Composing Aspects at Shared Join Points

István Nagy, Lodewijk Bergmans and Mehmet Aksit

TRESE group, Dept. of Computer Science, University of Twente
P.O. Box 217, 7500 AE, Enschede, The Netherlands
+31-53-489{5682, 4271, 2638}
{nagyist, bergmans, aksit}@cs.utwente.nl

Abstract. Aspect-oriented languages provide means to *superimpose aspectual behavior* on a given set of *join points*. It is possible that not just a single, but several units of *aspectual behavior* need to be superimposed on the same join point. *Aspects* that specify the superimposition of these units are said to "share" the same join point. Such shared join points may give rise to issues such as determining the exact execution order and the dependencies among the *aspects*. In this paper, we present a detailed analysis of the problem, and identify a set of requirements upon mechanisms for composing *aspects* at shared join points. To address the identified issues, we propose a general and declarative model for defining constraints upon the possible compositions of *aspects* at a shared join point. Finally, by using an extended notion of join points, we show how concrete aspect-oriented programming languages, particularly AspectJ and Compose*, can adopt the proposed model.

1 Introduction

The so-called *join point model* is an important characteristic of every AOP language [6]. It defines a composition interface ("hooks") where the behavior of a (sub)program can be modified or enhanced, by *superimposing*[1] *aspectual* (crosscutting) *behavior*. Almost all AOP languages allow composing independently specified *aspectual behavior* at the same join point, which we will refer to as a *shared join point* (SJP). The composition of multiple *aspects* at the same join point raises several issues, such as: What is the execution order of the *aspects*? Is there any dependency between them? These issues are not specific to certain AOP languages but they are relevant for almost every AOP language.

This paper presents a novel and generic approach for specifying *aspect* composition at SJPs in aspect-oriented programming languages. The approach adopts declarative specifications of both ordering constraints and controlling constraints among *aspects*. In the following section (2), we will first introduce an example, which will be used for explaining the problems that may occur when composing *aspects* at SJPs. This analysis results in a set of requirements. In section 3, for specifying *aspect* composition at SJPs. we introduce a simple, generic model, which we term as Core Model. In section 4, we show how the concepts of Core Model can be integrated with aspect-oriented programming languages. Finally, section 5 discusses the related work and the contributions of this paper.

2 Problem Analysis

The superimposition of multiple *advices* on a particular join point involves several concerns. To explain the possible problems, we introduce an example application, which will be used throughout the paper.

2.1 Example

The example consists of a simple personnel management system. Class Employee, shown in Fig. 1, forms an important part of the system. In particular, we will focus on the method increaseSalary(), which uses its argument to compute a new salary.

[1] We use this term to designate the weaving of behavior at one or more locations in the program.

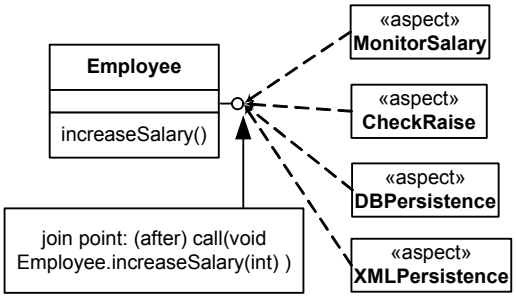

Fig. 1. Class Employee and its superimposed *aspects*

Our example has been defined as a scenario, which introduces a new requirement at each step. Applying the principle of separation of concerns, we implement each of these requirements by separate *aspects* that will be superimposed on the same join point (as well as others): in this example, *after* the execution of the method increaseSalary() of class Employee. We will use AspectJ for illustrative purposes.

2.2 Primary Requirements

In this example we compose, one by one, four *aspects* with class Employee. Each of them will be superimposed on the same join point[2]. At each step, we show the possible problems that can occur at the SJP. We present an analysis of these problems and formulate the requirements towards their solution.

2.2.1 Step 1 – Monitoring Salaries

Assume that the first requirement in this scenario is to introduce a logging system for monitoring changes in salaries. This requirement is implemented by the *aspect* MonitorSalary in Fig. 2:

```
public aspect MonitorSalary{
   …
   pointcut salaryChange(Employee e, int l):target(e) &&
                                              call(void increaseSalary(l));
   after(Employee person, int level):
      salaryChange(person, level){
         System.out.println("Salary increased to level"+level+
                            " for person "+ person);
         … }
}
```

Fig. 2. The *advice* of the *aspect* MonitorSalary

Whenever a salary is increased, this *aspect* will print a notification, including the information about the employee and the type of salary change.

2.2.2 Step 2 – Persistence

Assume that the second requirement in the scenario states that certain objects must store their state in a database. After each state change occurs in the corresponding objects, the database have to be updated as soon as possible. We consider persistence as a separate concern to be implemented as an *aspect*[3].

The *abstract aspect* DBPersistence contains the *advice* that performs the update operation on a persistent object:

---

[2] Note that not every aspect will be superimposed on the same set of join points. However, for all aspects there is a common join point which can be designated by the pointcut "`call(void Employee.increaseSalary(int))`" in AspectJ.

[3] There are several issues, such as connection, storage, updating and retrieval that have to be considered when dealing with persistence. For simplicity, we will focus here only on updating. More details about implementing persistence by aspects can be found in [8].

```
public abstract aspect DBPersistence
   pertarget (target(PersistentObject)){
   abstract pointcut
      stateChange(PersistentObject po);

   after(PersistentObject po): stateChange(po){
      System.out.println("Updating DBMS...");
      po.update();
      … }
   … }
```

Fig. 3. The abstract aspect DBPersistence

The following definition applies the abstract *aspect* DBPersistence to class Employee:

```
public aspect DBEmployeePersistence extends DBPersistence{
   /* Class Employee implements the interface of PersistentObject */
   declare parents:
       Employee extends PersistentObject;

   pointcut stateChange(PersistentObject po):
      call(void Employee.increaseSalary(int))
       && target(po) && … ;
   … }
```

Fig. 4. An implementation of DBPersistence: DBEmployeePersistence

These two *aspects* together implement the necessary behavior for making class Employee persistent. Here, we would like to focus on DBPersistence due to its significance. If the data of a persistent object changes, the corresponding information must be updated in the database too (Fig. 3, the *advice* of the aspect). Changes to the state of the object are captured by the *pointcut designator* stateChange (PersistentObject po), which is implemented in DBEmployeePersistence. Note that the *aspect* MonitorSalary, which was required for the first scenario step, and the DBEmployeePersistence are now superimposed at the same join point.

Even though in most AOP languages *aspects* can be specified independently, once they are superimposed on the same join point, they may affect each others functionality. The concept of shared join point may be experienced when both *aspects* and classes are superimposed. Fig. 5 illustrates these two cases. On the left hand side, we show that superimposing a new *aspect* (CheckRaise) introduces a SJP, together with the previously superimposed *aspect* MonitorSalary. On the right hand side of the figure, it is illustrated that adding a new class can also introduce a new SJP, particularly when there are wildcards in *pointcut designators*.

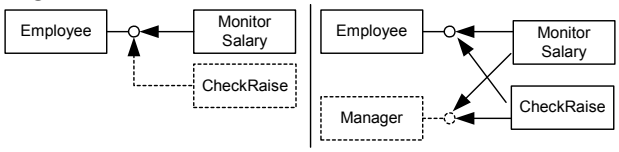

Fig. 5. Examples of creating possible SJPs

Problem: Because the database needs to be updated as soon as possible after the state change occurs in the object, the *advice* of the *aspect* DBPersistence has to be executed before the *advice* of the *aspect* MonitorSalary.

Analysis: As the example illustrates, due to semantic interference, different execution orders among *aspects* at SJPs may exhibit different behavior. We distinguish the following categories of interference:

(A) *No difference in the observable behavior* – For example, consider two *aspects* where each does not refer to the effect of the other but solely maintains its own state. Changing the execution order of the two aspects at a SJP will not be observable after the execution of the *advices* of these two *aspects*.

(B) *Different order exhibits different behavior* – We have distinguished three subcategories of this category:

(B1) The change in the order affects the observable behavior but there is no specific requirement what the behavior should be – As an example of this case, assume that *one* aspect is designed to trace the change in salary and the other one to notify the employee's manager about any change in the salary.

If the requirement is solely "both *aspects* should execute", it does not matter which aspect executes first. If there is an explicit requirement, however, the following category may apply:

(B2) The order of aspects *does matter* because there is an *explicit requirement* that dictates the desired order of aspects – A typical example is the interference between the *aspects* MonitorSalary and DBPersistence. The order between these *aspects* may seem to be not relevant, because they are defined as independent *aspects*. However, for DBPersistence there is a requirement that it should execute as soon as possible after a state change[4] occurs. Since there is no such requirement for MonitorSalary, this implies that DBPersistence must be executed before MonitorSalary.

(B3) There is no explicit requirement for an order, but certain execution orders can violate the desired *semantics* of the aspects – For instance, when multiple *advices* lock shared resources, deadlocks may occur in certain execution order of *advices*. This means that due to the semantics of these *advices*, there are in fact *implicit ordering requirements* to be considered.

Requirement 1: Ordering Aspects – To ensure the required behavior of the superimposed *aspects* at SJPs, it must be possible to specify the execution order of the aspects[5].

2.2.3 Step 3 – Checking Salary Raises

Assume that the next requirement in this scenario is to ensure that an employee's salary cannot be higher than his/her manager's salary. Thus, a raise is not accepted if it violates this criterion. This is enforced by the *aspect* CheckRaise:

```
public aspect CheckRaise
   pertarget(target(Employee) ){
   private boolean _isValid;
   public boolean isValid(){ return _isValid; }

   before(Employee person, int level):
      MonitorSalary.salaryChange(person,level){
        _isValid = true;
   } // workaround for conditional execution

   after(Employee person, int level):
      MonitorSalary.salaryChange(person,level){
        Manager m=person.getManager();
        if ((m!=null) && (m.getSalary() <= person.getSalary()) ){
         //Warning message
         System.out.println("Raise rejected");…
         //Undo
         person.decreaseSalary(level);
         //workaround for conditional execution
         _isValid = false;
        }}}
```

Fig. 6. The *aspect* CheckRaise

The *advice* of this *aspect* (Fig. 6) will check the new salary after the method increaseSalary() is executed[6]. If the rule is violated, a warning message will be printed and the salary will be set back to its original value.

Problem: Adding the *aspect* CheckRaise affects the composition; if this *aspect* fails the DBPersistence *aspect* must not be executed because the employee's data has not changed. That is, the execution of the *aspect* DBPersistence depends on the outcome of the *aspect* CheckRaise.

Analysis: Implementing conditional execution of *aspects* is not trivial since the AOP languages do not provide explicit language mechanisms for this purpose. For example, in AspectJ we can use so called *workarounds*, such as maintaining Boolean member variables in *aspects*, but effective (incremental) composition cannot be achieved in this way; in other words, it is necessary to introduce extra *advices* to maintain the Boolean variables and additional *if-statements* in the existing *aspects* to handle these variables.

---

[4] In fact, in this case the rationale for this feature has to do with the observable different behavior in the case of crashes.

[5] Some AOP languages, for example AspectJ, provide means to specify precedence between aspects, which implies an execution order.

[6] An alternative solution could be the prevention of an invalid raise using a *before advice* (as a pre-condition) instead of an *after advice*. However, this is not feasible in all cases; e.g. it is undesirable to repeat complex salary calculations, as this creates replicated code and may also incur a performance penalty.

Consider for example, Fig. 7 which shows a modified version of DBPersistence. A new if-statement has been added to check if the raise has been accepted by the *aspect* CheckRaise before executing the original behavior of the *advice*.

```
public aspect DBPersistence{
   pertarget (target(PersistentObject)){

   private boolean _isUpdated;
   public boolean isUpdated(){ return _isUpdated; }

   …// workaround for conditional execution

   after(PersistentObject po): stateChange(po){
      if (CheckRaise.aspectOf(
                        (Object)po).isValid()){
         System.out.println("Updating DB...");
         po.update(po.getConnection());
      }
   }
}
```

Fig. 7. The modified version of DBPersistence composed with CheckRaise

Another disadvantage of this solution is that *aspects* will depend on each other. That is, to realize the expected behavior of the composition, *aspects* will need to refer to each other directly. The invocation of the method isValid in Fig. 7 is a typical example of such a dependency. Besides, problems will also occur when CheckRaise, for some reason, is removed from the project.

Requirement 2: Conditional execution – This requirement refers to a case when the execution of an *aspect* depends on the outcome of other *aspects*. Only if the outcome of these *aspects* satisfy a certain criterion, the dependent *aspect* is allowed to execute. To avoid workarounds and their shortcomings, direct language support is needed for expressing this type of dependency.

2.2.4 Step 4 – Updating XML Representations

Assume that the fourth requirement in the scenario states that if the database is not available, persistence must be implemented using XML files. This means, for each instance of Employee, an XML file has to be generated. If the regular persistence does not take place (e.g. because of database connection problems), the file must be updated after each state change of an instance of class Employee. This is realized by the *aspect* XMLPersistence in Fig. 8. This *aspect* has one advice, which calls the method that rewrites the XML file if the salary (or other data) changes.

```
public aspect XMLPersistence {
   after(XMLPersistentObject po):
     stateChange(po){
      if ((CheckRaise.aspectOf(
                     (Object)po).isValid())
          &&(!DBEmployeePersistence.aspectOf(
                     (Object)po).isUpdated())
      po.toXML();
     }
}
```

Fig. 8. The *aspect* XMLPersistence

In this example, XML files must be updated only if the *aspect* DBPersistence has not been able to update the database. This means that XMLPersistence must be executed only if DBPersistence has failed and CheckRaise has succeeded.

We identified several dependencies among *aspects* at SJPs. If there is no explicit language support for expressing the dependencies, they have to be implemented as workarounds in the realization of *aspects*. This has generally a negative impact on adaptability and reusability. There is a need for introducing new operators for expressing composition of aspects at shared join points. These operators must be capable of expressing both ordering among aspects and conditional execution of aspects. The composability of aspects should significantly improve in case the operators are largely orthogonal to each other.

2.3 Software Engineering Requirements

In the previous section, we presented the requirements from the *aspect* intereference viewpoint. In this section, we list software engineering requirements that may play an important role in the quality of programs.

2.3.1 Modularization of dependency specifications

From a software engineering perspective, not only the orthogonality of operators but also the structure and modularization of composition specifications play an important role. In particular, new dependencies are introduced since the specifications need to refer to specific join points, *advices* and *aspects*.

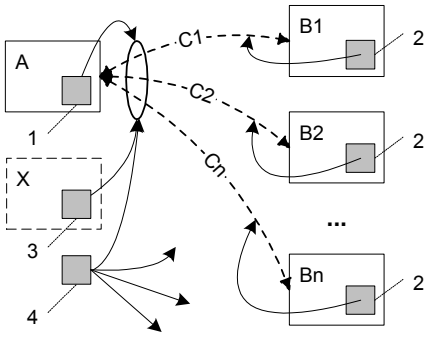

Fig. 9. Four alternative modularizations of constraint specifications; *A*, *X* and *Bi* are *aspect* specifications, *Ci* are composition specifications, and the grey squares (1 to 4) indicate alternative specification loci.

Fig. 9 illustrates a situation where between the *aspect A* and a series of *aspects B1* to *Bn*, the composition specifications *C1* to *Cn* apply, respectively. The figure shows four alternative modularizations of the composition specifications; each of these is shown as a grey square, labeled with a different number. We will discuss each of these numbered alternatives briefly:

1. A combined specification of *C1* to *Cn* is embedded in the specification of the *aspect A*; consequently, this *aspect* will depend on (refer to) *B1* to *Bn*. The introduction of a new *aspect,* say *Bm*, can either be handled automatically by the use of an open-ended specification (as will be discussed in section 2.3.3), or it can require an additional effort to modify the corresponding specification of the *aspect A*.
2. The composition specification is partitioned and the corresponding specifications are located in *B1* to *Bn,* respectively; as a result, each of these aspects will now depend on *A*. A newly introduced *aspect*, say *Bm* must then incorporate the composition specification *Cm*.

A critical issue in the above two cases is that the *aspects A* and *Bi*, now include knowledge about how they depend on each other. In certain cases, this may be exactly what is required, but for example if the two aspects come from different (third-party) libraries, this is not desirable.

3. The composition specification is represented in a separate module (labeled *X* in the figure); *Aspect* specifications in this case do not depend on each other. *X* can be either defined as a dedicated module for describing the composition of aspects, or it is a part of another module (e.g. *aspect* or class). Obviously, *X* will now depend on both *A* and *B1* to *Bn*. Changes to any of these may require an update to *X*. This allows for localizing composition specifications in a set of dedicated modules, if desired.
4. All the composition specifications are collected in one global module (c.f. a configuration file); this is a special case of alternative (3), and has the same dependency issues. In this case, all composition specifications are collected in a single location, which makes it easier to get an overview. However, scaling up to a large system will be more difficult, as the module consequently becomes larger. Obviously, each change to the structure of the system requires a potential revision of this global module.

Based on this analysis, we conclude that it is not desirable to offer a solution which satisfies only a single case; AOP languages should offer a rich set of language mechanisms for composition specifications so that the programmers may choose the right specification for their problem.

2.3.2 Safety and Correctness: Identifying conflicts

An important design consideration is that programmers should be warned if their specification is not sound. A specification is sound when it contains no inconsistencies. This is especially important if the complete specification is made up from several sub-specifications defined at different locations. For example, creating circular relationships is a typical error that can occur in such a case. When a programmer creates a new composition specification, he or she must be warned if the new specification is in conflict with other specifications.

2.3.3 Evolvability: Supporting open-ended specifications

Open-ended specifications in this context ensure that a specification is resilient to changes. Open-endedness may appear in the following forms:

1. The specification directly refers to an abstraction (a language element) that is not (yet) defined. In this case, open-endedness means that the specification is still correct and usable, even though some abstractions that the specification refers to have not been yet defined.
2. The specification indirectly (by defining a number of selection criteria) refers to a set of potential abstractions. In this case, open-endedness means that, if a new abstraction appears in the environment, and it satisfies those criteria, it will be in the set of actual abstractions designated by the specification.

If developers use open-ended specifications for composing *aspects* in SJPs, these specifications will be able to handle (1) *aspects* that are referred to, but not yet present, and (2) *aspects* that are introduced later, but are already designated by the current specification.

3 Core Model

The problem of shared join points is general to AOP languages. For this reason, we propose a generic solution model that can be possibly built into various AOP languages. The presentation aim of this section is not to present a formal foundation, but to illustrate the approach in an intiutive and concrete but language-independent way. This requires a set of assumptions about AOP languages, which are presented in Section 3.1. Section 3.2 presents *composition constraints* as a means to specify composition of *aspects* at SJPs.

3.1 Basic Entities

In this section, we outline the key elements of AOP models, that we consider relevant to our purpose. In order not to be too restrictive, it is important to make only a few assumptions about these entities In particular, we focus on join points and *actions*[7].

Join Points

AOP languages have different means to designate join points. Thus, the range of the possible join points that can be designated varies from language to language. We do not make further assumptions about the designators. We just assume that there are certain points (join points) in the execution of a program where aspectual behavior can be executed.

Actions

In our model, the aspectual behavior that can be executed at join points, is abstracted under the concept of *action.* An *action* has a name that is used for identification, and can have a return value. For the purposes of our model, we are only interested in Boolean return values. These typically indicate a success (*true*) or a failure (*false*) of the *action*[8]. For example, in the case of the example problem where persistence was required, the *action* that is responsible for updating the database will indicate a failure if it cannot connect to the database for some reason. If an *action* returns a value which is not Boolean, or it does not return a value at all, we use the keyword *void* for this purpose. In our model, the return values of actions will be used to express certain dependencies among *actions* at the same joint point.

By default, every *action* assigned to the join point will be executed, unless specified otherwise. The execution of *actions* is sequential[9], that is, only one *action* executes at a given time. In the absence of

[7] These two entities have been identified among the main 'ingredients' of AOP languages [6].

[8] A key reason for this restriction to Boolean values is that it guarantees uniform interfaces between the actions; allowing for more freedom in choosing return types would create undesired coupling between actions, since actions would become dependent on–the compatibility of–the return types of other actions.

[9] Parallel execution is an orthogonal issue; if synchronization between (actions executing in) multiple threads is needed, this is not a different problem from regular issues of thread-safe code. In this paper, we focus on the

ordering constraints, the execution order of the *actions* is undefined[10]. How to handle this is considered a language-design issue. Typically, a fixed order can be determined at compile-time, and be applied for each execution. Alternatively, a random order of *actions* may be generated for each execution; this can result in a non-deterministic execution order.

3.2 Constraints

Our proposed model for composing aspects at SJPs is based on declarative specifications of constraints. Constraints define dependencies between *actions*. We distinguish between two main categories of constraints: *ordering constraints* and *control constraints*. Ordering constraints specify a partial order upon the execution of a set of *actions*. Control constraints specify conditional execution of *actions*.

Ordering Constraints

Ordering constraints specify a partial ordering over *actions*. When several *actions* are superimposed upon the same join point, all these *actions* are assumed to execute once, in an unspecified order. This implies that there can be many possible valid orderings. By applying ordering constraints, the number of possible orders can be decreased. For example, assume that four *aspects* are superimposed on the same join point, for example as shown in section 2.2. Without any ordering constraints, the number of possible execution orders is 4!=24. To be able to specify ordering, we need to introduce an ordering constraint.

Constraint pre

The pre constraint specifies that the execution of corresponding *action* should precede the execution of another *action* at the SJP. The definition of the pre constraint is the following:

pre(x,y) – The order of *actions* is such that x should never be executed after the execution of y has been completed. Besides, y should be executed only after x has been executed at this join point[11]. (The two actions do not have to follow each other directly; other actions can be executed between them.)

We use Table 1 to illustrate the definition of constraints that are applied on two *actions*, respectively *x* and *y*. The topmost row of the table shows the applied constraints. Let us now focus on the column of the constraint pre. The leftmost column lists the possible values (true, false and void) that the *action* x can have after its execution. The last item in this column is the special case when the *action* x has not been executed for some reason. According to the applied constraint and the return value of x, the remaining cells of the second coulumn from left indicates if y is allowed to execute after the execution of x or not. We can see in this figure that the pre constraint is not influenced by the return value: in each case y can be executed after x is executed. The last cell in this column shows that y is not allowed for execution if x has not been executed .

	pre(x,y)	cond(x,y)	skip(x,y,R)
x: true	y:yes	y:yes	y:no, {$y_{return} \leftarrow R$}
x: false	y:yes	y:no	y:yes
x: void	y:yes	y:no	y:yes
x: did not run	y:no	y:no	y:yes

Table 1. The execution semantics of the composition constraints: **y:yes** means that y can be executed according to this specification; $\mathbf{y_{return} \leftarrow R}$ means that return value of y is substituted with R

In Table 2 we illustrate how the pre constraint decreases the number of possible orders. We use the case that we have introduced in section 2. As a short hand notation, we show only the first letter of the name of an *action*. We assume that all four *actions* (C = CheckRaise, D = DBPersistence, M = MonitorSalary, X = XMLPersistence) are superimposed upon the same join point. In the middle column, we list the constraints applied, and correspondingly in the right column we list all the possible orders which are valid. In the first row (Case I.) we apply only one constraint specifying that DBPersistence should be executed before MonitorSalary. The last six possible orders of Case I. are those cases where the execution of DBPersistence and MonitorSalary are interleaved with other actions (C and/or X).

---

sequential execution of aspects. Parallel execution of actions at shared join points is outside the scope of this paper. In particular, we have not encountered any motivation for exploring this further.

[10] In this case, the programmer should be warned about possibly unspecified orderings.

[11] In general, constraints do not allow for the execution of an action if the dependent action did not execute. In other words, we deal with *hard* constraints. To be able to specify open-ended constraint specifications, we introduced additional functions that are discussed in Appendix A of [10].

Case	*Constraints*	*Possible Orders*
I.	**pre(D, M)**	***DM***CX, C***DM***X, CX***DM***, ***DM***XC, X***DM***C, XC***DM*** | ***D***C***M***X, ***D***CX***M***, C***D***X***M***, ***D***X***M***C, ***D***XC***M***, X***D***C***M***
II.	pre(D, M), **pre(D, X)**	C***DMX***, C***DXM***, ***D***C***MX***, ***D***C***XM***, ***DMX***C, ***DM***C***X***
III.	pre(D, M), pre(D, X), **pre(C, D)**	***CDMX, CDXM***

Table 2. The possible execution orders decrease as new constraints are added.

In Case II., we add a new pre constraint, which specifies that DBPersistence should precede XMLPersistence as well. By applying two ordering constraints, the number of valid orders are reduced to six in this case. In the third row (Case III), after applying three constraints, there are only two alternatives left. Here, only the order between MonitorSalary and XMLPersistence is not fixed.

Control Constraints

Control constraints express conditional execution dependencies between *actions*. The general form of a control constraint is the following: *Constraint(Condition, ConstrainedAction)*. The *Condition* is represented by an action, or a Boolean expression built up from actions with logical connectors (AND, OR, NOT). Control constraints use the return value of the executed *actions* for constraining the execution of *ConstrainedAction*.

Constraint Cond

The cond constraint specifies that an *action* is conditionally executed depending on the return value of another *action*. The definition of the cond constraint is the following:

cond(x,y) – Action y can execute only if x returns true. That is, y will not execute in case of the following four conditions: (1) if x returns false; (2) if x returns void; (3) if x has not been executed, or (4) if x is not present at the join point.

For the cond constraint, a Boolean return value is desired. Hence, if strong typing is applied to the return values of *actions* and the arguments of constraints, the void case (which is also used for all non-Boolean return values) can be avoided. We have deliberately included the return value void as a legitimate case to make the system more flexible and applicable to a wide range of languages; either with or without strong typing.

The column of cond(x,y) in **Table 1** illustrates the meaning of the cond constraint: y can execute only if x succeeded (i.e. x returned true). Note that when x did not execute, cond does not allow for the execution of y. Again, a 'soft' constraint would allow for this.

Case	*Constraints*	*Possible Cases*		
		C: true	C:false	C: did not run
IV.	pre(D, M), pre(D, X), **cond(C, D)**	***CDMX, CDXM***	***CMX, CXM***	***MX,*** *XM*

Table 3. Using the *cond* constraint

Table 3 demonstrates the effect of the cond constraint. In Case IV, we have changed the third constraint of Case III to cond(CheckRaise, DBPersistence). Depending on the return value of CheckRaise, there are two sets of possible orders. When CheckRaise returns true (the first column of *Possible Cases*) the possible orders are the same as the one of the pre constraint. However, when the return value of CheckRaise is void or false (the second column of *Possible Cases*) DBPersistence will not be executed. The right-most column, "C did not run", shows that both CheckRaise and DBPersistence have not been executed in this case.

Constraint Skip

The skip constraint specifies that the execution of an *action* may be skipped, based on the result of the logical expression built up from the results of other actions. The definition of the skip constraint is the following:

skip(x,y,R) – The execution of y is skipped and y marked as 'executed' with the return value R, only if x yields true.

R substitutes the original return value of y if y is skipped. For example, R can be true, false or void, but an arbitrary logical expression can also be used to express the return value. In **Table 1**, we show the behavior of skip: y is skipped only if x has been succeeded (i.e. x was true). In addition, the return value of y is substituted with R.

Case	*Constraints*	*Possible Cases*		
V.	pre(D, M), cond(C, D), **skip(D, X,F)**	$C \wedge D$	$C \wedge \overline{D}$	$\overline{C} \wedge D$ = *not _ run*
		CDM{X←F}	***CDMX, CDXM***	***CX CX***

Table 4. Using the *skip* constraint

In Table 4, the first column on the left hand side under the cell *Possible Cases* shows that when both CheckRaise and DBPersistence succeed, XMLPersistence is skipped as if it has returned a false value. In the middle column, CheckRaise succeeds but DBPersistence fails, so XMLPersistence is executed. The third column on the right hand side shows that XMLPersistence will also be executed in the absence of DBPersistence.

Note that other possible cases may occur for both control constraints. We have chosen only those cases that we considered important for the illustration of the behavior of control constraints. With the cond constraint the execution of an *action* is controlled on the basis of information that originates from the *past*. Using the skip constraint it is possible to control the execution of an action that will be executed in the *future*. Although it is perhaps possible to introduce additional constraints, based on the example cases that we have carried out, it seems to be that the three constraints pre, cond and skip are powerful enough for expressing a large category of conditional constraints.

3.3 Composition Rules for Multiple Constraints

The constraints discussed so far, are to a large extent orthogonal to each other. However, when multiple constraints apply to the same *action*, certain rules must be considered to resolve the composition of constraints.

Precedence of Constraint Types

If different type of constraints apply to the same *action*, e.g. skip(x, **z**, true); cond(y, **z**) the constraints are evaluated in a given order according to their type. The precedence order of the three constraints is the following (starting with highest priority): pre, skip, cond. It is important to note that when a new type of constraint is introduced, its relative precedence has to be determined according to this list.

Composition of Constraints

Control constraints are composed with AND logic; an action can be executed only if none of the constraints applied to it forbids its execution. For example, in a set of constraints, if there is a cond constraint that does not allow for execution, the *action* to which it applies cannot be executed. As an example, consider the following pair of constraints: cond(x,**z**); cond(y,**z**). Since both constraints are applied to z, in order to execute z both x and y have to be true. In fact, the above mentioned pair of constraints can be re-written into the following one: cond(x $\wedge$ y, **z**). On the other hand, note that the execution of z can be skipped and marked as executed by an additional skip constraint, since the skip constraint has a higher precedence than the cond constraint.

If complex Boolean expressions are used in cond constraints, they are composed with AND logic as well. Consider the following example, where two cond constraints with different Boolean expressions are applied to the same *action*: cond(a $\vee$ b, **z**); cond(!c, **z**). These two constraints can be rewritten in the following manner: cond((a $\vee$ b) $\wedge$!c, **z**).

Note that OR composition is supported as well, as this is the default way of composition.

Multiple Skips

In case of multiple valid skip constraints with different substitution values, a runtime-conflict occurs, since it is ambiguous which value should be used for substitution. As an example, consider the following pair of constraints: skip(x,z,True); skip(y,z, False). Since both x and y can result in True values after their executions, it is not obvious if the return value of z should be substituted with True or False. Since this problem can be determined only after x and y have been executed, we indicate this by a runtime-conflict when it is necessary. Note that there can be conflict situations that can be determined statically as well. For instance, skip(x,z,True); skip(x,z, False) is a statically detectable conflict, since different substitution values are used with identical conditions.

Cascading Constraints

The sequential composition of *actions* through constraints can have cascading effects. For instance, consider cond(A, B); pre(B,C) as an example. If B is not executed it implies that C will not be executed as well.

3.4 Enforcing Ordering and Control Constraints

The enforcement of a given constraint specification involves two main steps: generating a valid execution order (which may be done statically), and managing execution based on the specified control constraints. Due to lack of space, the reader can find the detailed description of the algorithms that realize these two steps in Appendix C of the technical report [10].

3.5 Hard & Soft Converter Functions

Both the ordering and control constraints introduced previously are termed as a 'hard' constraint. This means that the definition of a constraint does not allow for the execution of the *constrained action* if any *action* that is part of a *condition* is not present at the join point. That is, the constraints aim at ensuring the presence of *actions*. This may be important for the sake of safety and correctness. However, a 'soft' constraint may be preferred sometimes from the perspective of evolvability and maintainability: soft constraints are considered 'tolerant' to the absence of an *action*; if they can handle situations where a specification refers to an action that is not present in the system. This feature can be important to provide open-ended specifications. To support open-ended specifications we introduced hard and soft convert functions that can be used within the scope of constraints. Due to lack of space, for a detailed description of these functions we refer to [10].

3.6 Structural Constraints

Structural constraints form another important category of constraints. They aim at specifying what *actions* have to be or cannot be mutually present at a shared join point. We discuss two kinds of structural constraints: the include constraint defines that the presence of an *action* requires the presence of another *action* at the join point. In contrast, the exclude constraint defines that the presence of an *action* excludes the presence of another *action* at the join point. A more detailed description about these constraints, as well as the possible conflicts that can occur in their specification, can be found in Appendix B of [10].

4 Integration with AOP Languages

In this section, we will show the application of the concepts of Core Model to concrete AOP languages. As we pointed out before, Core Model is intended to be a succinct representation of the core concepts for controlling the interaction among aspects. Hence, it does not address programming language issues such as comprehensibility. It is rather intended as a model that can be adopted by AOP languages. This section is structured as follows: first, we extend the join point concept, as it is available in most AOP languages. Then, we use the extended join point construct to integrate our core model with AspectJ. We revisit the example that we introduced in the problem analysis section and show how the extended version of AspectJ can resolve the identified problems.

4.1 Extending Join Points with Properties

Most AOP languages provide reflective information about the *current* join point by representing the join point as a first-class entity. The 'instance' of the join point can be accessed within the body of the *advice* that is being executed when the join point is reached. For example, in AspectJ, the JoinPoint type represents the concept of join point. The variable thisJoinPoint is an instance of that type and it can be used only within the context of *advices*. The Joinpoint type in AspectWerkz [1], Invocation type in JBoss [7], and ReifiedMessage [3] type in Compose* serve the same purpose.

To implement the conditional execution of *aspects* (i.e. cond constraint) and other concepts of our Core Model presented in section 3, we have extended the interface of type join point with new operations. These operations allow for placing and retrieving extra information into and from an instance of the join point – this extra information may originally not pertain to the join point itself. In this way, the join point will act as a communication channel/bus among the *aspect* instances that are sharing the same join point. Thus, *aspect* instances being executed on the same join point can exchange information among each other through the extended join point interface. In addition, the extra information placed in the join point can also be recognized and maintained by a weaver to direct the weaving process.

Extra Information: Properties

The extra information is represented in the form of properties. A property is a *key-value* pair that belongs to the join point during the execution of *advices*. The *key* is the fully qualified name of the property: a fully qualified representation where the property was created (the namespace and the name of an *aspect* and its *advice*), plus the identifier of the property itself. For example, the *value* is a fully

qualified reference to a constant defined in Java. **Fig. 10** illustrates the structure of properties and an example property.

Definition
Property := <Key; Value>
Key := <Namespace.Aspect.Advice.Identifier>
Value := Fully Qualified Constant References in Java

Example
< Persistence.update.isSucceeded; BooleanConstants.True>

Fig. 10. An example definition of property and its application in an example

Manipulation of Properties
In general, properties can be manipulated by two parties: the weaver and programmers. Before or after the execution of an *advice* the weaver can create, access, change or release a property related to the join point. We refer to the properties recognized by the weaver as *built-in* properties. Programmers can also use built-in properties to direct the weaver. Built-in properties are independent from particular applications; typically, they are used by the weaver for maintaining standard interactions among *aspects*. We consider the conditional execution of aspects as an example of such an interaction. Programmers can also create their own properties and manipulate them within *advices*. We refer to the properties created by programmers as *user-defined* properties. User-defined properties are application specific properties. In this case, a user-defined property realizes a common parameter passing mechanism among *aspects* to exchange information.

4.2 Integration with AspectJ

Before we adopt Core Model in AspectJ, we need to carry out two simple extensions to the language:

Named Advices
As we mentioned above, every property has a fully qualified name for two reasons: to be able to trace back to the origin of the property and to provide a unique name for the property. For this reason, the *advice*-construct of AspectJ needs to be extended with an identifier[12].

Extending the Join Point Interface
To be able to handle properties, the org.aspectj.lang.JoinPoint interface needs to be extended with the following methods:

void createProperty(String propertyId, Object value) throws PropertyExists – creates a property with the given value. If the property already exists, the method throws an exception.

Object getProperty(String propertyName) throws AmbiguousPropertyIdentifier – returns the value of the given property. The propertyName is either the fully qualified name, or only the identifier of the property. If the property with the given identifier or fully qualified name does not exist, the method returns a null value. When only the identifier is used as propertyName and there are more properties with the given identifier, the method looks up and returns the one that is in the default namespace (That is, it looks up the property that is created in the current *aspect* & *advice*). If there is not such a property, the method throws an AmbiguousPropertyIdentifier exception.

void setProperty(String propertyName, Object value) throws AmbiguousPropertyIdentifier – sets the value of the given property. The look up strategy is the same as described at the method getProperty. If the given value is null the property is released.

Fig. 11 illustrates the use of these extensions by a simple example. Within a named *advice* (checkRaise) a property (isSucceeded) is stored with a given value. The initial value (True) is a constant defined in a utility class (BooleanConstants) that contains Boolean constants for our purpose.

```
public aspect EnforceBusinessRules{
   after checkRaise(Employee p, int l): MonitorSalary.salaryChange(p,l){
      …
      thisJoinPoint.createProperty('isSucceeded', BooleanConstants.True);
      …
   }}
```

[12] A number of AOP languages (e.g. AspectWerkz, JBoss, Compose*) already support the identifier of the construct that represents the superimposed behavior. (Typically, this construct is called advice in AOP). However, this does not apply to AspectJ, where advices are unnamed. To keep the backward compatibility of weaver, the name of the advice is an optional syntax element. However, properties can be created only within named advices.

Fig. 11. Placing a property into a join point in AspectJ

Adopting Core Model in AspectJ

Before discussing how AspectJ can adopt our core model, we have to mention that AspectJ has already introduced the declare precedence construct to order the execution of *advices* at shared join points. For this reason, we do not provide a mapping from AspectJ to the ordering constraints in our approach. Consider the following significant characteristics of Core Model:

Granularity of actions: Advices are mapped to the *actions* of Core Model. A built-in property called isSucceeded is introduced to indicate the success or failure of an *advice*. This built-in property can be set by the above described operations, as shown in Fig. 11. To enforce conditional constraints, such as cond, the weaver uses the isSucceeded property of each *advice* that is used in a *condition* of a control constraint. It is not mandatory for programmers to set isSucceeded in each *advice*. If isSucceeded is not set for an *advice* but the *advice* is used in a condition, the weaver takes the void case (neither success nor failure)[13] by default.

Specification of constraints: A new construct is introduced in AspectJ to define specifications of control constraints termed as declare constraints. A set of *constraint statements* is introduced, which aims at providing the desired control constraints. We list the statements along with their mapping to Core Model:

Control constraints (x and y represent advices):

x **if** y; ⇔ *cond(y, x);*

skip x **with** const **if** y; ⇔ *skip(y, x, const);*

Structural constraints (x and y may represent both advices and sets of advices, see details below):

x **includes** y; ⇔ *include(x, y);*

x **excludes** y; ⇔ *exclude(x, y);*

x **m_includes** y; ⇔ *include(x, y); include(y, x);*

x **m_excludes** y; ⇔ *exclude (x, y); exclude (y, x);*

Designation of actions: In general, the arguments of the constraint statements (x and y) designate *advices*, which can be specified according to the template `namespace.Aspect.advice`. For structural constraints, the arguments can designate a set of possible *advices*, which means that the constraint statement is repeated over the elements in the Cartesian product of the argument(s). For example, the arguments of an include constraint statement can be resolved as follows:

{a1, a2} includes {a3, a4} ⇔ *include(a1, a3); include(a1, a4); include(a2, a3); include(a2, a4);*

This is equivalent to four *include* constraints with each of the possible combinations of *advices* a1 to a4. In effect, this illustrates that the constraint statements can express crosscutting constraints.

Modularization of specifications: In AspectJ, the constraint specification, similarly to other declare constructs, is modularized by *aspects*. Note that it is not necessary to place a constraint specification in an aspect that is referred by the specification itself; any aspect can contain arbitrary constraint specifications.

```
public aspect ApplicationConstraints{
   declare constraint:
      DBPersistence.update if EnforceBusinessRules.checkRaise;
}
```

Fig. 12. An example constraint specification in (extended) AspectJ

Fig. 12 shows an example of a constraint specification. It specifies that the *advice* update of the *aspect* DBPersistence executes only if the *advice* checkRaise of the aspect EnforceBusinessRules has been succeeded.

Example Revisited

In Fig. 13 we revisit the second step (section 2.2) of our scenario. In this figure, we show how the extended version of AspectJ can realize the composition of DBPersistence and CheckRaise, without introducing the problems we have identified in its original AspectJ version. In the aspect CheckRaise we have made three modifications: (1) the code that was responsible for resetting the Boolean variable has been removed; (2) the *advice* that is responsible for checking the salary has been named as checkRaise; (3) instead of the Boolean variable that was used for the workaround of conditional execution, the ísSucceeded property has been introduced to indicate the success or failure of checkRaise. The realization of DBPersistence (regarding the update functionality) has been modified in two places: (4)

[13] It is important to note we write the Boolean property into the join point and do not touch the original return value of an advice.

the *advice* that was responsible for updating the database has been named update; (5) the code that was responsible for the conditional execution has been removed. Naturally, it is necessary to express the conditional execution between DBPersistence and CheckRaise. This is done in the constraint specification at (6). As we wrote before, control constraints do not specify the execution order of *advices*; this also has to be provided to achieve the correct composition of aspects.

Note that we have removed all code that was related to the workaround of conditional execution. The remaining code now represents clearly the intended responsibilities of *aspects*, since the conditional execution is realized by the weaver and it is not tangled with the affected aspects. The interaction between the *aspects* is expressed in the form of a declarative specification, which is much closer to the design, as opposed to the tangled realization. Besides, the two *aspects* have become independent from each other, since they do not contain references to each other anymore. As a result, there is a low coupling between these *aspects*; they can be developed and maintained independently.

```
public aspect CheckRaise pertarget(target(Employee) ){
(1) /* removed maintenance code */

(2) after checRaise(Employee person, int l):MonitorSalary.salaryChange(person,l){
      Manager m=person.getManager();
      if ((m!=null) && (m.getSalary() <= person.getSalary()) ){
         //Warning message
         System.out.println("Raise rejected"); …
         //Undo
         person.decreaseSalary(l);
         //setting Boolean for conditional execution
(3)      /* _isValid = false; */
         thisJoinPoint.createProperty("isSucceeded", BooleanConstants.False);
      }
      thisJoinPoint.createProperty("isSucceeded", BooleanConstants.True);
    }}

public aspect DBPersistence pertarget (target(PersistentObject)){
    …
(4) after update(PersistentObject po): stateChange(po){
(5)   /* if (CheckRaise.aspectOf((Object)po).isValid()){ */
         System.out.println("Updating DB...");
         po.update(po.getConnection());
      /* } */
   }}

public aspect EmployeeConstraints{
    declare precedence:
       EnforceBusinessRules, DBPersistence;
(6) declare constraint:
       DBPersistence.update if EnforceBusinessRules.checkRaise;
}
```

Fig. 13. Realization of the second requirement in our scenario using the extended version of AspectJ

4.3 Integration with Compose*

Core Model is generic in the sense that it can be adopted by different AOP languages. For example, we have also provided an integration of Core Model with Compose* in a way that is similar to AspectJ: The join point type of Compose* (the ReifiedMessage class) has been extended to handle properties, and we introduced a pre-defined property (named isSucceeded) to map filtermodules to *actions*. The mapping has been realized using similar steps as we have discussed for AspectJ. The full description of the integration with Compose* can be found in [10].

5 Conclusion

5.1 Related Work

Composition of *aspects* at shared join points is a common problem, which has been –partially– addressed by several AOP languages. In the following, we examine some of them with respect to the requirements that we identified in section 2.

In AspectJ [8][16], the order between *actions* can be controlled by the declare precedence statement. The precedence determines the execution order of *advices* superimposed on the same join point, depending on the type of the *advice*. The precedence declaration can be placed either in the *aspect* that defines the *advice*, or in other independent *aspects*; this allows most of the modularizations discussed in section 2.3.1. Circular relationships among *aspects* are detected only if they are superimposed on the

same concrete join point. The precedence is defined at the level of *aspects*, which implies that different pairs of *advice* of the same two *aspects* cannot have different precedence. As in most other AOP languages, in AspectJ, conditional executions are not supported,. However, to the best of our knowledge, among the current AOP languages, AspectJ is the one that supports the identified software engineering requirements to the largest extent.

Constantinides et. al. [4], emphasizes the importance of the 'activation order' of *aspects* that have been superimposed on the same join point. In their framework, they propose a dedicated class, called *moderator*, to manage the execution of *aspects* at shared join points. The moderator class, as defined in [4], can express conditional execution of *aspects*, but they cannot specify partial ordering relationships between *aspects*. The implementation of the moderator class allows the activation of an *aspect* only if all the preceding *aspects* are pre-activated successfully. In our work, a conditional execution is defined between individual *aspects*. In this way, the execution of an *aspect* does not depend on the order of other *aspects*, *except* the one of which the *aspect* uses as a condition. Note that since the application programmer can implement new moderator classes, it is possible to introduce other activation strategies; however, for certain cases, to define these strategies might not be straightforward in an imperative way as defined in Java. With the composition constraints we propose, the execution strategies are derived in a declarative way. Besides, extending the Aspect Moderator Framework to support partial ordering relationships would allow for a more sophisticated way of the activation of *aspects*.

In JAC [13], wrappers are responsible for the manipulation of intercepted methods. A wrapper is implemented by a class that extends the Wrapper class. The order of the wrappers that can be loaded into the system is handled in a global configuration file. In this file the wrapper classes are listed in their wrapping order. This means in JAC the wrapping order is determined and fixed when the application is loaded, whereas in our approach the order can be adapted, since it is automatically derived when a new *aspect* is superimposed through new constraints.

EAOP [3] defines several operators that are comparable to our constraints. The *Seq* operator specifies an exact order of *aspects*. Unlike pre in our model, it does not allow for partial ordering. The EAOP operators Cond and Fst, are related to the Cond constraint of our model. However, in EAOP the composition *operators* are used *to construct* a composition of *aspects*, whereas in our model we use the *constraints to derive* a possible composition of *aspects*. The difference between the two approaches is that EAOP may require the re-construction of the composition of *aspect* instances whenever a new *aspect* instance has to be included. In our model, by adding one or more new constraints, the composition of the new *aspect* is automatically derived. Further, in EAOP the specification of composition is not open-ended (it requires concrete *aspect* instances) and conflict analysis is not available, yet planned to be integrated in the tool.

5.2 Discussion

SJPs are not a new phenomena, nor specific to any AOP language. To the best of our knowledge, SJP composition has not been explicitly analyzed in-depth in the literature before. In particular, in the current approaches, we have encountered mostly ordering constraints, but little or no control constraints and structural constraints. In this paper, we have first performed an extensive analysis on the issues that arise when multiple *aspects* are superimposed at a SJP. Based on this analysis, we identified a set of requirements that drove our design (section 2). As a generic solution, independent of any specific AOP language, we have proposed a constraint-based, declarative approach to specify the composition of *aspects* (section 3).

The proposed constraint specification can express the composition of *aspects* from different libraries, provided by third parties. This is important for large scale-systems, where a large number of *aspects* are involved in the development process. Unlike other approaches, the composition is expressed in form of declarative specifications, rather than in form of imperative code within methods. This declarative specification allows for defining the composition of *aspects* already in the design phase.

We have implemented and tested the algorithms that are necessary to check the soundness of the constraint specification and detect possible runtime conflicts. By the underlying constraint model and conflict detection techniques we aimed at providing safe use for programmers.

We have extended the join points with the *property* construct to provide a mechanism by which *aspects* can exchange information with each other and control the weaver at shared join points. We claim that this extension is applicable to a wide range of aspect-oriented programming languages that offer an explicit join point type. By using the extended join point type and a dedicated property, we have provided mappings of two specific AOP languages, AspectJ and Compose*, to our Core Model.

Finally, to provide an intuitive use of the constraint model, we proposed a small and clear-cut composition language in AspectJ and Compose*.

6 References

[1] AspectWerkz project; http://aspectwerkz.codehaus.org

[2] L. Bergmans, M. Aksit, "Composing crosscutting concerns using composition filters", *Communications of the ACM*, Volume 44, Issue 10, October 2001.

[3] "Compose* portal", http://composestar.sf.net

[4] Constantinides, C. A., Bader, A., and Elrad, T., "An Aspect-Oriented Design Framework for Concurrent Systems", *Proceedings of the ECOOP'99 Workshop on Aspect-Oriented Programming*, Lisbon, Portugal, 1999.

[5] R. Douence and M. Südholt, *A model and a tool for Event-Based Aspect-Oriented Programming*, Technical Report no. 02/11/INFO, École des Mines de Nantes, 2002.

[6] T. Elrad, M. Aksit, G. Kiczales, K. Lieberherr, H. Ossher, "Discussing Aspects of AOP", *Communications of the ACM*, Volume 44, Issue 10, October 2001.

[7] JBOSS project: http://www.jboss.org

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm & W. Griswold, "An Overview of AspectJ", in *Proceedings of ECOOP 2001*, LNCS 2072, Springer Verlag, 2001

[9] D. Knuth, *The Art of Computer Programming* – Vol.1/Fundamental Algorithms, Addison Wesley, Reading, 1973.

[10] I. Nagy, L. Bergmans, M. Aksit, Declarative Composition of Aspect, *Technical Report*, http://trese.ewi.utwente.nl/publications/publications.php?action=showPublication&pub_id=346 University of Twente, April 2005

[11] C. Noguera, *Compose* - A Run-time for the .NET platform*, EMOOSE M.Sc. thesis, Vrije Universiteit Brussel, September 2003

[12] H. Ossher , P. Tarr, "Multi-Dimensional Separation of Concerns and the Hyperspace Approach", in *Software Architectures and Component Technology: The State of the Art in Research and Practice*, M. Aksit (Ed.), Kluwer Academic Publishers, 2001

[13] R. Pawlak, L. Seinturier, L. Duchien and G. Florin, "JAC: A Flexible Solution for Aspect-Oriented Programming in Java", in *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns,* Kyoto, Japan, 2001.

[14] A. Rashid, R. Chitchyan, "Persistence as an Aspect", in *Proceedings of the 2nd international conference on Aspect-oriented software development*, Boston, Massachusetts, 2003.

[15] P. Tarr, H. Ossher, *Hyper/J User and Installation Manual*, IBM corporation, 2000.

[16] The AspectJ Team, *The AspectJ™ Programming Guide*, 1998-2001 Xerox Corporation, 2002-2003 Palo Alto Research Center.