

Status of work on AOP at the OCM group, APRIL 2001
(Ecole des Mines de Nantes, Technical Report no. 01/4/INFO)

The OCM (Objects, Components, Models) group of the computer science department at Ecole des Mines de Nantes is interested in Aspect-Oriented Programming (AOP). This report is a collection of three short articles presenting work in progress at our lab. We focus on two approaches: AOP from a monitoring perspective (articles 1 and 2) and different approaches to achieve AOP weaving (article 3).

Aspect Oriented Programming from a monitoring perspective Rémi Douence, Olivier Motelet, Mario Südholt.	2
Sophisticated crosscuts for e-commerce Rémi Douence, Olivier Motelet, Mario Südholt.	6
How to weave Noury M. N. Bouraqadi-Saâdani, Thomas Ledoux.	12

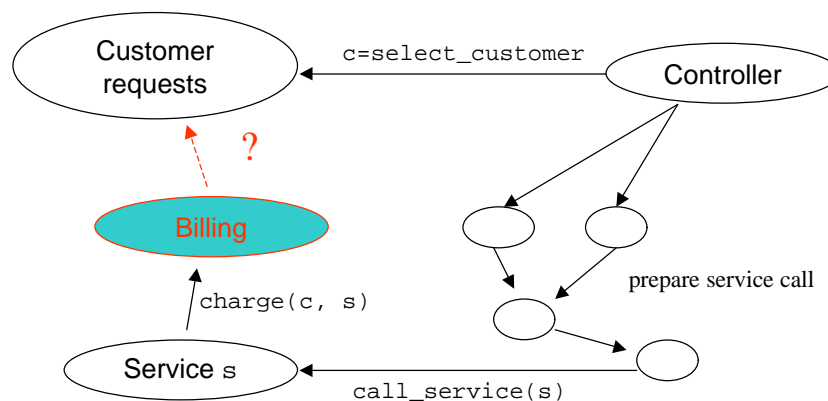
This work has been partially funded by the EU project "EasyComp" (www.easycomp.org), no. IST-1999014191 and by the French government in the context of the RNTL project ARCAD.

Aspect oriented programming from a monitoring perspective

Rémi Douence, Olivier Motelet, Mario Südholt
Département Informatique, Ecole des Mines de Nantes, Nantes, France
Corresponding author: douence@emn.fr

Whereas separation of concern is a most relevant and powerful concept, there are still few dedicated tools to implement this notion. Recently, AspectJ [KIC01] provides a first practical solution by offering both a domain specific language for defining crosscuts and a corresponding code weaver. In this article, we argue that crosscutting is the key notion of AOP which sets it apart from other code structuring techniques. We present execution monitors as a very general and operational model for crosscuts. This model should help to design better crosscut languages and code weavers.

Monitor-based AOP



We believe that crosscutting is the key notion of AOP. Crosscuts relate together different program points or execution points such that common functionality can be inserted there. Imagine a controller (cf. the illustration above) that is intended to make secure a communication between clients and servers which are not security-aware. A typical way to achieve this is to use a multi-step protocol such as the following: first select a customer request, second perform a (potentially) complex computation in order to authenticate the partners and finally call a service. In such an application, the last selected customer is not available when a service is called. However, this information is mandatory in order to upgrade the application such that the service (not the controller!) can bill the customer.

A conventional solution to this problem is to propagate the customer identity by threading this piece of information through the complex computation code (e.g. add an extra argument to all functions implementing this code). An AOP-like solution is to define a billing crosscut that relates a customer request selection with the corresponding called service.

Execution monitors can naturally be used to define such a crosscut: they can detect specific sequences of actions occurring in the monitored program (e.g. a customer selection followed by a

service call), pause the program execution, perform an action (e.g. call a billing function) and afterwards resume the program.

This example demonstrates that monitors can be viewed as an operational model for AOP. In this context, the base program execution should generate events so that monitors could survey the execution. These events should model both the control flow (e.g. method/function calls) and the data flow (e.g. assignments) of the base program. A crosscut could then be defined as a pattern of events: a sequence of execution points specified in a domain-specific language. Basically, pattern detection is done at each event emission. This detection can be modeled as a function of the past events: for each event the monitor must decide either the pattern is still not detected and the application execution is resumed until the next event, or the pattern is detected and an action (e.g. call a function) is performed before the application execution is resumed. So, an aspect could be defined by gathering in a rule a pattern of events with an action to be triggered:

```
pattern => action.
```

Several aspects could be simply defined as concurrent rules.

Crosscuts as event patterns

An event pattern of a rule defines a crosscut in the application execution. Indeed, the monitor has a global view of the application execution, so it can relate non-local events [BAR95, DUC97]. For example, the aspect defining the billing rule

```
c = select_customer(); call_service(s) => charge(c,s)
```

can be interpreted as: monitor the application execution in order to detect the next call to the `select_customer` function, pause the execution, store the value of the variable `c`, resume the execution, monitor the execution in order to detect the next call to the `call_service` function, pause the execution, store the value of the variable `s` and call the function `charge`. So this pattern relates every customer selection to the next service called during program execution.

In general, there are multiple occurrences of one pattern in an application execution. So, one rule can be applied several times. In this case, the programmer should be provided with means to define how to order the different applications of one rule. For example, in case of the following execution trace:

```
c1 = select_customer(); ... c2 = select_customer(); ...  
... call_service(sa); ... call_service(sb);
```

keywords such as `SEQUENTIAL` or `NESTED` could be used to make precise whether the actions to be performed are either `charge(c1,sa); charge(c2,sb)` or `charge(c1,sb); charge(c2, sa)`.

Defining crosscuts as patterns can also help to compose different aspects. In fact, several aspects interact when their patterns overlap. For example, the two following rules trivially overlap:

```
c = select_customer(); call_service(s) => charge(c,s)  
c = select_customer(); call_service(s) => authorize(c,s)
```

Static analysis techniques could detect overlapping patterns in more complex cases and ask programmers to provide a suitable precedence ordering. In the previous example, the call to `authorize` should be performed first because it can cancel the function call `call_service`.

Frequently, the picture is made more complicated because users want to dynamically change the set of active rules during the execution of the application. For example, consider the following rule that checks for memory initialization:

```
add = malloc() ; no(mwrite(add,val)); mread(add)
=> raise exception "non-initialized memory"
```

In order to activate this rule only when super user code is executed, the scope of this rule could be limited by simply filtering out events that are not emitted between a `login(root)` event and a `logout()` event.

Implementation

Standard implementation techniques for monitoring can be applied to the Aspect-Oriented Framework sketched above. In particular, a monitor can be implemented as a process which runs in parallel to the monitored execution. The monitoring process suspends execution of the monitored ones at each event in order to check pattern matching and to perform an action when a pattern has been detected.

This implementation method is frequently subject to severe efficiency concerns. Indeed, the monitored program must pause every time an event is emitted. However, when we deal with simple patterns this general implementation method can easily be optimized. The monitor and the monitored program can be merged together by weaving pieces of the monitor into the program. In the simple case, when an event pattern simply is a single event, the corresponding action can be inserted into the program instead of the code emitting the event. For example, the rule

```
call_service(s) => log(s)
```

which generates a log message every time a service is called, could be implemented by simply inserting the call to the `log` function in the beginning of the `call_service` definition. However, the complexity of the implementation of the monitoring framework depends on the expressive power of event patterns. When a pattern has more than one single event, it may be required to keep track of past events. For example, the rule

```
c = select_customer(); call_service(s) => charge(c,s)
```

requires a global variable storing the last selected customer in order to charge the customer when a service is called. In the same way, the rule

```
add = malloc() ; no(mwrite(add,val)); mread(add)
=> raise exception "non-initialized memory"
```

could be implemented using a hashtable of addresses and boolean variables to register whether a memory location has been initialized or not. When a memory location is read, the action may raise an exception according to the value stored in the hashtable.

In general, the more expressive a pattern is, the more complex its implementation will be. For instance, with multiple nested occurrences of the billing example discussed above (repeated here for reference)

```
c = select_customer(); call_service(s) => charge(c,s)
```

we must introduce a stack of variables instead of a single variable in order to record customer selections. Indeed, such a stack is mandatory in order to charge customer `c1` with the service `sb` and customer `c2` with the service `sa` in the following execution trace of events:

```
c1 = select_customer(); ... c2 = select_customer(); ...
... call_service(sa); ... call_service(sb);
```

Arbitrary complex event patterns can lead to rather expensive implementations. In practice, this problem is likely to be solved by developing (time and space) cost analysis associated to event patterns. Such an analysis could guide the user towards a better use of AOP: the user is responsible for the trade-off between cost and expressiveness.

Conclusion

In this brief overview, we sketched a generic framework for Aspect-Oriented Programming based on execution monitors. Monitoring has the advantage that sophisticated join points can be specified and quite a large subset of these can be reasonably implemented. We are currently working on a well-defined general framework for the specification and implementation of monitor-based AOP.

References

- [KIC01] G. Kiczales et al.: *Aspect-Oriented Programming with AspectJ²*, European Conference on Object-Oriented Programming (ECOOP), 2001, to appear.
- [BAR95] N. S. Barghouti, B. Krishnamurth, *Using Event Contexts and Matching Constraints to Monitor software Processes²*, ACM Computing Surveys, No 1, 1995.
- [DUC97] M. Ducassé, *OPIUM : An extendable Trace Analyser for Prolog²*, The Journal of Logic Programming. Special issue on Synthesis, Transformation and Analysis of Logic Programs, A. Bossi and Y. Deville (eds), 1999.

Sophisticated crosscuts for e-commerce*

Rémi Douence, Olivier Motelet, Mario Südholt

École des Mines de Nantes

4 rue Alfred Kastler, 44307 Nantes cedex 3, France

Corresponding author: motelet@emn.fr

1 Introduction

AOP [2] introduced the notion of crosscutting concerns in programming. An aspect groups a *crosscut* (aka. *pointcut* in ASPECT-J [3]) – which relates several *points of interest* (aka. *join points*) of the base application – with an *action* (aka. *advice*) to be performed. This article argues for a clear separation of *crosscut* and *action* definitions. The tools for AOP currently available support only simple crosscut definitions, which hinders the separation of crosscuts and actions. In this paper we advocate that more sophisticated crosscuts can solve this problem.

This article discusses two aspects in the context of a simple e-commerce application. First, we present this AOP example using simple crosscut definitions only. Then we give an alternative version where sophisticated crosscut definitions enable us to clearly separated crosscuts and actions. Finally, we briefly discuss benefits of elaborate crosscuts definitions in AOP.

2 A simple discount and security aspect for e-commerce

2.1 A basic e-commerce application

In this article, we consider the case of a web-based e-commerce application as a running example. A customer of such an application navigates inside the shop’s web site in search for interesting products: he goes forward from web pages to other web pages while displaying items. This navigation is represented in the following by the command `search`. During his search, he may also go back to some products that were displayed on previous pages by performing the `back` command. The customer can also purchase the displayed products by issuing the command `buy`. This last command does not generate a new web page and does not therefore act on the navigation state. The application allows any combination of these commands. For the sake of simplicity, we suppose the initial page already displays a selection of best-seller products to be bought and `back` has no effect on the initial page. In this setting, a concrete usage scenario could read as follows:

```
search; buy; search; back; search; buy
```

This trace means that the customer made a first search for a specific product and ordered it, then he performed a second search that he canceled, and he finally purchased the result of a third search.

The previous scenario illustrates the base functionality of the e-commerce application. We are interesting in introducing additional behavior like a discount policy for rewarding repeated purchases

*This work has been partially funded by the EU project “EasyComp” (www.easycomp.org), no. IST-1999014191.

and a security policy based on authentication before payments. Such transverse features can be implemented as aspects in the sense of AOP.

2.2 AOP from a monitoring perspective

To make the presentation more precise, we present the examples in a framework defining AOP from a monitoring perspective [1]. In this framework, execution monitors serve as an operational model for AOP. The execution of the base program generates events so that monitors can survey the execution. Such events could model both the control flow (e.g. method/function calls) and the data flow (e.g. through assignments) of the base program. A crosscut can then be defined as a pattern of events, which are tested for each time an event is emitted. An aspect can thus be defined by grouping in a rule a pattern of events with an action to be triggered:

```
aspect = when aPatternOfEvents perform aFunctionCall
```

This definition can be read as: each time the base application performs the execution sequences described in `aPatternOfEvents`, pause the base application execution, call the function `aFunctionCall` with eventually some information about matched events, and resume the base application execution. We presented a prototype for JAVA implementing this framework and a DSL for formal crosscut definitions in [1].

2.3 Informal definition of the discount and security aspects

In the monitor-based framework, the discount policy can be expressed as:

```
discount = when buy a product  
          perform apply discount if not first payment
```

In case of the trace given in Section 2.1, a discount will be applied to the second purchased product. Similarly, the security policy could be defined as:

```
security = when buy a product perform authenticate the user if not already not done
```

In this case only the first payment of the user requires his authentication. Obviously, in general the back action can interfere with the security aspect: once the user is back to a page loaded before the last authentication process, he will have to be authenticated again when the next purchase occurs. For instance, when performing the following sequence of commands `search; buy; back; search; buy`, the user must be authenticated twice.

3 Defining the aspects using simple crosscuts

Figure 1 defines a discount aspect in a JAVA-like syntax. The aspect defines two variables. The variable `firstBuy` which is initialized to `true` is used to identify the first occurrence of the command `buy`. The variable `discountRate` keeps track of the incremental discount to be applied. When a command `buy` is detected, *either* it is the first occurrence of this event and no discount is applied but the boolean tag is changed, *or* the price is reduced using `discountRate`.

Similarly, Figure 2 defines a security aspect. The boolean variable `authenticated` specifies whether the next command `buy` requires an authentication. The integer variable `afterAuthenticate` counts the number of `back` commands allowed such that the user remains authenticated.

In these two aspects definitions, we can distinguish two pieces of code in the actions introduced by the keyword `perform`:

```

aspect Discount {
    boolean firstBuy = true;
    float discountRate = 1.0;

    when buy perform {
        if (firstBuy)
            firstBuy = false;
        else {
            discountRate -= 0.01;
            price *= discountRate;
        }
    }
}

```

Figure 1: A discount aspect (version 1)

```

aspect Security {
    boolean authenticated = false;
    int afterAuthenticate;

    when search perform {
        if (authenticated)
            afterAuthenticate++;
    }
    when back perform {
        if (authenticated) {
            if (afterAuthenticate == 0)
                authenticated = false;
            else
                afterAuthenticate--;
        }
    }
    when buy perform {
        if (!authenticated) {
            authenticate();
            authenticated = true;
            afterAuthenticate = 0;
        }
    }
}

```

Figure 2: A security aspect (version 1)

```

aspect Discount {
    float discountRate = 1.0;

    when enableDiscount() perform {
        discountRate -= 0.01;
        price *= discountRate;
    }

    Crosscut enableDiscount() {
        Event e = nextEvent(buy);
        return enableDiscount2();
    }
    Crosscut enableDiscount2() {
        Event e = nextEvent(buy);
        { return new Crosscut(e);
          |||
          return enableDiscount2();
        }
    }
}

```

Figure 3: A discount aspect (version 2)

Book-keeping code. This code deals with boolean tags and integer counters. They are used to express sophisticated crosscutting conditions such as “unless it is the first payment” or “if it (i.e. authentication) was not done before” (see Section 2.3).

Action code. This code performs changes on the base level, i.e., dealing with variables of the base application (e.g. `price`) and “real” actions (e.g. `authenticate()`).

These aspect definitions are unsatisfactory because they do not provide a clear separation of crosscut and action specifications.

4 Defining the aspects based on sophisticated crosscuts

Figure 3 presents another definition of the discount aspect in which the crosscut is defined using a event patterns which represent *sequences of execution points* instead of a single event as in the previous section. The pattern matching is implemented by the function `enableDiscount()`. This function skips the first occurrence of the `buy` event and calls the function `enableDiscount2()`. This second function returns a crosscut when the next `buy` event occurs. At the same time, it also continue to detect the following occurrences of `buy` (i.e. the next crosscuts). These two concurrent tasks (i.e. returning the detected crosscut *and* continuing to detect further crosscuts) are implemented with the help of the parallel operator `|||` and a recursive call. (These constructions are formally defined in [1].) Note that this new version of the discount aspect does not need a boolean tag in order to distinguish the first occurrence of `buy` from the following ones. Note also that the action introduced by *perform* is not polluted with book-keeping code as is the case in the previous definition.

```

aspect Security {

    when authRequired() perform {
        authenticate();
    }

    Crosscut authRequired() {
        Event e = nextEvent(buy);
        { return new Crosscut(e);
          |||
          return authRequired2(0);
        }
    }
    Crosscut authRequired2(int n) {
        Event e = nextEvent();
        switch (e) {
            case search: return authRequired2(n+1);
            case back  : if (n == 0) return authRequired();
                       else      return authRequired2(n-1);
        }
    }
}

```

Figure 4: A security aspect (version 2)

Figure 4 redefines the security aspect. In this case, the pattern matching functions are `authRequired()` and `authRequired2()`. The function `authRequired2()` has an integer counter as parameter. Note that in this version the action introduced by *perform* is also not polluted with book-keeping code simply calls `authenticate()`.

These examples demonstrate that an expressive language for crosscut definitions is useful to obtain a clear separation between crosscut and action definitions. This separation makes the aspect specifications easier to understand: the programmer can read crosscuts and actions separately. The aspect specifications are also more reusable since the programmer can modify crosscuts and actions separately.

5 Discussion

In this article, we presented two different definitions of the same aspects. One, discussed in Section 3, relies on simple crosscuts definitions by restricting a crosscut to a single point of interest and polluting action specifications with book-keeping code. The other definition, exemplified in Section 4, relies on richer crosscuts definitions and specifies a crosscut as a pattern of events denoting sequences of execution points to be matched. Sophisticated crosscut definition provides a clean separation of crosscuts and actions. We argue that this separation supports the study of aspect interactions.

Study of aspect interactions using sophisticated crosscuts. Aspect interaction is an important issue of AOP. In our e-commerce scenario, both aspects may interact. For instance, in the usage scenario

search; buy; back; search; buy, both discount and security aspects crosscut at the second buy. In this case, the two corresponding actions must be carefully ordered. Indeed, a failed authentication should cancel the discount procedure: the security action must be executed before the discount action. On the other hand, simple and realistic restrictions on the application scenario could prevent their interaction: for instance, if each buy flushes the web page cache in order to forbid too large a number of back commands, both aspects can never interact and their order is no more important.

Aspect interaction issues can often be handled as crosscut interaction issues and can be studied by formally analyzing crosscuts. A clear separation of crosscut and action then allows to focus the analysis on the sole crosscuts definitions.

Perspectives. The tools for AOP currently available do not provide satisfying support for sophisticated crosscut definitions. There are some interesting first steps (like ASPECTJ's `cflow()` primitive which avoids polluting the action code with a stack-like crosscutting behavior) but they are limited to predefined primitives.

In [1], we introduced a crosscut language expressive enough to define sophisticated crosscuts and allowing a clear separation of crosscut and action. Its semantics could serve as a formal base for crosscuts definition analysis as we detailed in proving certain crosscut equivalences — a special case of interaction. This topic remains to be studied further to achieve a general method for the analysis of aspect interaction.

References

- [1] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. Technical Report 01/3/INFO, École des Mines de Nantes, 2001.
- [2] G. Kiczales et al. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *11th European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.
- [3] G. Kiczales et al. An overview of ASPECTJ. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2001. To appear, preprint version: see ASPECTJ home page, www.aspectj.org.

HOW TO WEAVE?

Noury M. N. Bouraqadi-Saâdani & Thomas Ledoux
{Noury.Bouraqadi, Thomas.Ledoux}@emn.fr
Ecole des Mines de Nantes

AOP is an emerging paradigm that allows the separation of multiple concerns in the software development process. Weaving which stands for knitting aspects together into a coherent application is one of critical issues of the AOP technology. In this paper, we focus on approaches to achieve weaving and related issues.

A. A point of view on AOP

In order to present our point of view on AOP, we start from the basics of computing and the concept of program. A given *program* P can always be viewed as a sequence of statements that are aimed to produce some *result* R . By result, we mean any kind of outcome (a calculation, a display...). This result R is obtained through the execution of the program P . This execution is done by some platform (hardware, operating system, virtual machine...) that *interprets* the program's sequence of statements. As an example, a Java program is a sequence of byte-codes interpreted by a virtual machine. A program written in C is compiled into a chain of bits that is interpreted by the CPU. Thus, any result R of a computation depends on both a program P and an interpreter I . A different result may be obtained by changing, at least, one of the elements of the couple $\langle P, I \rangle$.

Now, let's go back to AOP with this context in mind. The definition of AOP given by Kiczales et al. [1] states that a property of an application implemented using AOP can be either represented as:

- a component if it is cleanly encapsulated in a building block of the programming language, or
- an aspect if it can not be cleanly encapsulated in any construct of the programming language. Aspects are properties that cross-cut components and tend to affect component performance and semantics.

We consider the set of application components as a unique entity, which is a program P_0 written for some interpreter I_0 . This couple $\langle P_0, I_0 \rangle$ produces some result R_0 . Aspects affect application execution in such a way to produce another result R_1 . This new result can be obtained by changing, at least, one of the elements of the couple $\langle P_0, I_0 \rangle$. The change is either a transformation of the program P_0 , or a transformation of the interpreter I_0 , or both transformations. We claim that these transformations allow describing the whole set of possible approaches to achieve aspect weaving.

B. Approaches to achieve AOP

1. Weaving through program transformation

This approach consists in transforming only the program P_0 within a couple $\langle P_0, I_0 \rangle$. The interpreter I_0 is kept unchanged. A new program P_1 is built from both the application aspects and the program P_0 . Each aspect is a set of transformation rules that refer to some join-points. Aspect weaving consists in applying these transformation rules to the initial program P_0 . This

approach has been adopted in different works [3, 4] including AspectJ [2]. Note that in these works the interpreter I_0 is implicit.

As an example, consider the following simple program P_0 (written in pseudo-code):

```
class A {
void foo(){
    //do something}
void bar(){
    //do another thing}
}
```

Now, suppose we want to log in some file all invocations of the `foo()` method. This logging is a particular aspect which refers to the join point "invocation of the `foo()` method". This aspect is defined using a transformation rule that inserts a log statement at the beginning of the `foo()` method:

```
aspect Log{
    insert 'logFile.write(currentDate)' in method foo()
}
```

Aspect weaving consists in applying the log aspect transformation rule to our program P_0 . We obtain the code below that plays the role of the new program P_1 :

```
class A {
void foo(){
    logFile.write(currentDate)
    //do something}
void bar(){
    //do another thing}
}
```

2. Weaving through interpreter transformation

This approach consists in transforming only the interpreter I_0 in a couple $\langle P_0, I_0 \rangle$. The program P_0 is kept unchanged. A new interpreter I_1 is built from both the application aspects and the interpreter I_0 . Each aspect is a set of transformation rules that refer to some join-points. Aspect weaving consists in applying these transformation rules to the initial interpreter I_0 . As a result, the interpretation semantics of program P_0 is changed according to the aspects.

As an example, consider the program P_0 of the previous section and the following simple interpreter I_0 :

```
class Interpreter{
Object invoke(Method m, Object[ ] args, Object receiver){
    //invokeMethod m with arguments args on receiver}
Object read(Field f, Object receiver){
    //return value of field f in receiver}
}
```

In order to log all method invocations in our program P_0 , we define a log aspect as a transformation rule:

```

aspect Log{
    insert 'if(m.name == "foo") logFile.write(currentDate)'in method Interpreter::invoke()
}

```

This aspect refers to the same join point "invocation of the foo() method" used in the example of section B.1. But, the reference is expressed in a different way here, since the transformation rule applies to the interpreter.

Aspect weaving consists in applying the log aspect transformation rule to our interpreter I_0 . We obtain the code below that plays the role of the new interpreter I_1 :

```

class Interpreter{
Object invoke(Method m, Object[ ] args, Object receiver){
    if(m.name == "foo") logFile.write(currentDate)
    //invokeMethod m with arguments args on receiver}
Object read(Field f, Object receiver){
    //return value of field f in receiver}
}

```

Interpreters are usually complex pieces of software, difficult to grasp and modify. Thus, an infrastructure that eases the definition of transformation rules is needed. *Reflective systems* play the role of such an infrastructure [5, 6, 7]. Such a system provides developers with an interface to adapt and extend its interpreter, while hiding implementation details.

C. Related Issues

1. Reuse

Building a particular application using AOP, requires to define aspects that are somehow linked to application components. In both examples of section B, log aspect refers explicitly to the join point "invocation of the foo() method". This explicit reference which is independent from the used approach, makes reuse of the log aspect difficult. More generally, strong coupling between aspects and application components restricts the opportunities of aspect reuse.

In order to reuse aspects in different applications, aspects should be generic. Then, transformation rules defined within aspects should not explicitly reference application components. However, weaving requires linking aspects to application components. To satisfy these two contradictory constraints, the explicit reference between aspects and components should be externalized in a new entity we name *configuration*. To illustrate this new entity, we rewrote the log aspect provided in the example of section B.1 as following:

```

aspect Log{
    insert 'logFile.write(currentDate)' in methods belonging to setOfMethods
}

config{
    setOfMethods = { foo() }
}

```

This principle of externalization of the reference between aspects and components was adopted in some works related to AOP [4][5].

2. Run-time adaptability

By adaptability, we mean the ability to replace, add, or suppress one or more aspects of some application. Adapting an application at run-time can be useful in critical applications that should be evolved without being stopped (e.g. network servers). When adaptation can be predicted before running the application, all the possible aspects and adaptation conditions can be defined prior to weaving. In this case, run-time adaptability results into a state change (possible use of the State design pattern [8]).

But, when adaptation is not predictable before running the application (e.g. upgrading some aspect), some infrastructure is needed to "re-weave" aspects at run-time. This infrastructure should not only be able to weave at run-time but it should also be able to replace (partially or totally) the application. According to the used approach, the program or the interpreter should be replaced. This ability of run-time replacement and more generally the ability of a system to act upon itself are characteristics of reflective systems. In other words, some reflective capabilities are required for such run-time adaptability.

3. Weaving non-orthogonal aspects

Two aspects are said to be non-orthogonal if they refer to the same join point. Transformation rules of these conflicting aspects apply to the same parts of the program (respectively the interpreter according to the approach). The precedence of conflicting transformation rules is usually important. A change of the precedence may lead to different results.

In order to obtain a precedence that solves the conflicts automatically on weaving-time, some works [4,7] suggested to extend aspect definitions with extra information. This information is application independent, and describes partially the effect of transformation rules (code insertion/replacement, used variables...). Thus, the precedence that avoids conflicts between aspects can be deduced automatically. However, some cases are non-decidable, then developers still have to define the precedence manually.

D. Conclusion

Weaving can be viewed as a transformation of a couple $\langle P, I \rangle$ where P is a program written for some interpreter I. Thus, aspects can be seen as sets of transformation rules that apply either to the program P, or to the interpreter I, or to both of them. These transformation rules must be application independent in order to make aspects reusable. As a consequence of this reusability, the weaving process is extended with an extra step where transformation rules are linked to the application.

E. Bibliography

- [1] Kiczales G. et al. "Aspect-Oriented Programming". In proceedings of ECOOP'97. LNCS 1241, Springer-Verlag, 1997.
- [2] AspectJ web: www.aspectj.org
- [3] HyperJ web: www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm
- [4] Demeter and Adaptive Programming web: www.ccs.neu.edu/home/lieber/demeter.html
- [5] Proceedings of Reflection'99. Cointe P. Editor. LNCS 1616, Springer-Verlag, 1999.
- [6] Kiczales G. "Beyond the Black Box: Open Implementation". IEEE Software, vol. 13, n° 1, 1996.
- [7] MetaclassTalk web: www.emn.fr/bouraqadi/MetaclassTalk
- [8] Gamma E. et al. "Design Patterns". Addison Wesley Publisher, 1995.