communication Networks]: Distributed Systems—*Distributed applications*; D.1.m [Programming Techniques]: Miscellaneous; D.3.3 [Programming Languages]: Language Constructs and Features—*Frameworks*; D.3.4 [Programming Languages]: Processors—*Run-time environments*

## Keywords

Aspect-oriented programming, middleware, space-based computing, .NET, declarativity, interception

## 1 INTRODUCTION

Compared to local applications, distributed applications introduce a number of coordination issues, such as data transfer, concurrency control, and event notification requirements. Middleware layers strive to make that inherent complexity manageable: they take over issues of distributed application development, abstracting them by providing a defined coordination model.

As a particularly interesting model of abstraction, *space-based* middleware systems (space-based computing, SBC) define the model of a *virtual space*. Clients can put data tuples (objects) with defined interaction properties into the space, letting others see and modify them. Providing a very natural way of stateful communication

for data-driven applications, the space abstraction completely decouples the application from traditional message passing concerns. Low-level messaging-related problems, e.g. polling for data changes, are hidden via middleware services for event notification, locking, and data replication. Examples of space-based systems include JavaSpaces [16], Coordination Kernel CORSO [4], XVSM [5], and Linda tuple spaces [8]. Such a high-level communication paradigm should naturally be interfaced via an equally sophisticated programming paradigm, but traditionally, it isn't [14].

In the AOP community, it has been known for years that concerns of distributed applications are typical cross-cutting concerns, orthogonal to application functionality [11]. Aspect-oriented programming is therefore a well-suited paradigm for dealing with such applications: it allows the developer to cleanly encapsulate the concerns of distribution, which results in better separation of concerns, improved code and software quality, and—due to clear separation of responsibilities—a better team development process [14].

There already exist some successful projects which combine middleware functionality with an aspect-oriented programming model. JBoss AOP [2], for example, is an aspect-oriented framework to be used from within the JBoss J2EE Application Container. Spring [9] is an application framework which also contains aspect-oriented features. However, existing AOP frameworks are often tied to a specific container architecture, define language extensions, need dedicated compilers or virtual machines, or require complicated XML-based configuration. Other approaches are rather academic, defining special-purpose programming languages, running on modified Java virtual machines, etc, resulting in them not being adopted by the industry. And none of them is explicitly suited for space-based application development.

This paper therefore describes XL-AOF, an extensible lightweight aspect-oriented framework with declarative elements, developed to support space-based computing. It is currently being implemented for the .NET runtime environment (including Microsoft .NET and Mono support), but could easily be ported to Java 1.5 with some minor modifications. XL-AOF aims to fulfill the following goals:

- Provide an aspect-oriented environment for distributed applications, while building on existing space-based know-how,
- provide a set of predefined aspects for common concerns of space-based computing applications,
- allow for easy addition of custom aspects,
- focus on reusable aspects and aspect-friendly program design,
- leverage the declarative features (metadata, attributes) provided by .NET and Java for natural aspect configuration, and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.  
AOMD '05, November 28-December 2, 2005, Grenoble, France  
Copyright 2005 ACM 1-59593-265-8/05/11 ...\$5.00

- make it easily adoptable, requiring no special compiler, special-purpose programming language, application container, or operating system.

XL-AOF is not meant to be a can-do-it-all application container. It is not designed for arbitrary or even high-performance AOP use cases (although many of these can be implemented using XL-AOF). Its explicit purpose is to facilitate the development of space-based, distributed applications, resulting in an overall improved development cycle of such applications.

The rest of this paper is structured as follows: section 2 explains the architecture of XL-AOF, describing its dedicated lightweight aspect and join point model, as well as a number of typical aspects in the scope of space-based computing. In section 3, we give some details on the implementation of XL-AOF, and in section 4, we give insight in what has already been done and what is planned for the future. Section 5 concludes the paper.

## 2 LIGHTWEIGHT ARCHITECTURE

Currently, the reference model of aspect-oriented programming environments is that of AspectJ [12]. AspectJ is an extension of the Java programming language, allowing for implementation of complex aspect-oriented programs within different areas of application. XL-AOF therefore takes over many notions and terms from AspectJ, but there is one important difference—XL-AOF is a lightweight AOP environment: to achieve good acceptance in the industry, XL-AOF specifically does *not* define a new programming language or language extension. It is defined completely within the boundaries of the .NET Common Language Infrastructure [3], and does not require any non-standard tools. It therefore cannot simply take over the proven concepts of AspectJ (which were designed to be implemented with a weaving compiler), but has to define its own model, always with the goals of light weight and space-based computing in mind.

To establish a common vocabulary, we will use the words *aspect*, *join point*, *advice*, and *pointcut* with their usual meanings [10, 12]. For readers less acquainted with AOP, we will nevertheless shortly describe them on first use. The well-known term *weaving* will be used to denote the process of combining an aspect and a functional component (which stands for any class implementing non-cross-cutting application functionality), regardless of the technical implementation of this process. The functional components an aspect is woven to are called its *target components*.

### 2.1 Aspect Model

In XL-AOF's aspect model, aspects are modules of encapsulation, applied to functional components to change them in two ways:

- To extend their public interface, and
- To alter their behavior.

For the first functionality, consider, for example, a class *Shape*, which is to be used in a distributed application. One typical concern of distribution—which is not specific to space-based computing—is that *Shape* objects need to be serialized in order to be transferred over the network (and put into the space). Serialization is an orthogonal concern which can be implemented as a separate aspect. When the aspect is applied to a *Shape* component, the component's interface is extended so that other objects can call serialization methods on it. This is explained in more detail in section 2.2.

The second point is roughly equivalent to AspectJ's concept of join points and pointcuts. During the imperative program flow of an application, there are numerous points where aspects can join in, so-called join points. Aspects can provide methods (advice) which are invoked at these points and change the behavior of the target components. Detailed information on this is given in section 2.3.

#### 2.1.1 Aspect Definition and Application

Aspects are implemented in form of ordinary classes. They can define fields and methods, implement interfaces, derive from other classes, just like any ordinary object-oriented class. A class becomes an aspect only through a dedicated *factory attribute* for that aspect; a custom metadata annotation type (as defined by the .NET Common Language Infrastructure), which, when applied to a functional component, causes it to become the target of the aspect. Figure 1 illustrates the relationship between component, aspect, and factory attribute.

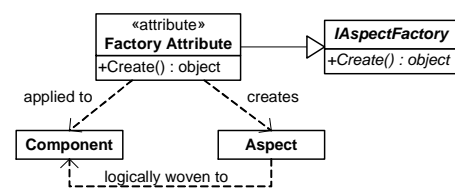


Figure 1. Aspect, component, and factory attribute.

As shown in the figure, factory attributes implement the *IAspectFactory* interface predefined by XL-AOF. The framework uses this interface to distinguish factory attributes from ordinary, purely informational custom attributes, as well as for retrieving aspect instances: factory attributes implement a *Create* method, which is invoked by XL-AOF and must return an aspect instance. By implementing this method, the factory attribute controls the instantiation model of the aspect, which is one of three possibilities:

**Singleton aspects** only have one instance, which is used for all target components. This is an efficient instantiation model for aspects which hold only global state or none at all.

**One-to-one aspects** have exactly one instance per target component. This is the usual instantiation model for aspects which hold state related to the target component.

**One-to-many aspects** can have several instances, each of which controls a group of target components. This is sometimes useful (e.g. when a security aspect manages all components with the same access policy), but not as common as the first two instantiation models.

#### 2.1.2 Aspect Dependencies

Aspects will usually not be able to fully work on their own. A transaction handling aspect, for example, probably needs to have access to the middleware layer so that it can create, commit, and abort transactions. A serialization aspect needs to access the component's state. The entities which can be accessed by an aspect, i.e. the *aspect dependencies*, are also described by the aspect's factory attribute. With XL-AOF, aspects can depend on the following:

- The component itself and the public interface of its class,
- The interface extensions provided by other aspects, and
- Environmental objects, the most noteworthy being the middleware connection.

XL-AOF uses the dependency information given by the factory attribute and calculates the right order of instantiation of target component and aspects. For an example, figure 2 shows a component, which implements an interface *IOne*. *AspectA*, which is woven to the component, extends its public interface by adding the methods specified by interface *ITwo*. *AspectB*, which is also woven to the component, depends on both interface *IOne* and *ITwo*. Therefore, XL-AOF guarantees that the component and *AspectA* are instantiated before *AspectB*, which can therefore rely on its dependencies to be fulfilled. If XL-AOF cannot fulfill a dependency (e.g. because a required interface is neither implemented by the target component nor added by any aspect) or if circular dependencies are detected, the aspect design is invalid and an error is raised.

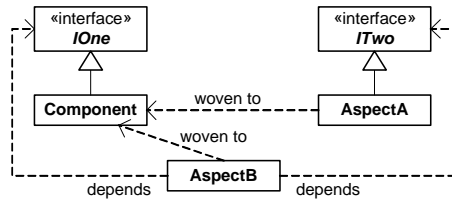


Figure 2. Aspect dependencies.

## 2.2 Interface Extension

The first functionality of aspects is that of extending the public interface of a component. XL-AOF makes this completely transparent: at runtime, the public interface of a component is automatically augmented by those interfaces implemented by the aspects woven to it. For example, consider figure 3. It shows the serialization aspect mentioned before, which is applied to a class *Shape* (by means of a factory attribute not shown in the figure). The aspect implements the interface *ISerializable*, providing *Write* and *Read* methods for serializing and deserializing component state. This extends the public interface of *Shape* objects, so that, at runtime, they themselves seem to implement the *ISerializable* interface. In reality, all calls to *Read* and *Write* are routed to the aspect's implementation.

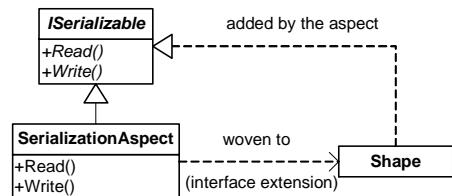


Figure 3. Interface extension through aspects.

## 2.3 Behavior Alteration

The second functionality of aspects is that of influencing the imperative program flow of an application. In the literature, this is often considered the main purpose of aspect-oriented programming: aspects are applied to arbitrary object-oriented code, which is *oblivious*, i.e. completely unaware of the additional orthogonal concerns added by the aspects. The power of aspect-oriented systems is then said to be determined by its means for *quantification*, i.e. the possibilities it offers for binding aspects to oblivious code. [7]

XL-AOF doesn't completely agree with this opinion. While it is definitely a fascinating idea to have fully oblivious functional components being extended by aspects, reality has shown that obliviousness leads to awkward work-arounds, specialized and ad-hoc

pointcut expressions, and hardly reusable aspects [15]. We therefore are of the opinion that fully oblivious functional components are not desirable in an aspect-oriented environment. Instead, one should strive to achieve an *aspect-friendly* design within the functional components in order to fully leverage the power of aspect-oriented programming. So-called *shy* aspects [15] should be preferred, which are not specifically designed to fit a certain functional component, making aspect reuse much easier.

With XL-AOF, changing behavior via aspects is done by means of a dynamic join point concept. Because XL-AOF cannot rely on a separate compiler for the weaving process, it does not define low-level join points for field access, control flow, or similar. Instead it defines a number of high-level join points well-suited for space-based development, the most important of which are listed here:

**Component creation:** Enables an aspect to provide a factory advice method for its target components. If a component instance is to be created whose class is connected with an aspect bound to this join point, the factory advice is used instead of the default *new* operator. This is useful for implementing caches and pooling in distributed environments (see section 2.5).

**Component construction:** This join point is reached after the target component of an aspect has been fully constructed. This is especially useful for singleton aspects that need to perform certain operations when its target components are created.

**Component finalization:** An aspect can bind to this join point to be informed when a target component is garbage-collected. That way, it can dispose of middleware resources when they aren't needed any more.

**Component disposal:** With the *IDisposable* pattern [13], .NET objects can deterministically release any resources they hold when they are not needed any more (e.g. because they go out of scope). This is sensible in case of very scarce resources, for which indeterministic finalization via a garbage collector is insufficient. The join point is reached when the user of a component employs the *IDisposable* pattern to dispose of it. Its use cases are similar to those of the finalization join point.

**Before/after/around method and property invocation:** These classical join points are reached before or after a method of the functional component is executed. Around advice methods are invoked instead of the original method and can choose whether, when, and how often it should be invoked.

**Exception escaping from a method:** In distributed environments, it is often the case that unforeseen exceptions occur. Connections can be broken, data can be lost, concurrency conflicts and authorization issues can be detected, etc. For such cases, error handling aspects can be attached to this join point in order to handle all or certain unexpected exceptions.

In addition to these predefined join points—and in contrast to classic aspect-oriented environments—XL-AOF's join point model is extensible: every aspect and functional component can define new kinds of join points and trigger them at any point within the imperative program flow. Aspects can bind to these custom join points exactly as they bind to the predefined ones. This enables software developers to create decoupled aspect-oriented designs with shy, reusable aspects, while it brings expressiveness similar to that of classic low-level join point models.

Some examples of custom join points will be given in section 2.5, together with the aspects that define them.

## 2.4 Binding Advice Methods to Join Points

As previously explained, join points describe points in the imperative program flow of an application to which advice methods can be bound. An aspect only sees those instances of join points that are triggered from the context of its target components. This means that an aspect must be woven to a certain class (using the factory attribute) in order to bind an advice method to one of its join points.

However, an aspect would usually not be interested in all join point instances triggered by its target components. Therefore, specific join point instances can be selected via *pointcuts*. XL-AOF distinguishes between two kinds of pointcuts:

**Client-specified pointcuts** allow the client to denote the join point instances which should trigger an advice method. The aspect defines a custom attribute, which is then applied to all target points within the component (*tagging*). An interesting benefit of this mechanism is that the tags can contain declarative parameters, which can influence the advice method's behavior. This is usually used with before, after, and around method join points, as well as exception join points.

**Aspect-specified pointcuts** allow the aspect to directly bind to all instances of a join point, transparent to the component. This is usually used with component creation, construction, finalization, and disposal, as well as most user-defined join points.

This twofold pointcut model significantly simplifies pointcuts. AspectJ, for example, specifies a very sophisticated pointcut expression language, which is necessary for aspects binding to oblivious functional components using fine-grained pointcuts. For example, to bind an advice method to all property getters of a class, an AspectJ aspect has to specify a pointcut similar to the following (given here in textual form rather than in AspectJ syntax): "bind to each method whose name starts with 'get', 'has', or 'is'".

In XL-AOF's aspect model, such fully oblivious functional components are not supported. We simply turn around responsibilities: components themselves can select the join point instances which cause advice methods to be executed. In the getter method example this means that each such method is declaratively tagged in order to cause advice to be executed. While this reduces obliviousness, it leads to reusable aspects with well-defined responsibilities (rather than relying on conventions and assumptions), as well as clear and robust pointcuts. We believe that these advantages definitely outweigh the disadvantage of a less expressive pointcut language.

For an example of each pointcut kind, consider a transaction handling aspect. This can be implemented as a singleton aspect, applied to functional components which perform operations with transaction safety requirements. The transaction handler needs to keep track of its target components, so it binds to their construction join points. All instances of this join point are important, so an aspect-specified pointcut is used.

The aspect also needs to react on method invocations in order to cause the middleware layer to begin and commit transactions. For this, it binds to the around method invocation join point. However, not all methods of the target components need transactions to be created and committed: *ToString* or *Equals*, for example, don't. Therefore, the aspect defines a custom attribute and uses a client-specified pointcut. The client tags critical methods with the custom attribute, and the advice is only invoked when needed.

This setup is illustrated in figure 4, where the *TransactionHandlerAspect* binds to a *BankAccount* component using two different pointcuts. The pointcut binding to the *ComponentConstructed* join point is aspect-specified. The pointcut binding to the *AroundMethod* join point is client-specified, requiring the *Transactional* attribute to be applied to all target instances (*Deposit*, *Withdraw*, and *Transfer*; in this example).

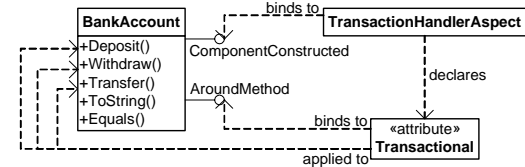


Figure 4. An aspect binding with two kinds of pointcuts.

## 2.5 Concerns of Space-Based Applications

When developing a space-based distributed application, there are a number of often-arising concerns, for which XL-AOF provides pre-defined aspects (most of them interact with the middleware layer):

**Serialization and State Retrieval:** As has already been mentioned, space-based applications require components to be serialized in order to put their data into the space. Usually, the middleware system defines its own data format, so that the programming language's serialization mechanisms cannot (easily) be used. Serialization is a typical orthogonal concern, which can be implemented by an aspect.

A generic serialization aspect could be implemented by using a reflection mechanism for retrieving the component's private state, by using its public properties, and in a number of other ways. Therefore, and for improved modularity, state retrieval had better be done by a separate aspect. Figure 5 shows these two aspects: *ReflectionStateAccessAspect* extends the component's public interface, adding an implementation of an *IStateAccess* interface, which is used by the *SerializationAspect* for implementing the middleware-defined *CorsoShareable* serialization interface.

Not shown in the figure, the serialization aspect also provides custom join points triggered before and after data is serialized and deserialized, so that other aspects can influence the process, e.g. for compressing or encrypting the serialized data.

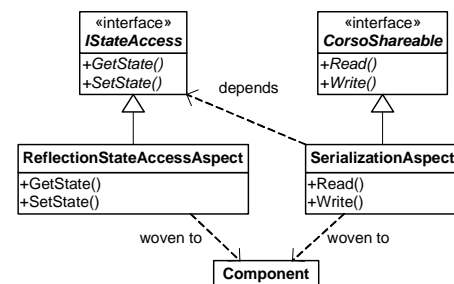


Figure 5. Serialization aspect.

**Object Sharing and Object Pooling:** In high-level space-based middleware layers, data entities in the space can have an object identity (used, for example, for building complex distributed data structures). If this identity is needed by the application, components need to store the identifier of the associated space entity. This orthogonal concern is handled by the

object sharing aspect. The pooling aspect ensures that identical space entities are always represented by the same (cached) local objects, which improves efficiency of the application.

**Change Notification and Automatic Refresh:** Space entities are usually concurrently manipulated by several distributed clients. For many use cases, local clients need to be informed when another client has changed an object. Space-based middleware layers provide notification mechanisms for this purpose. The change notification aspect encapsulates this concern and interfaces with the mechanism provided by the middleware; it also provides a custom join point triggered when a notification arrives.

In addition, some applications may require the local components' data to be refreshed in near-time. The automatic refresh aspect handles this concern by reacting on the custom join point and deserializing the object from the space.

**Transaction Handling and Transaction Error Compensation:**

Since many clients can access space objects at the same time, the concurrency issues of race conditions and non-atomic operations arise. Middleware layers usually provide some sort of transaction mechanism for this purpose, which is encapsulated by the transaction handling aspect (which was shortly described in section 2.4). When a transaction cannot be committed due to concurrency issues, it must be decided whether the application should retry the operations, ignore the failure, or display an error message of some sort. This is done by the transaction error compensation aspect.

**Unexpected Exception Handling:** This aspect does not involve any middleware functionality. Instead, it provides an easy way of handling unexpected errors, which are introduced by the properties of distribution, in a single place. For example, connection loss will often need to be handled the same way in many different places in an application: try to reestablish the connection for a while, then inform the user. Using the unexpected exception handling aspect, this can be done in a single place.

Examples for additional, user-implemented aspects include:

**(Soft) Access Control:** For example, an application could use aspects to perform highly customized and configurable security checks before executing certain operations. For this, the aspect would define a client-specified pointcut, which could then be used to tag all security-relevant operations. The tagging could also include information about what privileges are needed to execute the operation, etc.

**Data Encryption:** By binding to the custom join points of the serialization aspect, aspects can be used to transparently encrypt and decrypt data sent over the network.

**Operation Tracing:** For debugging purposes, aspects can be used to transparently trace the operations performed by an application. If they write the trace data into the space, other clients on the network can evaluate it. This has already been used for user-interface testing purposes [1] and for implementing a testing framework, checking whether all components perform their operations in the correct order and with the right results.

### 3 TECHNICAL REALIZATION

One of the most important goals of XL-AOF is to make it easily adaptable. New application development environments are often ignored if they do require special technology or extensive training.

Many good aspect-oriented environments unfortunately require both: special technology, e.g. compilers, preprocessors, application launchers, runtime environments, or virtual machines, and extensive training, mainly for new aspect programming languages. *D* [11], for example, defines an interesting—and beneficial—aspect-oriented environment for developing distributed applications, but has never found its way to the industry.

XL-AOF does not require a weaving compiler or define a new programming language. By defining a lightweight aspect model, it ensures that all concepts presented so far are implementable fully within the mechanisms provided by the Common Language Infrastructure, as it is realized by Microsoft .NET and the Mono project. This section shortly describes a possible realization of XL-AOF based on these mechanisms, namely dynamic code generation, class proxies, method interception, reflection, and custom attributes.

The first step of weaving an aspect to a component is to intercept the creation of the component. With .NET, object creation (e.g. using the *new* operator in the C# programming language) can—unfortunately—not be intercepted. XL-AOF therefore provides an object factory, which must be used for creating target component instances. When an component is created by the factory, XL-AOF first analyzes the attributes attached to its class. For every factory attribute among these, it analyzes the respective aspect's dependencies. It performs a topological sort and then creates aspect and component instances in the correct order. If one of the aspects has an advice method for the component creation join point, the component is created by this advice method; otherwise the factory creates it itself. The component construction join point is triggered by the factory when the instance has been created.

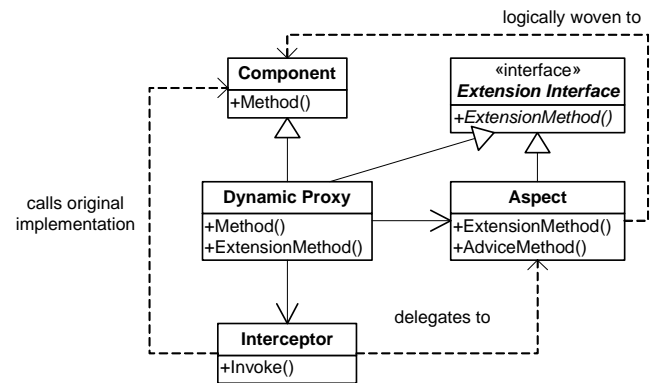


Figure 6. Runtime class model of a woven component.

In the second step of weaving, the interface extensions are performed. When the factory (or an aspect) creates the component instance, it is not directly instantiated using the *new* operator. Instead, a class proxy is created for the target class. The proxy's code is dynamically generated in such a way that it directly derives from the target class, and, in addition, implements all extension interfaces by delegating to the respective aspect instance. Through polymorphism, the proxy object can be accessed like an instance of the target class as well as through each of the extension interfaces.

To finish the weaving process, XL-AOF needs to ensure that all remaining join points are correctly triggered. To support the method invocation and exception catching join points, the dynamic proxy overrides all virtual methods defined by the target class, delegating calls to a common interceptor. The interceptor analyzes these calls for matching pointcuts (using reflection to find the pointcut

attributes applied to the methods) and calls the respective advice methods at the right points of time. By implementing a finalizer, as well as overriding the *Dispose* method, if available, the proxy is able to trigger the finalization and disposal join points.

For generation of the class proxy, the useful *DynamicProxy* library [17] of the *Castle Project* is used. Figure 6 shows the full runtime class model of a component to which one aspect is woven which adds one interface and implements one advice method.

## 4 EVALUATION AND OUTLOOK

Currently, we have a finished prototype, ATTRIBUTES &CO [6], based on the *DynamicProxy* library and implemented for the .NET 1.1 platform. ATTRIBUTES &CO is a proof of concept for our lightweight aspect model, which successfully implements five declaratively configurable, but hard-coded aspects from the list presented in section 2.5. It has shown good results; in a case study (a distributed, collaborative whiteboard application), an already advanced space-based application could still be significantly improved in terms of source code conciseness and clarity: the number of lines of code of classes dealing with distribution could be reduced by 61%. The performance impact through the dynamic proxying was not noticeable.

We are now at the beginning of implementing XL-AOF for the CORSO space-based middleware system on the upcoming version 2.0 of the Microsoft .NET platform. The work will extend the ATTRIBUTES &CO prototype to implement the full XL-AOF model, including all aspects described in this paper.

For the future, we are investigating an implementation of XL-AOF with Java 1.5, which seems to be possible with some modifications, primarily workarounds for the fact that Java annotation types cannot specify method implementations and thus cannot act as factory attributes.

## 5 CONCLUSION

In this paper, we presented XL-AOF, an extensible lightweight aspect-oriented framework for developing space-based applications. XL-AOF defines an aspect-oriented environment for distributed applications based on the space communication model, significantly reducing development complexity of such—already highly abstracted—applications. It provides a natural, declarative way of aspect configuration, and a lightweight aspect model, which can be employed without any special compiler or runtime environment. Its extensible join point model allows for decoupled aspect designs and reusable aspect implementations, not only in space-based, but in general aspect-oriented applications. Prototypes of XL-AOF have shown great potential, and the full framework is currently under development for the upcoming .NET 2.0 platform.

## 6 References

- [1] R. Abe, M. Beinhart, et al. Need for rigorous methods and tools in collaborative mobile software solutions – a use case study on a travel service application. Technical report, Vienna University of Technology, E185/1, Vienna, Austria, 2003.
- [2] B. Burke. *JBoss Aspect-Oriented Programming (AOP)*, 2004. Available from: <http://www.jboss.org/products/aop>.
- [3] ECMA International. Standard ECMA-335 – Common Language Infrastructure (CLI), 3rd Edition. Technical report, 2005. Available from: <http://www.ecma-international.org/publications/standards/ecma-335.htm>.
- [4] eva Kühn. Fault-tolerance for communicating multidatabase transactions. In *IEEE Proceedings of the 27th Hawaii International Conference on System Sciences (HICSS, Wailea, Maui, Hawaii, January 4–7 1994)*, pages 323–332, 1994.
- [5] eva Kühn, M. Beinhart, and M. Murth. Improving data quality of mobile internet applications with an extensible virtual shared memory approach (to appear at the Iadis WWW/Internet 2005 Conference, Lisbon, Portugal, October 19–22 2005). 2005.
- [6] eva Kühn and F. Schmied. Attributes &Co – collaborative applications with declarative shared objects (to appear at the Iadis WWW/Internet 2005 Conference, Lisbon, Portugal, October 19–22 2005). 2005.
- [7] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns (OOPSLA)*, 2000.
- [8] D. Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.
- [9] R. Johnson, J. Hoeller, et al. *Spring - Java/J2EE Application Framework*, 2005. Available from: <http://static.springframework.org/spring/docs/1.2.x/spring-reference.pdf>.
- [10] G. Kiczales, J. Lamping, et al. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.
- [11] C. V. Lopes and G. Kiczales. D: A language framework for distributed programming. Technical Report SPL97-010, P9710047, Palo Alto, CA, USA, February 1997. Available from: [citeseer.ist.psu.edu/lopes97language.html](http://citeseer.ist.psu.edu/lopes97language.html).
- [12] C. V. Lopes and G. Kiczales. Recent developments in AspectJ. In *Proc. European Conference on Object-Oriented Programming (ECOOP 98)*, 1998.
- [13] R. Mariani. Garbage collector basics and performance hits. Technical report. Available from: <http://msdn.microsoft.com/library/en-us/dndotnet/html/dotnetgcbasics.asp>.
- [14] F. Schmied and eva Kühn. Distributed peer-to-peer application development with declarative and aspect-oriented techniques. In *Conference Proceedings of International Symposium on Leveraging Applications of Formal Methods (ISoLA, Paphos, Cyprus, October 30 – November 2 2004)*, pages 150–157, 2004.
- [15] K. Sullivan, W. G. Griswold, et al. On the criteria to be used in decomposing systems into aspects. In *Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005, Lisbon, Portugal, September 5–9 2005)*, 2005.
- [16] Sun Microsystems. *Javaspace<sup>TM</sup> service specification*. Technical report, 2003. Available from: <http://java.sun.com/products/jini/2.0/doc/specs/html/js-title.html>.
- [17] H. Verissimo. *Castle's DynamicProxy for .NET*, 2004. Available from: <http://www.codeproject.com/csharp/hamiltondynamicproxy.asp>.