An extensible framework for middleware design based on concurrent event-based AOP

Edgar Marques, Luís Veiga, Paulo Ferreira
Distributed Systems Group
INESC-ID/Technical Univ. of Lisbon
Rua Alves Redol N. 9
1000-029 Lisbon
Portugal
emarques@gsd.inesc-id.pt, {luis.veiga, paulo.ferreira}@inesc-id.pt

ABSTRACT

Middleware simplifies application development by encapsulating common low-level concerns in modular reusable components. However, the traditional methods of software decomposition fail to properly encapsulate so-called *cross-cutting concerns* thus leading to scattered (and sometimes repetitive) code and difficult to maintain designs.

Aspect-Oriented Programming (AOP) aims to solve these issues by encapsulating such code within reusable components called *aspects*. However, current AOP implementations suffer from restrictive programming models leading to limited aspect reusability.

In this paper we present a new Java framework for middleware design and development based on *Concurrent Event-based AOP*. We focus on simplicity, generality and reusability. The programming model is based on *Attribute-Oriented Programming*. Aspects are declared and used by writing plain Java code and tagging it with plain Java annotations. The framework is small and can be easily extended to build more sophisticated frameworks targeting different kinds of devices. We describe the implementation of an initial prototype and evaluate its performance.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming, Parallel programming*; D.2.8 [**Software Engineering**]: Metrics—*Performance measures*; D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Frameworks*; D.3.4 [**Programming Languages**]: Processors—*Code generation*

General Terms

Design, Performance, Measurement

ARM 2010, November 30, 2010, Bangalore, India.

Keywords

Concurrent Event-based Aspect-Oriented Programming, Attribute-Oriented Programming, source-level weaving, program transformation, Java framework

1. INTRODUCTION

As the users of modern mobile devices (such as smartphones and netbooks) demand for richer, more complex applications that enable them to work anywhere, any time, on a daily basis, the limitations of such devices (e.g. variable bandwidth, limited battery capacity, etc.) become an obstacle that considerably restrains the kind of tasks these devices can perform. Therefore, modern mobile applications must rely on *distribution* (i.e. delegate data and tasks to other network nodes) and *adaptation* (i.e. changing one's own behaviour during runtime) in order to better achieve the user's current objectives and/or better manage resources available locally and remotely.

However, providing support for these features is a daunting task, which requires the programmer to directly deal with low-level issues such as data management, security, awareness, among others. Several middleware solutions provide these features in a modular, reusable fashion, thereby delivering considerable savings in development effort, time and cost. However, integrating an application with the supporting middleware constitutes by itself a considerable effort, which usually requires the programmer to adapt the application to fit a particular programming model. In addition, some features may be employed in several business logic modules thus becoming scattered throughout the source code, leading to a complex design which is difficult to understand and maintain.

In this paper we present a new framework for the Java language which is based on the *Concurrent Event-based Aspect-Oriented Programming* (CEAOP) paradigm [11, 10]. This framework was designed as a support tool for middleware design and development but its design is generic enough for it to be of practical use in other areas as well.

The programming model of our framework is based on the *Attribute-Oriented Programming* paradigm. Aspects, as well as pointcuts, are expressed as *plain old Java objects* (POJOs) which are then tagged with a set of predefined Java *annotations*. To improve aspect reusability, we have separated the declaration of a new aspect from the declaration of its pointcuts. This way programmers can use aspects written by others and aspects can be distributed

even in binary form. Pointcuts can be modified in runtime thus giving the programmer greater control over application runtime behaviour. To keep our framework small, simple and general-purpose, we deliberately avoided addressing such issues as aspect composition and coordination. Instead, our framework provides extension mechanisms for creating higher-level frameworks on top of it and delegate to these such issues. Also, our programming model is simple and generic enough to be implemented on different platforms in a variety of ways, according to the capabilities or constraints of the underlying running environment.

This paper is organized as follows. We compare our solution with other related work in Section 2. Section 3 describes the architecture of our framework and its programming model. We discuss the implementation of our prototype in Section 4. In Section 5 we give an evaluation of our prototype and propose some optimizations. Finally, we describe our plans for extending the framework and its use in real-world applications and give some concluding remarks on Section 6.

2. RELATED WORK

AspectJ [13] is one of the oldest and well-known aspect languages and it helped to bring AOP to the mainstream. AspectJ is an extension of the Java language which defines a special syntax for declaring aspects. The first versions of AspectJ featured compile-time source code weaving and bytecode weaving. It later merged with AspectWerkz [1] which brought load-time weaving as well as AspectWerkz's annotation style to the language. The support for load-time bytecode weaving allows AspectJ to weave advice code into classes available only in binary (.class) form.

Our work differs from AspectJ in some key points. First, our focus is helping the programmer to write aspects that can be reused even if available only in binary form (for instance, middleware libraries) instead of applying aspects to precompiled code. This means that pointcut declarations must be separated from aspect declarations. We therefore define two different roles: the role of the *aspect programmer*, i.e. the programmer who writes an aspect and defines both the structural and behaviour changes it encapsulates (introductions and advices), and the role of the *application programmer* or *client programmer*, i.e. the programmer who uses the aspect and specifies in which circumstances should the aspect apply its advice (by defining pointcuts). Second, AspectJ's aspect model is a sequential model, i.e. aspect advice is executed sequentially with regard to other aspects and the target (advised) code. Our work is based on a model of concurrent aspects and is inspired by previous work by Douence et al. [11, 10].

Like AspectJ and other AO languages and frameworks [2, 6, 4], our programming model is based on *Attribute-Oriented Programming*. However, these approaches still depend on special syntaxes or complex models of composition or inheritance in one way or the other. For instance, although AspectJ allows the programmer to write aspects using only annotations (using the annotation style adopted from AspectWerkz), pointcut declarations still need to be expressed as strings written in a special regular expression language. On their turn, PostSharp [4] and JAC [17] both require the programmer to inherit from a special base class. Such restrictions make it harder to reuse existing code and require the programmer to restructure his application thus imposing an additional burden. Our solution does not make any such requirements. The programmer uses only plain Java annotations to declare and use aspects in his code. Whenever annotations are not expressive enough to specify certain requirements (such as pointcuts), we allow the programmer to express such requirements through predicates written as plain Java methods which are tagged using plain Java annotations. Although this results in more verbose code, it has the advantages of being more flexible and giving the programmer greater control over application behaviour.

The term *Dynamic AOP* is employed to denote the runtime weaving of advice code [1] or the runtime registering of *proxies* or *interceptors* [2, 3]) for a particular kind of event. Dynamic AOP is a powerful tool that allows the programmer to write applications whose behaviour changes in runtime and is supported by many AO tools including JAC [17] and JAsCo [18]. Like JAC and JAsCo, our framework treats aspects as loosely-coupled components which can be inserted or removed at runtime.

Dinkelaker et al. [8] proposed an aspect runtime with support for a meta-aspect protocol called POPART. POPART allows programmers to extend and modify the semantics of the aspect language at runtime without any changes to the aspects themselves. Our solution also allows the specification of meta-aspects but without requiring any special support from the framework as we show in the next sections.

3. ARCHITECTURE

The framework is built as a foundation for middleware development, as shown in Fig. 1. At the middleware level, each feature (a *cross-cutting concern* such as security or data management) is encapsulated in a reusable component called an *aspect*. Aspects can run concurrently with regard to the base program (advised code) and to each other. Communication between aspects and the base program is based on message passing (i.e. event-based). Aspect deployment, process/thread management and event handling are the core concerns of our framework, on top of which the middleware sits.

Our framework runs on top of standard Java VM. In order to keep the framework small, simple and generic, issues such as concurrency control, aspect composition and others are not addressed in its core. We consider these issues to be cross-cutting concerns of aspects or *meta-aspects* [8] (i.e. aspects of aspects) and we leave them to be addressed by specialized, higher-level frameworks sitting between the core framework and the middleware. At the top level we have the user applications which make use of the middleware libraries as well as of services provided by the levels below. Our framework is designed to be generic enough to support different implementations targeting different kinds of devices.

3.1 Concurrent Event-Based AOP

Aspects are implemented as *loosely-coupled*, *concurrent*, *isolated entities* that respond to events which are raised at specific points of interest within the execution flow called *join points*. The specific set of join points that a particular aspect is interested in is specified by a query which is referred to as a *pointcut*. Aspects are executed concurrently thus allowing applications to leverage, if available, the multiprocessing capabilities of the underlying platforms. Since aspects are weakly coupled, they are easily replaced at run-

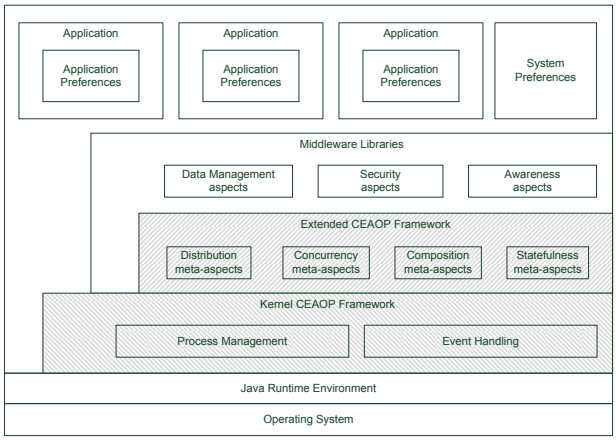

Figure 1: Global system architecture

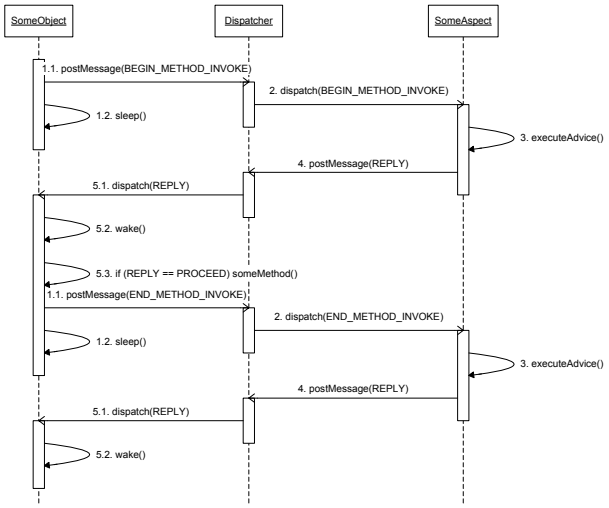

Figure 2: Event raising at a method call and advice execution

time which makes it easier and simpler to express changes in application behaviour. Finally, since each aspect isolates a specific concern from the rest of the application, applications also benefit from improved robustness.

When a join point (e.g. a method call) is reached in the execution flow of a program, events are raised right before and just after join point execution. This allows listening aspects to execute advice code before, around (i.e. instead of) and after the join point. Events are not delivered directly to listening aspects by the advised program. Instead, they are delivered to a *dispatcher thread* that is responsible for delivering the events to the appropriate aspects. When an aspect is activated (i.e. instantiated) it registers itself with the dispatcher thread stating which kind of events it is listening for. Decoupling event senders from event receivers allows for multiple implementations of the framework based on different message-passing mechanisms. The base program, the dispatcher and the listening aspects could be implemented as different threads within the same process communicating through shared objects or as different processes communicating through some inter-process communication mechanism.

Fig. 2 illustrates the sequence of actions which take place for the particular case of execution of a method call. For the sake of brevity the figure only shows the interactions between one application object and one aspect instance.

When an event is raised, such as a method call, the following actions take place:

1. The current thread (in which the application object is being executed) places a message in the message queue of a dispatcher thread (that will deliver the message) (step 1.1) and goes to sleep (step 1.2). In this particular example the message being sent is the `BEGIN_METHOD_INVOKE` message which states that a method is about to be called on the application object.

2. The dispatcher retrieves the message from its queue and delivers it to its target aspects (step 2).

3. Each aspect executes its advice code (step 3).

4. Upon completion each aspect replies back by posting a message to the dispatcher (step 4).

5. Upon receiving all replies the dispatcher sends back reply to the first thread (step 5.1) that will then resume execution (step 5.2). If the dispatcher replies back with the message `PROCEED` then the method call can take place as normally, otherwise the application should go around it as the listening aspects have already performed alternative actions in its place (step 5.3).

6. The previous sequence of events is repeated after the method call taking place (or being skipped) with the exception of step 5.3 and the `END_METHOD_INVOKE` message being dispatched instead.

The described sequence of steps ensures that aspects are executed synchronously with regard to the base program. However, an aspect can be executed asynchronously by running its advice code in a background thread and replying back before advice execution finishes.

3.2 Attribute-Oriented Programming

Aspects are regular Java classes which are tagged with pre-defined Java annotations. The following example shows the declaration of an aspect for logging application events. For the sake of brevity we do not show the full set of annotations and their corresponding elements.

```
@Aspect(Name="Logged",
        Elements={@Element(Name="LogFile",
            Type=String.class)})
public class Logger {
  ...
  @Member
  private int eventCount = 0;
  ...
  @Before(Actions={ActionType.METHOD_INVOKE},...)
  public void log(...) {...eventCount++;...}
  ...
}
```

The `Logger` aspect class is declared by tagging it with the `@Aspect` annotation. This annotation tells the framework's

build tool to extend the aspect class with the necessary boilerplate code at compile time. The build tool will also generate a new annotation type called `@Logged` which has an annotation element called `LogFile` of type `String`. This annotation will then be used by programmers who wish to use the `Logger` aspect class in their applications.

Introductions (i.e. inter-type declarations) and advices are declared by tagging type members with the `@Member` and `@Before` annotations respectively. In the above example the `Logger` aspect class declares a private instance field called `eventCount` that will be added at compile time to every type which uses `Logger`. It also declares an advice method called `log` that, among other things, keeps count of every method call that occurs for each instance of the type being logged.

The following example shows how the aspect class declared above can be used in an application.

```
@Logged(LogFile="log.txt")
public class SomeClass {
  ...
  @Pointcut(Aspects={Logger.class})
  public boolean predicate(AdviceType advice,
   ActionType action, ...) {...}
  ...
}
```

In this example the class `SomeClass` is tagged with the `@Logged` annotation which states that all method calls on instances of `SomeClass` should be logged to file `log.txt`.

Unlike other approaches, our framework separates pointcuts declarations from aspect declarations. This improves reusability, allowing the application programmer to include in his code aspects written by other programmers without having to restructure the application to fit a particular model. The aspect programmer therefore only specifies the aspect's join point model, i.e. the kinds of join points to which it can apply its advice (such as method calls, field accesses, etc.). By default an aspect will apply its advice to all join points of the kinds that were specified by the aspect programmer.

However, the application programmer may wish to restrict advice execution to a small set of join points or even to a small set of instances depending on the circumstances. He can achieve this by declaring pointcuts that specify whether a particular join point should trigger advice execution or not. Pointcuts are regular Java methods which are tagged with the `@Pointcut` annotation as shown in the above example. The application programmer must specify the aspect classes the pointcut applies to. A pointcut method defines a predicate which returns true or false (whether the advice should be executed or not) depending, among others, on the kind of advice to be applied (before, around or after) and the kind of join point (method call, field access, etc.). Since a pointcut is a simple instance method, the programmer can easily dynamically modify application behaviour simply by letting the pointcut return a different value based on instance state or by resorting to more elaborate means such as the Strategy design pattern [12].

3.3 Extending the framework

Our extension model is based on the notion of *meta-aspects* [8], i.e. aspects that encapsulate cross-cutting concerns of other aspects. Meta-aspects intercept the advice execution of other aspects. A meta-aspect can allow advice execution to proceed as normal or delay it, override it or even prevent it. After execution, it can also delay the delivery of the reply message or modify it.

Because in our framework advice execution is performed as a simple method call and aspects are ordinary classes, it follows that meta-aspects are simply regular aspects listening to method calls of other classes who happen to be also aspects.

The framework described in this paper is only a small part of a greater whole and by itself does not satisfy all the needs of real world applications. We now discuss how full-fledged frameworks providing for those needs are built on top of ours.

Stateful aspects. Stateful aspects [7, 9] are aspects that trigger not on a single join point but instead on a sequence of join points. The programmer can easily implement a stateful aspect as a *finite state machine* (FSM) whose state transitions are triggered by events raised at particular join points. Different advice code could be executed according to the FSM's current state (another implementation of the Strategy design pattern).

However, implementing a FSM is a tedious, error-prone, repetitive task that places additional burden on the programmer. To facilitate the development of stateful aspects, FSMs that capture complex sequences of events can be implemented as meta-aspects which prevent the delivery of events to stateful aspects until the goal state of the FSM is achieved.

Aspect composition. Aspect composition [14, 9, 15] can be implemented using meta-aspects which redirect intercepted events and replies to other aspects for pre-processing and post-processing respectively.

Concurrency control. A set of concurrently executing aspects can be coordinated by having a meta-aspect which intercepts events and replies and then places some on hold while others not according to some criteria.

Distributed aspects. A naïve approach for implementing distributed aspects [17, 16, 15] would be to have a meta-aspect that broadcasts intercepted events on the network and which dispatches received broadcasts.

4. IMPLEMENTATION

Our current prototype is based on compile-time source-weaving. We have implemented a build tool that parses program source code at compile-time looking for code tagged with our predefined annotations and generates the respective boilerplate code. Code generation is a multi-pass process. After each code expansion the code is parsed again in order to identify new points of expansion. The process goes on until no further expansion is possible. Our build tool is based on the RECODER [5] framework for Java source code metaprogramming. However, our programming model also allows for different implementations based on load-time bytecode-weaving.

The following example shows the program transformation pattern for a method call join point.

```
boolean _goAround000001 =
  beginMethodInvoke(HelloWorld.sayHello,
```

```
    myHelloWorld, {"<My name>"});
Object _returnValue000001;
if(!_goAround000001)
  _returnValue000001 =
    myHelloWorld.sayHello("<My name>");
endMethodInvoke(HelloWorld.sayHello,
  myHelloWorld, {"<My name>"},
  _returnValue000001);
```

In this example the method `sayHello` is being called on an object called `myHelloWorld` which is an instance of class `HelloWorld`. The method receives a single argument of type `String` which in this particular case has the value `"<My name>"`. The build tool has injected two method calls (`beginMethodInvoke` and `endMethodInvoke`) around the call to method `sayHello`. These method calls are the ones that will raise the events before and after the method call is executed by sending messages to the dispatcher thread.

Each of these methods receives as arguments the type of the object and the method being called on it, a reference to the object itself and an array containing the values of the arguments passed to the method being called. In the case of the `endMethodInvoke` method call, the return value of the method that was just called is also passed along. In this case, because the return value of `sayHello` was not being stored, the build tool generated a new variable for holding that value. These arguments will be sent to all listening aspects as part of the message sent and will be available to advice code.

The `beginMethodInvoke` method returns a boolean value that states whether the program should go around (i.e. skip) the method call. We simply assume that all aspects listening for this particular event will agree on whether the program should skip or not the method call. It is up to the programmer to ensure correct behaviour either manually or by using a higher level framework which features mechanisms for preventing and/or dealing with aspect interference.

Similar patterns are applied to other kinds of join points. Because the build tool needs to insert code before and after each join point, nested expressions in the source code will be flattened and temporary variables for holding intermediate values will be generated.

For the sake of brevity and also because the generated code depends on each particular implementation of the framework, we do not show the full program transformation patterns for aspect classes and the implementation of the dispatcher. Suffices to say that our current prototype executes the main program, the dispatcher and all aspects as different threads running in the same process which communicate through shared objects. The dispatcher thread runs with a higher priority than the rest to ensure that all aspects are notified in the least time possible.

5. EVALUATION

The time overhead introduced by the execution of our framework is mainly related to process/thread creation and management, event dispatching and execution of advice code. The process/thread creation and management overhead should be mitigated by the use of process/thread pools. Also process/thread creation and termination (associated with dynamic activation and deactivation of aspects) should not be a frequent operation in most cases.

We performed a synthetic micro-benchmark of our initial prototype. The purpose of the benchmark was to measure how the event dispatching and advice execution overhead scales when the number of concurrent aspects is increased and to compare it with the advice execution overhead of a sequential implementation. We focused solely on CPU-bound operations as I/O-bound operations don't scale well and therefore in the benchmark each aspect executes a set of floating-point operations which is the same for all aspects. We simulated the sequential execution of all aspects by running the set of operations in a loop. The results, which were obtained on a dual-core equipped computer, are shown in Fig. 3.

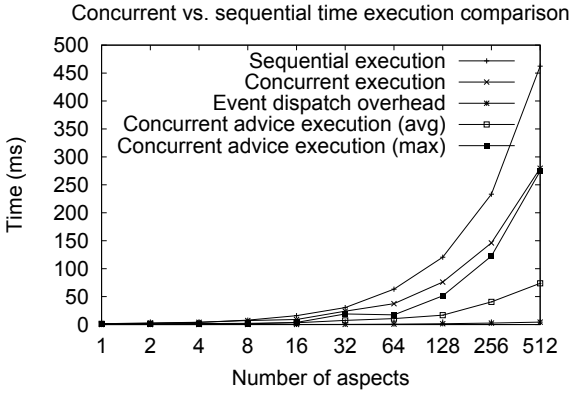

Figure 3: Time overhead for advice execution on a join point

The graphic shows the total time (in milliseconds) it takes to execute advice code for all aspects interested in a joint point when that join point is reached. As expected, the time overhead scales linearly with the number of aspects, doubling as the number of aspects doubles, for both implementations. The benefits of executing aspects concurrently can be seen as the time overhead for our prototype is reduced almost by half in a dual-core machine. However, due to the overhead of process/thread management and event dispatching, the execution time is still well above 50% of the time taken by the sequential implementation. The event dispatch overhead is shown to scale linearly as well, but is clearly dominated by the advice execution time as the number of concurrent aspects grow. Predictably, the average time it takes for an aspect to execute its advice code (measured as the time elapsed between an event being received and the completion of all operations performed in response to it) also increases linearly with the total number of concurrent aspects, as each aspect has to wait for other aspects scheduled to run first to finish executing (remember that in our prototype the dispatcher thread runs with higher priority which means that all aspects are notified before any begins executing). This is demonstrated by the maximum time it takes an aspect to execute its advice code, since the aspect which takes the longest time will be invariably the last one to execute, thus its execution time will approach the total execution time.

6. FUTURE WORK AND CONCLUSIONS

We have proposed a new framework for middleware design based on the Concurrent Event-based Aspect-Oriented

paradigm. Compared to other AO languages and frameworks, ours is focused on simplicity and ease of use. Aspects are declared and integrated in applications using solely plain Java annotations. Our framework can be extended by implementing meta-aspects that modify the semantics of other aspects.

The work presented is still at an initial stage and further analysis is still required in order to assess the scenarios to which our approach is best suited for. We intend to focus on scalability issues such as the execution of I/O-bound operations in advice code, the scalability of the event dispatch mechanism and the overlapping of both advice execution and event delivery for better performance (the latter would be particularly useful in distributed environments).

We also plan to focus on the implementation of higher level frameworks and address issues such as distributed aspects and concurrency control. These frameworks will then be used to conduct research in the implementation of middleware solutions for adaptability and context-awareness in distributed mobile applications and software development for application servers in cluster / farm infrastructures.

7. REFERENCES

[1] Aspectwerkz. `http://aspectwerkz.codehaus.org/`.

[2] Jboss aop. `http://www.jboss.org/jbossaop`.

[3] Joyaop. `http://joyaop.sourceforge.net/`.

[4] Postsharp. `http://www.postsharp.org/`.

[5] Recoder. `http://recoder.sourceforge.net/`.

[6] Spring. `http://www.springsource.org/`.

[7] T. Cottenier, A. van den Berg, and T. Elrad. Stateful aspects: the case for aspect-oriented modeling. In *AOM '07: Proceedings of the 10th international workshop on Aspect-oriented modeling*, pages 7–14, New York, NY, USA, 2007. ACM.

[8] T. Dinkelaker, M. Mezini, and C. Bockisch. The art of the meta-aspect protocol. In *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 51–62, New York, NY, USA, 2009. ACM.

[9] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 141–150, New York, NY, USA, 2004. ACM.

[10] R. Douence, D. Le Botlan, J. Noyé, and M. Südholt. Concurrent aspects. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 79–88, New York, NY, USA, 2006. ACM.

[11] R. Douence, D. Le Botlan, J. Noyé, and M. Südholt. Towards a model of concurrent aop. Int. WS on Software Engineering Properties of Languages and Aspect Technologies (SPLAT'06), March 2006.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with aspectj. *Commun. ACM*, 44(10):59–65, 2001.

[14] S. Kojarski and D. H. Lorenz. Pluggable aop: designing aspect mechanisms for third-party composition. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 247–263, New York, NY, USA, 2005. ACM.

[15] R. Mondejar, P. Garcia, C. Pairot, P. Urso, and P. Molli. Designing a distributed aop runtime composition model. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 539–540, New York, NY, USA, 2009. ACM.

[16] L. D. B. Navarro, M. Südholt, W. Vanderperren, B. De Fraine, and D. Suvée. Explicitly distributed aop using awed. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 51–62, New York, NY, USA, 2006. ACM.

[17] R. Pawlak, L. Seinturier, L. Duchien, G. Florin, F. Legond-Aubry, and L. Martelli. Jac: an aspect-based distributed dynamic framework. *Softw. Pract. Exper.*, 34(12):1119–1148, 2004.

[18] D. Suvée, W. Vanderperren, and V. Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29, New York, NY, USA, 2003. ACM.