# Eventizing Applications in an Adaptive Middleware Platform

# Eventizing Applications in an Adaptive Middleware Platform *

Andreas Frei, Andrei Popovici, and Gustavo Alonso
Department of Computer Science
Swiss Federal Institute of Technology Zürich
CH-8092 Zürich, Switzerland
{frei,popovici,alonso}@inf.ethz.ch

## ABSTRACT

Adaptive middleware is increasingly being used to provide applications with the ability to adapt to changes such as software evolution, fault tolerance, autonomic behavior, or mobility. It is only by supporting adaptation to such changes that these applications will become truly dependable. In this paper we discuss the use of event based systems as a platform for developing adaptive middleware. Events have the advantage of supporting loosely coupled architectures, which raises the possibility of orthogonally extending applications with the ability to communicate through events. We then use this ability to change the behavior of applications at run time in order to implement the required adaptations. In the paper we briefly describe the mechanisms underlying our approach and show how the resulting system provides a very flexible and powerful platform in a wide range of adaptation scenarios.

## 1. INTRODUCTION

Advances in hardware architectures and the widespread availability of wireless networks have radically changed the computing environments a software application must face during its active life time. A way to cope with the increasingly dynamic nature of the computing environments (e.g., mobile or pervasive computing) is to use adaptive software architectures. Of the several possibilities for implementing such architectures [9], event-based systems offer a wide range of advantages [11]. The main one is the high degree of decoupling between components. By using events as the way to communicate, components are independent of each other and can therefore change and evolve independently of other components. This property fits rather well with the need to cope with unexpected changes in the computing environment. One can treat changes as new types of events being thrown into the system and adaptation as the ability to dynamically react to or generate new events.

The type of changes and the range of adaptations we have in mind can be illustrated with a few examples. As a first example, consider changes in network technology that result in an increase in available bandwidth. Applications that today exchange few and very compressed events to avoid cluttering the network may have to cope later with a larger number of more complex events that are generated taking advantage of the additional bandwidth. Ideally, we would like older applications to be able to deal with the new stream of events without having to redesign or change them in a significant manner. One way to do this is to dynamically add a software layer that deals with the new events and acts as a translator for the older, less capable application.

A second example are changes in policy that may force applications to cope with new types of events not foreseen at the time they were designed. An automobile control system, for instance, may be required to generate an event when the combustion in the engine is less than perfect. The event can then be used to monitor the pollution level caused by the automobile. The ability to monitor the combustion is present in most modern cars but the software that turns the information into an event is probably not. Ideally, it should be possible to dynamically add the ability to generate such an event without having to change the software already installed in the automobile.

These examples are similar in that they require to dynamically extend already deployed and possibly already running applications. In this paper we outline an architecture that allows to eventize an application without a prior knowledge of events. An application designed and already running can be extended to publish relevant events and react to events produced by its surrounding infrastructure.

The architecture is based on the peer to peer concept of JXTA [10, 3] which allows to discover and activate new application extensions. At activation time of the application extension the underlying dynamic AOP system based on PROSE [15, 14], extends the running application with the new functionality. The dynamic AOP system is responsible for crosscutting the running application defined by an aspect inside the application extension. Once a crosscut occurs the application extension takes over the task for handling events like generating an appropriate event and publishing it. The underlying event system is then responsible for sending the event and handling the subscriptions.

The rest of this paper is structured as follows. Section 2

*The work presented in this paper was supported (in part) by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322.

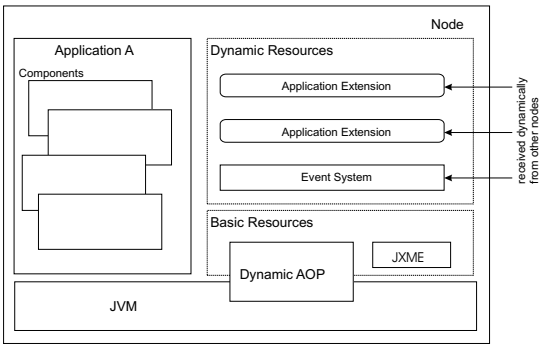


**Figure 1: System architecture; *a JVM, a set of basic resources, a set of dynamic resources, and the actual application***

discusses the system's architecture, including an explanation on how eventization of applications can be achieved with aspect-oriented programming and an event system. Section 3 gives some examples of how the adaptations can be done. Section 4 tests the viability of the approach we propose. Section 5 concludes with examples where the proposed architecture can be used.

# 2. SYSTEM ARCHITECTURE

## 2.1 Overview

Figure 1 depicts the basic architecture of the system. We assume there is an application running on the same JVM and that this is the application that will be adapted as needed. On top of the JVM, a first layer implements the *basic resources* of the system. These basic resources constitute the JXME peer to peer layer and include a platform for dynamic AOP. A second layer is devoted to the *dynamic resources* acquired at run time. As a first dynamic resource the event system is required which can be used by the application extensions to publish events and subscribe for events. Application extensions may include a dynamic AOP aspect and the behavior of the extension itself. The dynamic AOP aspect inserts crosscuts into the application at its activation. On the other hand the behavior of the extension is responsible for creating and publishing events and subscribing for relevant events and reacting accordingly.

The adaptations that we consider are of three generic types. The first type of adaptation consists of endowing an application with the ability to generate events (either because it did not generate any event at all or because it did not generate the necessary events). The second type of adaptation transforms applications into consumers of events (again, either because the application did not accept any events to start with or because it did not accept the necessary events). The third type of adaptation affects the loaded application extensions itself and therefore also the event system. This allows to extend dynamic resources itself for further changes.

The way the system works is as follows. The JXME layer gets informed when a new dynamic resource is available and loads the discovered application extension. When the application extension gets activated the AOP aspect is inserted into the AOP platform. The dynamic AOP platform intercepts the execution of the application and monitors their progress. Whenever it reaches selected points in the execution, the dynamic AOP platform redirects the execution to the appropriate application extension. Once the extension is executed, the control flow returns to the application that then resumes its work (exactly where and how work is resumed depends on the nature of the extension being executed).

The basic resources have been designed as a minimal required library when the application is started. The total size of this architecture is currently about 1 MByte.

## 2.2 Basic Resources

### Dynamic AOP Support

The basics for adaptation in our architecture is *dynamic Aspect Oriented Programming* [7, 5, 14, 12].

Aspect-Oriented Programming (AOP) [8] is a software design technique that allows the separation of orthogonal concerns within an application. These orthogonal concerns are then programmed as separate *aspects* rather than locating them in many different places in the code. The main advantage of AOP is precisely the possibility of abstracting out concerns that *crosscut* through the application (i.e., appear in many different places throughout the code). These aspects can then be treated as separate software modules, thereby increasing the modularity of the design. For instance, Zhang and Jacobsen [17] argue that aspect-oriented re-factorization can enhance the modularity of middleware and reduce structural complexity.

An aspect defines what to do (e.g., invoke an additional method) when a particular point is reached in the code (e.g., when invoking methods with certain signatures, when modifying a variable, etc.). Conventional AOP uses a *weaver* to add the aspect code to the base code of a program at compile time, e.g., AspectJ [16]. In dynamic AOP, the aspect code is added (*woven*) at run time by executing it whenever the specified point in the execution is reached. Aspects can also be dynamically withdrawn (*unwoven*) leaving the application in its original state.

In our system, we use PROSE [15, 14] as the platform for dynamic AOP. PROSE hooks into the JVM and intercepts method calls for the points in the execution where aspects are to be executed.

Figure 2 shows how PROSE is used to introduce extensions that deal with events. The figure assumes an application that is not aware of the events and depicts how extensions can be used to turn the application in a consumer and/or producer of events.

PROSE [15] uses the debugger interface for the interception of method calls. Therefore the application has to be run on a PROSE enabled JVM. Figure 2 shows PROSE first as an extension to the JVM and second as a library inside the basic resources which is used by an application extension.

### JXME Overview

The second basic resource required by our system is based on the peer to peer concept of JXTA [10, 3]. Since we are mainly interested in pervasive and mobile computing applications, we have chosen a decentralized solution unlike, e.g., Jini [1], where a centralized lookup service is responsible for distributing services. An independent lookup mechanism is a significant advantage in an ad hoc application scenarios where it is not feasible to have all the time a centralized server. Similarly, JXTA uses peer groups to combine peers

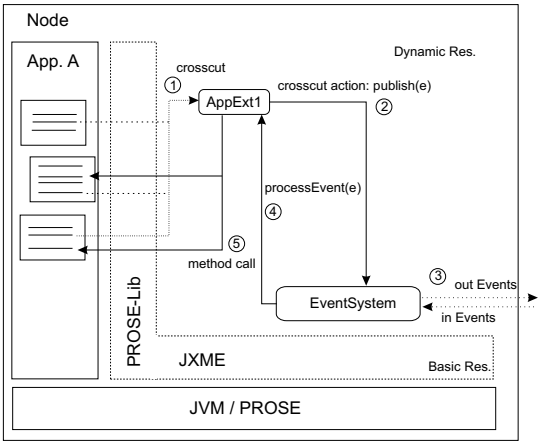


**Figure 2: Dealing with events through extensions; *dashed line: aspect call, dotted line: event call, solid line: method call***

with similar services or behavior. As such, a peer may reside in different peer groups at the same time. We use this functionality to apply extensions to entire groups rather than to individual nodes. Being able to do so is a significant advantage when dealing with realistic scenarios where several dozen devices are involved in the interaction.

As JXTA has been built for a wired Internet peer to peer network it is much too heavy for smaller devices. As we require a discovery mechanism only in a wireless environment a striped down JXTA has been implemented. This new JXME implementation uses multicast to send out messages whereas it is listening at the same time for messages. This basic messaging allows to advertise services by a service provider. Any other node in the multicast domain may then discover such service advertisements and download the service from the provider. With this basic messaging we are able to download as a first service the event system.

By implementing the JXME layer directly on top of UDP we are independent of a remote procedure call infrastructure like RMI or CORBA. This leads to a smaller infrastructure and the possibility to extend the concept to further environments not based on the chosen RPC infrastructure.

## 2.3 Dynamic Resources

An application extension found through JXME is inserted as a .jar file containing a manifest file for the extensions meta information. To allow dynamic resources to be unloaded again every application extension is loaded in its own classloader. After removal of an application extension its classloader gets garbage collected. Extensions can use classes from other dynamic resources by specifying in their manifest a dependency list to other extensions. This allows for example the *AppExt 1* in Figure 2 to use the *EventSystem* extension. The activating class is specified through the *Main-Class* entry in the manifest. By taking Java's jar specifications for a manifest to describe the application extension the extension can be activated inside JXME with the main-classes main method.

The event system designed as a dynamic resource itself has to be loaded and activated by a node before eventization is required by application extensions. The application extensions are then able to use the event system to publish and subscribe to events.

## Event System Implementation

The event system we use is based on a publish and subscribe model [4, 13, 6, 7]: a producer node publishes an event and a consumer node may subscribe for this event and be notified of the occurrence of the event. TPS [2] is a type based publish and subscribe model built on top of JXTA [10, 3]. In our implementation we use a similar approach to TPS. As we only rely on the JXME we have implemented our own event system on top of this discovery and messaging layer.

The event system interface is shown in the code example below. This is also the interface which can be used by the application extensions.

```
1  interface Event {
2    void init(Message msg);
3    Massage toMessage(Class clas);
4  }
5
6  interface Filter {
7    boolean matches(Event event);
8  }
9
10 interface EventListener {
11   void processEvent(Event e);
12 }
13
14 interface EventSystem {
15   void subscribe(Filter filter, EventListener listener);
16   void unsubscribe(Filter filter);
17   void publish(Event event);
18 }
```

The implementation of the *EventSystem* interface uses again the JXME messaging layer to send an event to other nodes. Before sending an event it gets serialized into the JXME message format which is then sent as a multicast datagram packet. On the other nodes the message gets deserialized into the event type. The subscription to events is done through an event filter following the type- and attribute-based subscription model introduced in [13, 2].

As we are targeting a decentralized infrastructure each node manages its own subscriptions. A subscription is therefore done only locally and does not lead to any traffic on the wireless channel. On the other hand when publishing an event it has to be sent to all nodes where it has to be checked for a registered subscription.

With some specific filtering attributes we are able to mimic the peer group concept defined by JXTA. New application extensions or messages can then be published by including the peer group attribute in the event.

# 3. EXAMPLES OF ADAPTATION

## 3.1 Producers of Events

Transforming an application into an event producer requires two things. The first is to detect the application state that should lead to the generation of an event. The second is to actually publish the event using whatever event system is available. We deal with these two problems using a single extension.

Once the designer has identified when events are to be generated (e.g., after a variable has been updated, when a method is invoked, etc.), an AOP aspect is created that traps that particular situation. Beside the aspect the application extension also contains the logic necessary to publish the corresponding event. In Figure 2, step 1 indicates the

points in the execution that are relevant for the generation of the events. When these points are reached, the extension is invoked and the event is passed on (step 2) to the underlying event system which will then publish it (step 3).

As an example, consider a calendar application running on a PDA. The following PROSE extension can be used to automatically generate an event for every meeting entry the user makes in the calendar. This way, other calendars can be synchronized and notifications being sent to other persons or applications (e.g., to reserve a meeting room).

```
1  class AddMeetingEventsAsProducer extends
2  Pointcut {
3    // mechanism for publishing events
4    EventSystem eventSystem;
5
6
7    // definition of the pointcut where events should be fired
8    PointCutter pointCutter() {
9      return Executions.before().AND(Within.methods("*"));
10   }
11
12   void METHOD˙ARGS(MeetingDates mdates, REST arguments) {
13      Event e = new Event(MeetingDate);
14      eventSystem.publish(e);
15   }
16 }
```

This aspect defines the operation that has to be intercepted (line 9). As explained, it corresponds to state changes in the MeetingDates class. Line 12 defines the extension to be executed. This extension creates an event object with the information about the meeting and then calls the EventSystem component to publish the event.

## 3.2 Consumers of Events

Transforming applications into consumers is done by including consumer behavior in an application extension. The extension does two things. First, it subscribes to events of interest. Then, upon arrival of an event, it calls the corresponding method(s) of the underlying application. The procedure is the reverse of the one used for event producers. In Figure 2, when an event arrives (step 3), the event system notifies the extension (step 4) which will then invoke the methods in the application (step 5).

Following the calendar example, this extension can be used to automatically generate entries in the calendars of users as a result of an entry being made in a particular calendar. Such an extension looks as follows:

```
1  class CalendarBecomesEventConsumer
2    implements EventListener {
3
4    EventSystem eventsystem;
5
6    { eventSystem.subscribe(MeetingDateFilter, new CBEC()); }
7
8    // action when events are received
9    void processEvent(MeetingDate received) {
10     // add the date to the calendar
11   }
12 }
```

When the extension is inserted it subscribes in the initialization phase 6 for MeetingDate events. When such an event arrives, the processEvent method is executed. This method inserts the date in the calendar and may notify the user of the new appointment or of potential conflicts.

# 4. BENCHMARKING

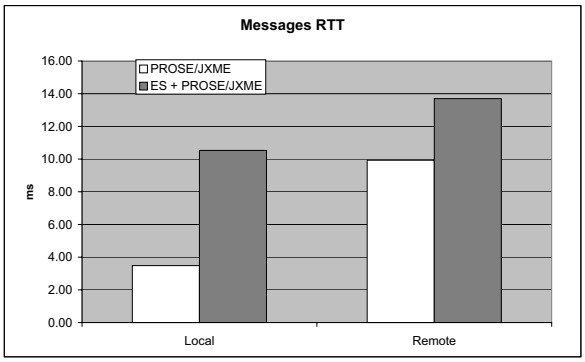


**Figure 3: Local, 2 Node Benchmark; *Messages Round Trip Time (RTT)***

In order to test the viability of the approach, we have conducted a number of experiments with our architecture.

We test the different building parts of the architecture and the scalability of the whole architecture for messaging and the event system in a wireless environment. The overhead of the dynamic AOP system has been measured in [15].

The local benchmarks have been measured on a IBM Laptop A31 running Red Hat 9.0 which is our master laptop. For remote benchmark machines we used IBM Laptops R32 running Red Hat 7.3 with built in wireless cards. In a first benchmark we measured the behavior of the architecture on the same node and on two different nodes with a wireless connection in ad hoc mode. In the scalability benchmark we increased the involved nodes up to 9 and analyzed the behavior with various wireless parameters.

The measurements in the first benchmark have a standard deviation of less then 1%. In the scalability benchmark we achieved less then 5% deviation as during the access point test the access point has been used by other people.

## 4.1 Local, and 2 Node Benchmark

In the first measurement section the benchmarking parts of the architecture have been categorized:

**PROSE/JXME** As PROSE and JXME run as a basic resource in every node we measured the time of sending messages between two different nodes. In a first test the two different nodes were running on the same machine. The second test sent messages over a wireless connection in ad hoc mode.

**Event System (ES)** As the event system is used by any application extension we included an additional test to measure our type- and attribute-based subscription mechanism.

First, we analyze the overhead for the application when started with the basic resources without any application extensions running. As PROSE uses the Java Debugger Interface on older JVMs a significant performance overhead occurs as the whole application runs in debugger mode. With the newer JVM 1.4 debugging is possible with full speed and

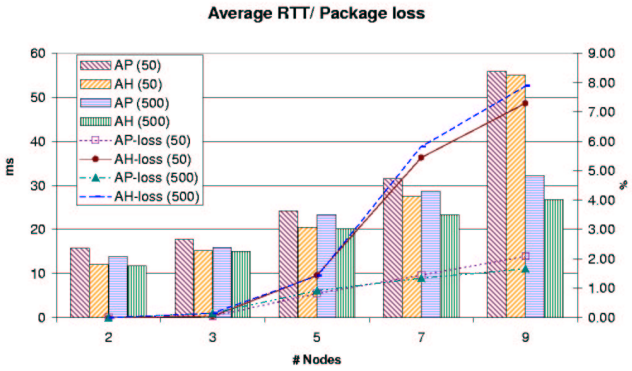


**Figure 4: Scalability Benchmark**

no overhead occurs when no aspects are inserted. The second basic resource, JXME, does not lead to a slow down of the application as no messages have to be processed.

Second, we measured JXME and the event system extension (Figure 3). It shows that the round trip time (RTT) of a basic message between two different JVMs on the same machine takes on JXME 3.5 ms. Whereas on two different machines and over wireless it takes 10 ms. When the event system is included an event round trip takes on the same machine in average 10.5 ms which is quite significant more compared to a basic message. This increase goes back to the serialization and deserialization steps of an event type into a basic message and finally on cost of the filtering mechanism involved in both JVMs. Finally, an event round trip over wireless in ad-hoc mode takes 13.7 ms.

## 4.2 Scalability Benchmark

In the scalability benchmark we analyze the behavior of having many nodes in a wireless infrastructure publishing events and subscribing for events. The infrastructure represents a star topology where the master sends events and the other nodes respond immediately. The system architecture remains the same in all tests but with increasing responding nodes. The master laptop sends out messages and events with a delay of 50 ms and 500 ms. Beside the average round trip time the package loss is included in Figure 4. We also compared the access point (AP) infrastructure mode with the ad hoc (AH) infrastructure less mode.

Figure 4 shows that with increasing number of nodes, the round trip time increases. As we measured in a star topology the master node becomes for every sent message as many messages as clients. The response messages are queued until the measuring thread is able to process them and stop the clock. When comparing the wireless infrastructure mode it can be seen that the round trip time in ad-hoc mode is in average 14% less than in access point mode. As in access point mode a message/event is sent over an access point it is apparent that the delay gets bigger. The drawback is the package loss, which increases up to 8% in ad hoc compared to 2% in access point mode. From this 9 node scalability benchmark we recognize a strong increase of round trip time when decreasing the time interval of sending messages. The system will therefore currently not scale for environments with a lot of nodes sending messages with a small time interval. Switching the wireless ad hoc infrastructure mode to an access point improves the package loss for the 9 node infrastructure by 75%. A combination of access point mode and larger time intervals may scale better for more nodes.

# 5. CONCLUSION AND FUTURE WORK

In this paper we have outlined an adaptive middleware platform based on events. The platform treats events and the management of events as aspects that can be changed at run time in response to new requirements or necessary adaptations. We are currently in the process of completing the implementation of the platform and exploring more advanced forms of adaptation, including dynamic changes to the event management system itself. Furthermore the platform runs on resource constraint devices like iPAQs where the implementation is still to heavy and where we will take advantage of lighter virtual machines.

We are also using the first prototype of the platform in a variety of applications. One of them is a cooperating robots scenario where autonomous robots with wireless communication capabilities coordinate their movements through the exchange of events. Through the adaptive platform described in the paper, we can separate the software that controls the movement of the robot from the software that deals with events and the coordinated behavior. This is a significant advantage over existing designs as it greatly simplifies development, maintenance, and offers much more flexibility in terms of adaptation. The management of events and the coordinated behavior are treated as dynamic extensions and can be changed at any time. For instance, a set of extensions implement a *train formation* movement where all robots follow a leading robot. The leading robot is remotely controlled with a joystick and its movements are communicated to all other robots via events. These events are interpreted by extensions on each robot that than control the movement of the robot as a function of its position in the formation and the movement of the leader.

The advantage of using this software architecture is that we can change at any time the extensions and completely modify the behavior of the system. As an example, the train formation extensions can be exchanged for a *line formation* extension where all robots move parallel to each other. The extensions can also be exchanged for new ones that implement more sophisticated behavior by generating new types of events such as robots randomly leaving the formation, speed control events, movement filters that prevent the formation from entering certain areas, etc.

# 6. REFERENCES

[1] K. Arnold, A. Wollrath, B. O'Sullivan, R. Scheifler, and J. Waldo. *The Jini Specification*. Addison-Wesley, Reading, MA, USA, 1999.

[2] Sébastien Baehni, Patrick Th. Eugster, and Rachid Guerraoui. Os support for p2p programming: a case for tps, 2002.

[3] Daniel Brookshier, Darren Govoni, and Navaneeth Krishnan. *JXTA: Java P2P Programming*. SAMS, March 2002.

[4] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 27(9), 2001.

[5] R. Douence, P. Fradet, and M. Sdholt. A framework for the detection and resolution of aspect interactions. In *Proce. of the ACM SIGPLAN/SIGSOFT Conf. on Generative Programming and Component Engineering (GPCE'02)*, October 2002.

[6] Ludger Fiege, Mira Mezini, Gero Mühl, and Alejandro P. Buchmann. Engineering event-based systems with scopes. In B. Magnusson, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2374 of *LNCS*, pages 309–333, Malaga, Spain, June 2002. Springer-Verlag.

[7] M. Haupt, M. Mezini, M. Cilia, and A. P. Buchmann. Towards Event-Based Aspect-Oriented Runtime Environments. Technical Report TUD-ST-2002-01, Software Technology Group, Darmstadt University of Technology, Alexanderstrasse 10, 64289 Darmstadt, Germany, 2002.

[8] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, New York, NY, June 1997.

[9] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile Application-Aware Adaptation for Mobility. In *Sixteenth ACM Symposium on Operating Systems Principles*, pages 276–287, Saint Malo, France, 1997.

[10] Open Source. Project jxta web site. http://www.jxta.org, 2001.

[11] J. Andrs Daz Pace and Marcelo R. Campo. Analyzing the role of aspects in software design. *Communications of the ACM*, 44(10):66–73, 2001.

[12] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A Flexible Solution for Aspect-Oriented Programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, pages 1–24, Kyoto, Japan, September 2001. Springer Verlag.

[13] P. Pietzuch and J. Bacon. Hermes: A distributed event-based middleware architecture, 2002.

[14] A. Popovici, G. Alonso, and T. Gross. Just in time aspects: Efficient dynamic weaving for java. In *2nd Intl. Conf. on Aspect-Oriented Software Development, Boston, USA*, March 2003.

[15] A. Popovici, T. Gross, and G. Alonso. Dynamic Weaving for Aspect Oriented Programming. In *1st Intl. Conf. on Aspect-Oriented Software Development, Enschede, The Netherlands*, April 2002.

[16] Xerox Corporation. The AspectJ Programming Guide. Online Documentation, 2002. http://www.aspectj.org/.

[17] Charles Zhang and Hans-Arno. Jacobsen. Quantifying aspects in middleware platforms. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 130–139. ACM Press, 2003.