

A model and a tool for Event-based Aspect-Oriented Programming (EAOP)*

Rémi Douence and Mario Südholt

December 2002 (2nd edition)

Abstract

Aspect-oriented programming promises support for incremental program development by providing new means for modularization of crosscutting code. In this paper, we briefly introduce a general model for aspect-oriented programming, Event-Based AOP (first introduced in [DMS01a]), which is based on monitoring of execution events. This model extends previous approaches by enabling the systematic treatment of relationships between execution points, supporting operators for aspect composition, and allowing the application of aspects to other aspects. Moreover, we present a tool which implements the model for Java.

Technical report no.: 02/11/INFO

*This work is partly funded by the EU project "EASYCOMP" (www.easycomp.org), no. IST-1999-014191

Contents

1	Introduction	3
2	Event-based Aspect-Oriented Programming	3
3	The EAOP tool: architecture and implementation	4
3.1	The preprocessor	4
3.2	The execution monitor	5
3.3	Events	6
3.4	Aspects	6
3.5	Aspect composition	6
3.5.1	Aspects of aspects	8
4	Example: managing discounts in e-commerce applications	8
4.1	Base application: an e-commerce shop	9
4.2	A lottery aspect	9
4.3	A discount aspect	9
4.4	Profiling discounts using a profiling aspect	11
5	Related work	12
6	Conclusion and future work	13

1 Introduction

Aspect-Oriented Programming (AOP) is a research domain [AOP01] which aims at support for incremental program development by providing new means for modularization. In fact, certain concerns (frequently denoted as “non-functional”¹) are intractable by means of traditional modularization mechanisms: programmers thus have to modify code in numerous places in order to integrate these concerns.

AspectJ [KHH⁺01, ASP], developed by Gregor Kiczales’ group at Xerox, was the first tool providing linguistic support for the modularization of crosscutting code. AspectJ has introduced the notion of a “pointcut” enabling the declarative specification of a large number of execution points (by means of generic signatures) where an aspect modifies the base program, and a notion of “advice” defining the modifications themselves. However, AspectJ constitutes only one point within the space of aspect definition languages. One important motivation for our work is the development of a testbed for AOP. We are thus not interested in efficiency issues (although the prototype supports selective instrumentation, see Section 3.1) but concentrate on the expressiveness of the mechanism provided for AO language definition.

We propose an approach to AOP which is based on the observation of execution events, Event-Based AOP (first introduced in [DMS01a]). This approach allows us to systematically treat relationships between pointcuts, operator-based composition of aspects, the definition of aspects which are applied to other aspects and dynamic instantiation of aspects.

This report is structured as follows: Section 2 reviews the main characteristics of our model for AOP. Section 3 shows how our model can be implemented for Java as a tool². Application examples are presented in Section 4. Finally, we discuss related work and conclude.

2 Event-based Aspect-Oriented Programming

Our work on Event-based Aspect-Oriented Programming (EAOP) is guided by a main goal: the study of descriptive means for the definition of expressive aspects. More concretely, our model has the following characteristics.

Aspects are expressed by means of events emitted during execution of the so-called base program. Aspect weaving is realized using an execution monitor, which enables event sequences to be detected. In order to keep our model simple and intuitive (and also enable links to be established between the implementation and the formal work presented in [DMS01a, DFS02]), we consider a sequential model for synchronous events: first, as soon as an event is emitted, its handled by all aspects; second, executions of the base program and aspects proceed in turns: the base program and the monitor are two coroutines.

An aspect is defined by means of two, well-separated, languages: a crosscut language, that allows the definition of execution points where an aspect may modify the base program, and

¹These non-functional concerns are not limited to technical services *à la* CORBA, e.g. distribution, persistence, ..., but include services, such as quota management or prefetching strategies.

²A public distribution of this tool is scheduled for December 2002.

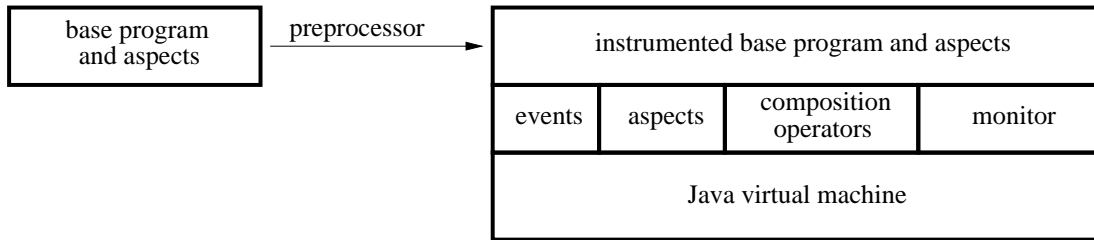


Figure 1: Architecture of the EAOP tool

the action language (called “advice” in AspectJ), which enables the execution of the base program to be modified. For instance, a security aspect can define a crosscut detecting sequences consisting of a “request” followed by a “service allocation” which triggers an action (e.g., an authentication).

In order to facilitate aspect definitions, our model provides operators for explicit aspect composition. These operators enable the elimination of conflicts caused by aspects interacting at the same crosscuts. For example, the security aspect mentioned before and a quota management aspect interact on service allocations; in this case, the security aspect should have priority.

Moreover, our model allows the application of aspects to other aspects. For instance, a logging aspect applied to the security aspect could generate a log for system administrators.

Finally, our model allows the dynamic instantiation and dynamic composition of aspects. For example, when a new critical service is discovered, a new instance of the security aspect may be created.

3 The EAOP tool: architecture and implementation

We have implemented a tool for EAOP, which realizes the model for Java. This tool is composed of five parts (see Figure 1): a preprocessor, a unique execution monitor and three libraries (for the definition of events, aspects, and composition of aspects).

3.1 The preprocessor

The preprocessor instruments the Java source code of the base program (as well as aspects if one wants to define aspects of aspects) in order to generate events and call the entry method `trace` of the execution monitor (cf. the instrumentation example shown in Figure 2). The instrumentation is based on the classic technique of method wrapping. First, a method `foo` to be wrapped is renamed to `foo_original`. The transformation then introduces a method `foo` whose body creates a method call event, calls the monitor, then calls the method `foo_original`, creates a method return event, calls the monitor, and, finally, returns to the callee.

Constructors are wrapped similarly. However, it is not possible to generate an event at the beginning of a constructor because its first instruction must be a (possibly implicit) call to `super` [GJSB00]. For this reason constructors are transformed into default constructors (creating

```

// original code
class Bar {
    int i;

    int foo(int l) { return i+l; }
}

// instrumented code
class Bar {
    int i;

    int foo(int l) {
        Event e = new MethodCall(this, "foo",l);
        Monitor.monitor.trace(e);
        if (! e.skip) e.res = foo_original(e.arg("l"));
        Monitor.monitor.trace(new MethodReturn(e));
        return e.res;
    }
    int foo_original(int l) { return i+l; }
}

```

Figure 2: (Simplified) example for the instrumentation of a base program

the object structure) and initialization methods (which can be instrumented with event generation statements).

The instrumentation is implemented by means of tool for the transformation of Java programs: Recoder [REC]. Recoder simplifies the formulation of transformations and ensures that transformations generate valid Java code (i.e., whose results can be compiled). In order to control the instrumentation, our tool includes a framework for the selective application of transformations to methods, classes, and aspects. This enables us to obtain a reasonable efficient implementation if the number of execution points to be instrumented is not too large. This mechanism aside, we are interested in a testbed enabling the definition of very expressive aspect languages and reserve efficiency considerations for future work.

3.2 The execution monitor

The monitor observes events emitted during execution of the base program. It propagates the event corresponding to the current execution point to all aspects. Our architecture is sequential: first, when the base program generates an event and calls the monitor, the base program suspends its execution. Once every aspect had the possibility to react to the current event, the monitor yields control to the base program which resumes its execution. Second, the monitor propagates the current event to an aspect and waits for the aspect finishing its treatment before propagating this event to another aspect. Hence, we eliminate any possibility of concurrency be-

tween monitors and aspects as well as among aspects. (Otherwise, the semantics of event-based systems becomes very complex.)

3.3 Events

Events are objects which represent execution points of Java programs. The current version of our tool supports four kinds of events: method call and return events as well as constructor call and return events. These four kinds are sufficient for our current experimentations, but new kinds are to be added as needed (for instance, events representing field accesses or entry into an `else`-branch). Our current infrastructure is prepared to accommodate such events: the instrumentation phase, for instance, is able to make explicit the control flow of programs and our event library can be extended easily. Events describe the nature of execution points but also their dynamic contexts. For example, a method call event contains the receiver, method name, argument values, depth of the execution stack and the identity of the code currently being executed (which distinguishes the base program from aspects). It also contains a boolean `skip` which is used to indicate that a wrapper must not call the original method (see the Figure 2). This enables an aspect to “replace” a method by another one; in this case a field of the event object (to be set by the aspect) defines the return value.

3.4 Aspects

An aspect can be seen as an event transformer. In fact, an aspect is a Java program which takes an event as parameter, performs some computation (which may modify the event), and waits for the next event. For instance, a security aspect may encrypt the arguments contained in the method call events it receives. The wrappers of the base program are responsible to extract the potentially modified values of event arguments before calling an original method. The return value of a call may be filtered by an aspect similarly.

An aspect is defined by subclassing the abstract class `Aspect` and defining the method definition. This method does not impose constraints on the structure of the code and uses the method `nextEvent` in order to obtain the following event. This last method is blocking and waits for the monitor “waking up” the aspect with the current event. This is implemented using Java’s threads and a library of coroutines ensuring that, at each moment, *only* the base program *or* the monitor *or* an aspect is active.

3.5 Aspect composition

If events would be propagated sequentially to all aspects, the monitor would be equivalent to an iterator over an array of aspects. Aspect composition provides for greater expressiveness. It is possible to restrict the propagation of events to certain aspects and this decision is dynamic: the monitor manages a binary tree in which aspects are leaves and interior nodes are aspect composition operators (which propagate events). The monitor attempts to propagate events sequentially to every aspect by performing a depth-first traversal of the aspect tree.

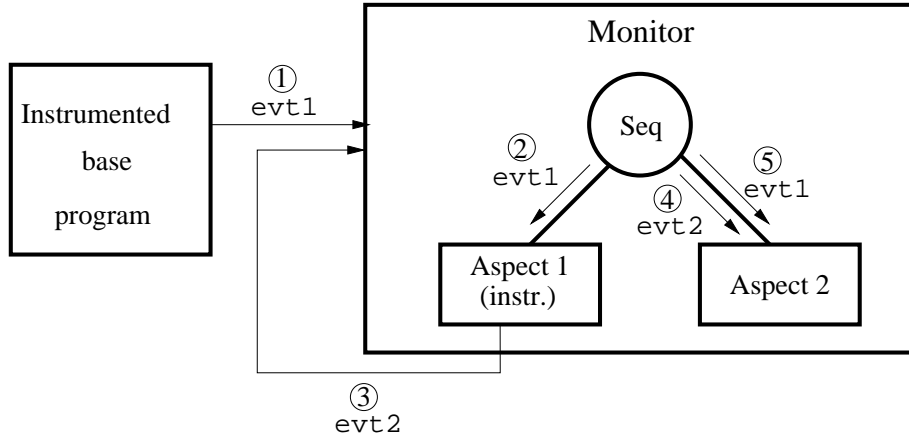


Figure 3: Aspect tree and event propagation

Our tool currently proposes four binary operators for aspects compositions (i.e., four kinds of binary interior nodes):

- `Seq` propagates the current events (coming from the parent) to its left child and then to its right one.
- `Any` propagates the event to its two children in an arbitrary order.
- `Fst` propagates the event to its left child, and then, if and only if the left child *did not* detect a crosscut, the event is forwarded to its right child. Every aspect and every composition operator maintains a boolean field `isCrosscutting` in order to propagate this information. The boolean is managed explicitly by the programmer in all aspects.
- `Cond` propagates the event to its left child, and then, if and only if the left child *did* detect a crosscut, the event is forwarded to its right child.

Consider, for instance, the aspect tree shown in Figure 3, which is composed of two aspects. During execution of the base program, an event `evt1` is generated (step 1). Then, the monitor propagates this event to aspect 1 (step 2) and finally to aspect 2 (step 5). When aspect 2 blocks on a call to `nextEvent`, control switches back to the base program. (For the moment, we ignore steps 3 and 4, which are discussed in the following section.)

New combinators can be developed as needed by subclassing the class representing binary nodes `BinaryAspectS`. Aspect trees can be restructured dynamically (in particular, when an aspect creates a new instance of an aspect and inserts it into the current tree of aspects). The existence of different nodes is also useful in order to implement rules for the dynamic restructuring of the aspect tree (e.g., `Any` is a commutative operator, `Seq` is associative).

3.5.1 Aspects of aspects

The EAOP tool allows the application of aspects to other aspects. On the implementation level, the main problem in this case consists in managing recursive calls to the execution monitor. When an aspect is woken up by the monitor with the current event, it executes its code up to the next (blocking) call to `nextEvent`. The execution of this code may emit events which are transmitted to the monitor. Reconsider the example shown in Figure 3: aspect 1 has been instrumented. Therefore, during its execution, an event `evt2` is generated and passed to the monitor (step 3), which propagates it to the aspects ready to consume it, here only aspect 2 (step 4). When aspect 2 blocks on `nextEvent` control goes back to aspect 1, which resumes its execution as described in the previous section.

During the execution of an aspect, it is not ready to receive an event (i.e., it is not blocked on `nextEvent` but waiting for the return of the method `trace` of the monitor). This state is implemented using a boolean `isRunning`, which, for each aspect, is managed by our infrastructure. This way, the monitor does not propagate events to aspects whose flag `isRunning` is true. This guarantees that weaving does not loop indefinitely (e.g., an aspect weaves itself, or two aspects weave each other). This does not mean that an aspect cannot be applied to itself but this requires the creation of different instances: for example, an instance of a statistics aspect may profile another instance of this statistics aspect, which, in turn, profiles the base program.

Finally, although the monitor sequentializes application of aspects and guarantees that exactly one thread (base program, monitor or aspect) is active at each point of time, the base program may be composed of multiple threads. For this case, the method `trace` of the monitor is declared to be `synchronized`. Finally, note that the monitor is re-entrant in order to cope with aspects of aspects.

In order to conclude this description, note that our tool does not really currently provide a dedicated language for the definition of crosscuts [DMS01a] (crosscuts and actions are both defined as Java code). However, the method `nextEvent` and the aspect composition operators can be seen as basic building blocks for such a language.

4 Example: managing discounts in e-commerce applications

We now show how an e-commerce application may be extended using three aspects. Since we are interested in aspects as a general structuring mechanism, we do not consider an example with non-functional aspects *à la* CORBA but introduction of business logic which would otherwise crosscut the base application. The aspects are defined on the basis of crosscuts detecting certain sequences of purchases/expeditions and illustrate how interaction between aspects can be handled using our tool. Note that this toy application is intended to illustrate the use of our model and tool, but does not pretend to be realistic. The code presented in this section is extracted from an executable application developed with the EAOP tool.

```

public void main() {
    ...
    customer1.getCatalog();
    customer1.searchByPrice(10);
    customer2.getCatalog();
    customer1.addToShoppingCart();
    customer2.searchByName("Les misérables");
    customer1.buy();
    ...
    shop.processOrders();
}

```

Figure 4: Usage scenario for e-commerce application

4.1 Base application: an e-commerce shop

The base application is mainly composed of five classes: Shop, Customer, Product, Shopping-Cart and Order. The method main (see Figure 4) shows a usage scenario: a client inspects the product catalog, puts products in its shopping cart and validates the purchase. The shop then processes its orders (e.g., on a daily basis).

4.2 A lottery aspect

Consider an aspect Bingo managing a lottery intended to improve customer fidelity (each 1000th customer pays only half of its purchase). This aspect requires instrumentation of the method buy of the base application. This can be done by specializing the method checkMethodOr-Constructor, which controls the selective instrumentation of Java code.

The class Bingo shown in Figure 5 extends an abstract aspect Buy which factors out two functionalities for billing: the method computeDiscount and the method nextBuy that returns the next event corresponding to a purchase. This code pattern can obviously be generalized in form of a library of predicates in order to provide a more expressive language for the definition of crosscuts. The definition of Bingo increments a counter for each purchase and modifies the amount of the current purchase if necessary (by reducing the price of the shopping cart of the customer calling buy; the reference to the receiver can be accessed through the event representing the method call).

A unique instance of this aspect is created by inserting the following line of code at the beginning of the base application:

```

Monitor.monitor.aspects = new Bingo();

```

4.3 A discount aspect

A discount aspect can be defined by summing up the different purchases of a customer. Each time this amount reaches a certain total (100 €, say), the amount of the following purchase is

```

abstract class Buy extends Aspect {
    Event nextBuy() {
        boolean ok = false;
        Event e = null;
        while (!ok) {
            e = nextEvent();
            ok = (e instanceof MethodCall)
                && ((MethodCall)e).method.getName().equals("buy");
        }
        return e;
    }
    float computeDiscount(float rate, Object o) { ... }
}

class Bingo extends Buy {
    int orderNo = 0;

    Event nextBuy() {
        boolean ok = false;
        Event e = null;
        while (!ok) {
            e = nextEvent();
            ok = (e instanceof MethodCall)
                && ((MethodCall)e).method.getName().equals("buy");
        }
        return e;
    }
    public void definition() {
        while (true) {
            Event e = nextBuy();
            orderNo++;
            if (orderNo % 1000 == 0) {
                float discount = computeDiscount(0.5, e.receiver);
                e.receiver.addToShoppingCart(new Product("bingo", -discount));
            }
        }
    }
}

```

Figure 5: A lottery aspect

```

class Discount extends Aspect {
    float accPurchases = 0;

    Customer newCustomer() {
        // get the customer from a constructor return event of Customer()
    }
    Order nextShip(Customer c) {
        // get the order from a method call event to ship()
    }
    public void definition() {
        Customer c = newCustomer();
        insert(new Discount());
        while (true) {
            Order o = nextShip(c);
            accPurchases += o.total();
            if (accPurchases > 100) {
                Event e2 = nextBuy(c);
                float discount = computeDiscount(0.1, c);
                c.addToShoppingCart(new Product("discount", -discount));
            }
        }
    }
}

```

Figure 6: A discount aspect

reduced by 10%.

The aspect `Discount` shown in Figure 6 implements this behavior. The class `Discount` defines a method `newCustomer` in order to detect when a new customer is created within the application. The method `nextShip` is used to react when an order is processed by the shop. These methods, which are similar to `nextBuy` in Figure 5, are based on the method `nextEvent`. A unique instance of this aspect is created initially:

```

Monitor.monitor.aspects = new Discount();

```

At the start of the definition of the aspect, a call to the method `insert` dynamically creates a new instance of the aspect and inserts it into the aspect tree at the current position. This new instance then is ready to manage the next new customer. As to the current instance of the aspect, it starts accumulating the purchases of the current customer `c`.

It is also possible to compose this aspect with the aspect `Bingo` in order to ensure that a client winning at the lottery does not benefit *at the same time* of a discount:

```

Monitor.monitor.aspects = new Fst(new Bingo(), new Discount());

```

4.4 Profiling discounts using a profiling aspect

If the shop manager is interested in evaluating the number of discounts granted by `Bingo` and `Discount`, a third aspect can be used to observe the two aspects and recognize calls to the method

```

class Profiling extends Aspect {
    Event nextComputeDiscount() {
        // get event of next call of computeDiscount() defined in class Discount
    }
    public void definition() {
        int n=0;
        while (true) {
            Event e = nextComputeDiscount();
            this.isCrosscutting = true;
            n++;
        } } }

```

Figure 7: A profiling aspect

computeDiscount as shown in Figure 7.

This new aspect is instantiated and composed with the other two using the following composition statement:

```

Monitor.monitor.aspects =
    new Seq(new Fst(new Bingo()),new Discount()),new Profiling());

```

In order to conclude this discussion, note that we have also applied our tool to the “Enterprise JavaBeans” component model [EJB]. We have successfully extended EJB’s predefined security aspect to inject precondition-like functionality. However, the restrictions consisting in sequentialization of execution have to be addressed in order to be able to treat fully distributed EJB applications.

5 Related work

This report is based on our previous work, in which we have exposed limitations of AspectJ-like approaches to AOP [DMS01b], proposed a formal and expressive model for crosscuts [DMS01a] and formally studied aspect interactions by restricting the expressiveness of the crosscut language (essentially to finite state automata) [DFS02].

Other researchers have presented work considering restrictions on the action language of aspects (essentially limiting them to the possibility to abort the base program execution) and optimizing the weaver in the context of aspects for the definition of security policies expressed over execution traces [CF00].

AspectJ [ASP, KHH⁺01] is the best-known tool for AOP and offers a specialized language for crosscuts definitions. AspectJ uses a preprocessor rather than an execution monitor, which is probably more efficient but also less flexible. AspectJ’s notion of aspect composition is very limited (essentially allowing to order several aspects) and aspects of aspects are not supported. Finally, in order to formulate expressive crosscuts relating sets of atomic ones, the programmer has to manually code dependencies between the involved execution points (e.g., using a counter

to express “the second time method `foo` is called after a call to `bar`”, whereas our approach allows to directly express the sequence `nextBar();nextFoo();nextFoo()`.³

HyperJ [TOHS99] is a weaver for Java code. It proposes a language allowing the specification of different kinds of fusion of classes and methods. However, it focuses on the code structure and does not support relations between execution points.

Computational reflection is a general technique used to make explicit and to modify the execution mechanisms of a programming language. This technique enable concerns to be separated from the base application by encapsulating them in one or several metalevels. It is thus not appropriate when relations between execution points have to be expressed which do not fit the structure of the base language (see p.ex. [Led98]). Moreover, programming at the metalevel is generally done using the base language itself. Hence, reflective approaches do not support AOP development using multiple aspect languages.

JAC [PSDF01] is a system for AOP based on a reflective infrastructure. As our tool, it supports the dynamic application of aspects. However, JAC does not provide support neither for dedicated aspect languages nor for the dynamic creation of aspects.

The composition filter model [BA01] inherits from previous reflective approaches and proposes method wrappers in order to filter method calls and returns. This approach includes a simple but very limited composition technique which is based on wrapper nesting. Composition filters do not enable relationships between different execution points to be formulated naturally, and do not provide support for attachment of filters to arbitrary groups of objects.

Finally, a large body of work exists on the notion of “trace analysis.” For instance, some approaches use Prolog to define dynamic analyses over traces for debugging purposes [Duc99]. Other papers apply temporal logic to safety properties in embedded systems [HR01]. All these approaches can be interpreted as AOP approaches featuring very rich crosscut languages (Prolog, temporal logic) but limited action languages (e.g., warn a user of a bug). In this framework, composition and dynamic creation of aspects are not considered.

6 Conclusion and future work

In this report we have presented a model and a tool for Event-based AOP. The model is based on execution traces of programs rather than their code. It provides expressive crosscuts defined as sequences of execution points. This allows to control interactions among aspects by composing aspects and to handle aspects of aspects. Furthermore, programmers have much flexibility to create, remove and reorganize aspects during program execution. We have provided evidence that the presented model for AOP through execution monitoring is expressive as well as intuitive. Finally, it can be instantiated relatively simply for different languages (and allows to integrate aspects written in different languages, e.g. by calling an aspect written in C by a monitor written in Java using appropriately defined events). We have implemented a tool for Java implementing the model and presented initial applications.

³Note that the only exception concerning expressive crosscuts in AspectJ is its `cflow`-construct, which allows to relate execution points to the call-event and return-event of a specific method call.

Our proposal is obviously incomplete and offers several directions for future work. First, new kinds of events, operators for aspect composition and manipulation of the aspect tree should be defined in order to be able to take into account realistic problems. These different constructions should be supported by specialized languages (e.g., a composition language) in order to simplify their use. Second, our model and implementation are essentially sequential. However, numerous applications rely on a concurrent or distributed execution model. Hence, the model should be extended to a suitable concurrent/distributed model for AOP based on monitoring (e.g., one monitor per distributed site). Third, the interactions between the base program and aspects may be complex. It is thus desirable to develop static analysis techniques in order to aid program development. We have defined a first proposal for interaction analysis based on regular crosscuts [DFS02]. It is then possible to statically determine if two aspects may conflict, that is, crosscut at the same execution point. Such analysis should be integrated in our tool. Moreover, our current interpretative implementation could be optimized using program specialization techniques. Finally, experimentations should be conducted in order to evaluate precisely the benefits and limitations of our approach.

Acknowledgements. Thanks to our colleagues Pierre Cointe, Jean-Claude Royer and Eric Tanter for their comments.

References

- [AOP01] *Aspect-Oriented Programming*, volume 44(10) of *Communications of the ACM*. ACM, October 2001.
- [ASP] Aspectj. Home page: <http://aspectj.org>.
- [BA01] L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, October 2001.
- [CF00] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Proc. of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 54–66. ACM Press, January 19–21 2000.
- [DFS02] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Proc. of the ACM SIGPLAN/SIGSOFT Conf. on Generative Programming and Component Engineering (GPCE)*, October 2002.
- [DMS01a] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Proc. of the 3rd Int. Conf. on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *LNCS*. Springer Verlag, September 2001.
- [DMS01b] R. Douence, O. Motelet, and M. Südholt. Sophisticated crosscuts for e-commerce. Int. Workshop on Advanced Separation of Concerns at ECOOP, June 2001.

- [Duc99] M. Ducassé. OPIUM: An extendable trace analyser for prolog. *The Journal of Logic Programming*, 39, 1999. Special issue on Synthesis, Transformation and Analysis of Logic Programs.
- [EJB] Enterprise javabeans. Home page: <http://java.sun.com/products/ejb>.
- [GJSB00] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 2nd edition, 2000.
- [HR01] K. Havelund and G. Roşu. Java PathExplorer — A runtime verification tool. In *Proc. of the 6th Int. Symposium on Artificial Intelligence, Robotics and Automation in Space, ISAIRAS'01*, Montreal, Canada, June 2001.
- [KHH⁺01] G. Kiczales, E. Hilsdale, J. Hugunin, et al. An overview of AspectJ. In J. Lindskov Knudsen, editor, *Proc. of the 15th European Conf. on Object-Oriented Programming (ECOOP)*, volume 2072 of *LNCS*, pages 327–353. Springer-Verlag, Berlin, June 2001.
- [Led98] T. Ledoux. Adaptabilité dynamique des aspects pour la construction d'applications réparties ouvertes. In *Colloque Int. sur les Nouvelles Technologies de la Répartition, NOTERE*, Montréal, Canada, October 1998.
- [PSDF01] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible solution for aspect-oriented programming in Java. In *Proc. of the 3rd Int. Conf. on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *LNCS*. Springer Verlag, September 2001.
- [REC] Recoder. Home page: <http://sourceforge.net/projects/recoder>.
- [TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N Degrees of Separation: Multi-dimensional Separation of Concerns. In *Proc. of the Int. Conf. on Software Engineering (ICSE)*, pages 107–119, Los Angeles CA, USA, 1999.