Associations & Rules: a flexible approach to manage aspects conflicts.

Sandra I. Casas[1], J. Baltasar García Perez-Schofield[2], Claudia A. Marcos[3]

[1]UARG, Universidad Nacional de la Patagonia Austral, Rio Gallegos, Argentina.

[2]Departamento de Informática, Universidad deVigo, Orense, España.

[3]Instituto de Sistemas de Tandil, Universidad Nacional del Centro, Tandil, Argentina.

lis@uarg.unpa.edu.ar, jbgarcia@uvigo.es ,cmarcos@exa.unicen.edu.ar

Abstract. *The separation of concerns at implementation level with Aspect-Oriented Programming (AOP) tools raises the conflicts among aspects. The conflicts detection and resolution are critical operations. First the detection of conflicts is a manual task in most of the AOP tools, second the resolution of conflicts is enclosed to order-schemes. The AOP tools structures and mechanisms do not always support them in a flexible way. In this work the shortcomings of AspectJ for the handling of conflicts are identified and overcome with Model of Associations. This approach is based on two main abstractions: associations and rules. Associations are aspectual relationships. The rules work with associations. They can be explicit or symbolic. An explicit rule solves a single conflict while a symbolic rule solves a set of conflicts. The rules can apply a variety of resolution categories. The detection of conflict is an automatic process and it is a fundamental part of the resolutions of conflicts.*

1. Introduction

The Aspect-Oriented Programming [Kiczales G. et al. 1997] raises the level of Separation of Concerns [Dijkstra E., 1976] [Hursch W. and Lopes C., 1995] within the systems. At the same time it also raises the level of conflicts among aspects. A conflict occurs when two or more aspects compete for its activation.

The conventional AOP models are inadequate for the handling of conflicts in several situations. Also, most of the AOP tools do not support automatic conflicts detecting and the resolutions mechanisms are restrictive. This is the case of AspectJ [Kiczales G. et al. 2001] tool. The AspectJ aspects are structured to carry out too many responsibilities, which obstruct the handling of conflicts.

However, the handling of conflicts with AspectJ [Kiczales G. et al. 2001] is hard for two reasons: the conflicts detection is a manual process and the conflicts resolution

is restrictive. In this work, the shortcomings of AspectJ are discussed and the Model of Associations is presented. This approach provides a method with a more suitable handling of conflicts. The Model of Associations is based on two mechanisms: the definition of associations and the rules. These strategies have been implemented in the programming environment MEDIATOR, making this approach highly flexible, effective and powerful.

This work is structured as follows: Section 2 is dedicated to the analysis of handling of conflicts with AspectJ; in Sections 3 the Model of Associations is presented; in Section 4 the AspectJ problems are solved with this approach; Section 5 exposes the related works; and section 6 presents the final conclusions.

2. Shortcomings of AspectJ

In AspectJ [Kiczales G. et al. 2001], the aspects are units of code that encapsulate the crosscutting concern logic. After the aspects are coded, a weaving process integrates the aspects with the components of basic functionality, generating the final application [13]. An aspect is composed of different constructions such as methods, attributes, introductions, declarations, pointcuts, advice, etc. The pointcuts declare a relationship among aspects and components of basic functionality. This structure has been imposed by AspectJ, which is the most diffused, popular and used AOP tool. The AspectJ structure has also been replicated by many other AOP tools.

The programming model of AspectJ confers too many responsibilities to aspects. The assumption is that if an aspect is responsible for: (i) encapsulating the logic and behavior of the crosscutting concerns (advice); (ii) establishing the link with components of basic functionality (pointcuts); and (iii) establishing the execution order (declares precedence), probably some of these responsibilities will not be fulfilled very well. In consequence, more flexible possibilities are restricted or limited. In AspectJ, aspects are programming constructs that crosscut the modularity of the basic functional classes of the application in predetermined ways.

The mechanism for conflict resolution, provided by AspectJ consists in a very restricted precedence-based scheme (also known as order or priority). In order to execute aspects code in a certain order, it is necessary to specify it with a statement of precedence declaration: "*declare precedence: A, B;*". The semantics means the advices of aspect A will be executed before the advices of aspect B will be. The declare precedence statement presents limitations in the following sceneries:

a)After the aspects are implemented, the developers identify that the aspects outline more than one conflict (Figure 1).

```
aspect A   {                              aspect B   {
 pointcut A1(): call(void CX.met());       pointcut B1(): call(void CX.met());
 Pointcut     A2():     execution(void     pointcut     B2():     execution(void
CY.met());                                CY.met());
 before(): A1()     { ........  }           before(): B1()     { ........  }
 after(): A2()     {  ........  }           after(): B2()    { ........  }
 }                                         }
```

Figure 1. Two different conflicts among aspects A and B.

Each conflict requires different order policies. In this case, it is necessary that the advice associated to the pointcut A1 will be executed before that the advice associated to the pointcut B1. In the other hand, it is necessary that the advice

associated to the pointcut B2 will be executed before that the advice associated to the pointcut A2. This situation is not possible to solve through the mechanism of precedence declarations, due to it is related to the aspects and not to the advice or pointcuts. In this case, the AspectJ solution consists in dividing one of the aspects (A or B) in two aspects, to apply 2 different declare precedence statements.

b) The quantification of join-points cause more than one conflict and each conflict requires different execution orders. For example, in Figure 2, the pointcuts A1 y B1 of A and B aspects respectively provoke more than one conflict (supposing CX class have several set methods).

```
aspect A {                                  aspect B {
  pointcut A1(): call(* CX.*(..));            pointcut B1(): call(* CX.set*(..));
  before(): A1()                              before(): B1()
  {     ........    }                         {     ........     }
}                                           }
```

Figure 2. The aspects A and B outline different conflicts.

In this case, if it is required to apply different order of execution for each conflict, the precedence declaration is insufficient. As the previous scenery, it cannot be defined two or more different precedence declarations that involve the same aspects.

c) After that the aspects implementations are carried out, the developers identify that the conflict is outlined by different pointcuts of the same aspect (Figure 3). The precedence declaration does not have effects for advice of the same aspect, the execution order of advice in conflict is indefinite.

```
aspect A {
 pointcut A1(): call (void CX.met());
 pointcut A2(): call (* CX.*(..));
 before(): A1()     {  ........  }
 before ():  A2 ()    {  ........  }
 }
```

Figure 3. A conflict in the same aspect.

In this case, the AspectJ solution consists in dividing the aspect A in two aspects to apply the declaration of precedence statement.

d) The order of aspect in conflict depends on a condition of the system or of the context. In Figure 4 it is indicated that the advice of aspect A will be executed before than the advice of aspect B if cond is true. Otherwise, it is executed after the advice of the aspect B.

```
if (cond) declare precedence: A, B;
else declare precedence: B, A;
```

Figure 4. Declare precedence with conditions.

This situation is impossible to implement in AspectJ. The declarations of precedence cannot be defined to conditions. Different declarations of precedence involving the same aspects also cause a compilation error.

e) Two or more aspects in conflict are ordered in a particular way in a deployabled application. If these same aspects are used in another application in which the aspects should be executed in a different order, the previous declare precedence

statement is not valid. “In this situation the aspect that contains the declaration of precedence statement should be modified”.

f) In a deployabled application the whole source code of the aspects must be examined to determine the execution order of the aspects in conflicts. This is due to a declaration of precedence, which involves several aspects, can be part of any of these aspects source code. For example, if the aspects A, B, C are in conflict, the precedence declaration can be defined in the aspect A or B or C or others.

The problems indicated in these sceneries force to re-design and to re-implement the aspects in a different way, and impact negatively in the software reuse and maintenance. Because of these restrictions, it is proposed an alternative approach, denominated Model of Associations.

3 Model of Associations

To solve the drawbacks discussed in the previous section, a different model is adopted for crosscutting concerns implementation. This approach assigns the responsibilities to different entities. The entities are: aspects, associations and rules.

3.1 Aspects

An aspect is an independent unit composed by a group of methods and attributes and encapsulates specific crosscutting concern logic. For example, the *Logging* aspect (Figure 5) is formed by the *loogedOperations(..)* method. Aspects are units of code, that they do not declare any crosscut information such as the join-points, pointcuts or advice.

```
aspect Logging  {
 public static void loogedOperations(..)
 {// send operation information to file or console }
}
```

Figure 5. Aspect Logging

3.2 Associations

Associations are entities defined in a separated way, instead of being tied to aspects, they link aspects with classes. That is to say, an association describes a relationship between an aspect and a class. For example, in Figure 6 the *LogASB* association is defined, relating the *Logging* aspect with the *Account* class. It establishes that every time that the *debit(float amount)* method of *Account* class is invoke and the parameter of the debit method is greater than 100 (the condition is optional), the *loogedOperations()* method of Logging aspect is executed immediately. In the last line it has been defined a numeric priority for the association (not for the aspect). This priority will be used later on for the handling of the conflicts. *Looging.loogedOperations()* can be related to other functional components with another priority. The join-point *Account.debit(float)* can be related to other aspects, other associations will be defined when needed. An association is always a one-to-one relationship. The wildcards are allowed to denote a set of join-points. An additional mechanism transforms this n-to-one relationship in n associations.

```
association LogASB  {
 call void Logging.loogedOperations();
 after void Account.debit(float $1);
 condition = ( $1 > 100) ;
 priority = 10;
}
```

```
association LogAcc  {
 call void Logging.loogedOperations();
 after void Account.*(..);
  priority = 10;
}
```

Figure 6. Associations LogASB and LogAcc

This approach allows the aspects to be independent of the systems in which they are used and they can be more reusable. Besides, the approach will facilitate the handling of associations in an isolated particular way. From this mechanism, a conflict only happens when two or more associations define the same relationship type for the same functional component. The handling of conflicts must be applied to associations.

3.3 Rules

A conflict happens if two or more associations define the same class member, the same relationship cut and relationship advice. For example, in Figure 7 the *LogAcc* and *StatAcc* associations are in conflict. Both associations have defined the same crosscutting relationship and advice (call and after) on the same class member (*Account.debit(..)*). In the Model of Associations, the conflicts among aspects outline about the associations.

```
association LogAcc {
call void Logging.consoleOperation(..);
after public void Account.debit(..);
priority = 10; }
```

```
association StatAcc {
call void Statistic.register(..);
after public void Account.debit(..);
priority = 15; }
```

Figure 7. Associations in conflict.

A resolution rule of conflicts is an entity that defines a resolution action for n ($n \geq 1$) conflicts (K) of a program, where each conflict (k) is composed by m associations ($m \geq 2$). A resolution rule is expressed clearly in two parts (Figure 8). The antecedent identifies the group of conflicts that the rule will solve. And the consequent part specifies a precise action of resolution of the associations in conflict. The rules can be explicit or symbolic.

```
R :   K(k1, k2, ..., kn)            // antecedent
   =>
      AR (k1 (a11,..,a1m), .., kn (an1,..,anm))  // consequent
```

Figure 8. Resolution Rule.

The explicit rules require that the antecedent is specified in concrete way. That is to say, it should be indicated the associations in conflict. In the consequent part, these associations are specified as part of the resolution action (Figure.9).

```
ER :   k (a1, a2, a3)
     =>
       AR (a1, a2, a3)
```

Figure 9. Explicit Rule.

Therefore, an explicit rule solves only one conflict. Using this strategy the developer must specify for each existent conflict an explicit rule. In consequence the process of detection of conflicts must be carried out previously.

The symbolic rules allow to apply a resolution action to subsets of conflicts. The symbolic rules can be absolutely general or partially general. As much as general is the antecedent of the rule, more conflicts will embrace the action of defined resolution in the consequent one. The more specific is the antecedent, the less amount of conflicts will be affected by the consequent of the rule. The specification of a symbolic rule does not require to know or to define the conflict. The conflict will be known at the moment that the rule is applied (Figure 10).

```
SR :  <conflict X exists>
    =>
       AR (apply resolution Y)
```

Figure 10. Symbolic Rule.

Where the expression <*conflict X exists*> is an expression that involves a set of conditions (general or specific). Here *X* can be all the conflicts, or only some of them. For example, the conflicts on the class Account. In this case, the rule evaluates the antecedent, such as a boolean condition. The expression <*apply resolution Y*> is a function that applies a resolution action of the conflicts that satisfies the condition. The consequent will assume concrete values of associations after the detection of conflicts. If none of the conflicts satisfies the condition (antecedent), then the action (consequent) is not applied to any association. Since the specification of symbolic rules implies that it is not required to know the existence of the conflict at the moment of its definition. It is necessary some approach that allows to apply the resolution actions to the associations in conflict. This approach is the priority of the associations. All the associations have a priority defined by the developer that will be used for the resolution of conflicts by means of symbolic rules. In Table 1 some examples of explicit and symbolic rules are shown.

Table 1. Examples of explict and symbolic rules.

Example	Rule - Category
Rule: R1 Conflict: {a, b, c} Action: order (b, c, a)	Rule: Explicit Category: Simple (Order)
Rule: R2 Conflict: {a, b, c} Action: if (cond) order (b, a); excluded (c); else annulled (a, b, c);	Rule: Explicit Category: Combined (Optional - Order - Exclusion - Nullity)
Rule: R3 Conflicts: {conflict over Account class} Action: OBP (b, a)	Rule: Symbolic. Sub-category: OBP (order by priority the association in conflicts over Account class)
Rule: R4 Conflicts: {all} Action: OBP (b, a) OBP (b, c) OBP (a, c)	Rule: Symbolic. Sub-category OBP (order by priority the association of all conflicts)

The resolution action can be simple category: order, optional, order inverse, exclusion or nullity. A combined category provides a resolution action that applies two or more simple category to solve a conflict. The sub-category resolution are: OBP (order by priority), IOBP (inverse order by priority), ELP (excluded less priority), EGP (excluded

great priority), OF (order first), OL (order last), ETA (excluded this association), EAA (excluded all associations), ANNULLED. The simple and combined category can be used in explicit rules. The sub-category can be used in symbolic rules.

3.4 Detection of Conflict

The detection of conflicts is an automatic process in MEDIATOR. This process is independent of the weaving, the compilation and the execution. The defined associations are analyzed to generate a list of conflicts. If two or more associations have the same crosscutting relationship, the same advice relationship and the same functional component (class and method) a conflict is created in the list. A conflict entity has all information about the conflict.

On one hand, the developer can define the explicit rules using the list of conflict. On the other hand, an automatic process verifies the list of conflicts against the symbolic rules in order to identify the conflicts that satisfy the condition. When it is satisfied, the system generates for each symbolic rule a set of validate resolution sentences.

3.5. The Weaving Strategy

The weaving process integrates the aspects with the classes to build the final application. The objective is to maintain the associations and the logic of resolution of conflicts as separated and isolated as possible from classes and aspects. The proposed strategy requires classes, associations and resolution rules to weave. The aspects are not necessary because all the necessary information for weaving process is defined in the associations and rules. The weaving process proceeds in two stages. First, a linking class is generated automatically in the compilation phase. The methods of this class are denominated in turn, linking methods. These methods relate functional components to aspects, obtaining the information from their corresponding associations. In Figure 11 the *link_Met1()* linking method is shown, generated from the *LogAcc* association. The *link_Met1()* method invokes the execution of the *Logging.loogedOperations()* aspect method, defined in the *LogAcc* association.

```
association  LogAcc
{
  call void Logging.loogedOperations();
  after void Account.debit($2);
  condition = ($2 >100)
  priority = 10;
}
```

```
class Link_Class {
 public  void static link_Met1($2)  {
    if ($2 >100)
        Logging.loogedOperations();
  }
  -----
} // end linking class
```

Figure 11. Linking method generated from LogAcc association.

The previous example is valid for associations free of conflicts. When the associations to weave are in conflict, the process is carried out in a similar way. In the first phase the objects that represent resolution rule are also required. The linking method concentrates the logic of resolution of conflict. For example, if *LogAcc* and *StatAcc* associations are in conflict and the action of explicit rule is *(order (LogAcc, StatAcc)*), then in the compilation phase, both associations are merged in a unique linking method. The method encapsulates the logic of resolution of the conflict. In

Figure 12, the *link_Met2()* linking method has been automatically generated starting from the action of explicit rule and *LogAcc* and *StatAcc* associations.

```
void static link_Met2( ) {
    Logging.loogedOperations();
    Statistic.register();
}
```

Figure 12. Linking method of associations in conflict.

The second phase proceeds during the execution of the application. In load-time, those classes affected by the associations are linked to the linking methods according to the type of relationship. This process has been implemented by means of the Javassist API [Chiva S., 1998].

Before the weaving process execution, two automatic processes are executed. First, the system checks the existence of rules interferences. The interference between rules happens when an explicit rule and a symbolic rule are applied to the same conflict, and each rule orders different resolution actions to solve the conflict. Then the system solves the interference using previous configurations of the rules mechanism. This configuration is defined by the developer initially. Second, the resolution sentences which are generated by the execution of the symbolic rules, are transformed in explicit rules. For example, if a symbolic rule solves 3 conflicts, then 3 explicit rules are created. The weaving process works with explicit rules.

In summary, in the functional components affected by some association are inserted a call to a linking method, according to the type of relationship (before - after) of the associations (on load – the bytecode modification is not permanent). The linking method invokes the aspect method directly, or it can include a group of sentences that apply a category of conflict resolution. Therefore, the following advantages are obtained: (i) the classes do not have any knowledge about what aspects crosscut them; (ii) the aspects preserve their original state and they can be associated to any other functional components; (iii) the conflicts resolution is hidden in the linking class, being specific for a certain application, and finally, (iv) if a new association or rule is defined it is only necessary to generate the linking class, it is not necessary to compile the other units (classes and aspects).

4. Solving Problematic Situations.

The solutions with Model of Associations to AspectJ restrictions (expressed in Section 2) are presented bellow.

a) Two aspects outline more one conflict. Each conflict requires different resolution policies. This situation is easy to solve in Model of Associations, the developer must specify explicit or symbolic rules that apply different resolution actions, over the associations in conflict. In Figure 13, the *Logging* and *Statistic* aspects outline two conflicts, *(LogAcc, StatAcc)* and *(LogBor, StatBor)*. The explicit rule *R1* applies a resolution on conflict *(LogAcc, StatAcc)* and the explicit rule *R2* applies different resolution to another conflict *(LogBor, StatBor)*. In this case, it has not been necessary to re-design or re-implement the aspects. Simply different rules are declared to solve the conflicts.

```
association LogAcc   {
  call Logging.console();
  after void Account.debit(float );
}
```

```
association StatAcc   {
 call Statistic.register();
 after void Account.debit(float);
}
```

```
association LogBor   {
  call Logging.console();
  after void Borrow.cancel( );
}
```

```
association StatBor   {
   call Statistic.register();
   after void Borrow.cancel();
}
```

```
Rule: R1
Condition: {LogAcc, StatAcc}
Action: order (LogAcc, StatAcc);
```

```
Rule: R2
Condition: {LogBor, StatBor}
Action: order (StatBor, LogBor);
```

Figure 13. Solving two different conflicts among aspects A and B.

b) Two associations provoke more than one conflict because of the use of quantification of join-points and each conflict can be solved in isolation by a rule. In the Model of Associations it is possible; because of the n-to-one associations are automatically converted in n associations. In Figure 14, the *Log* and *Stat* associations use quantification, after they are converted in associations *Log1, Log2, Stat1* and *Stat2*. Two conflicts arise from this situation: *(Log1, Stat1)* and *(Log2, Stat2)*. The developer can define two different rules to solve these conflicts.

```
association Log   {
  call Logging.console();
  after * Account.*(..);
}
```

```
association Stat   {
  call Statistic.register();
  after * Account.*(..);
}
```

```
association Log1   {
  call Logging.console();
  after int Account.getId();
}
```

```
association Stat1   {
  call Statistic.register();
  after int Account.getId();
}
```

```
association Log2   {
  call Logging.console();
  after String Account.getName();
}
```

```
association Stat2   {
  call Statistic.register();
  after String Account.getName();
}
```

Figure 14. Conflicts and quantification.

c) The same aspect outlines a conflict and it is required the application of a specific resolution. This situation is easy to solve in Model of Associations, the developer must specify an explicit or symbolic rule for the associations in conflicts. For example in Figure 15, the *Log1* and *Log2* associations are in conflict and they make reference to the same aspect (Logging). The rule *R5* can be defined to solve this conflict.

```
association Log1   {
  call void Logging.console();
  after int Account.getId();
}
```

```
association Log2   {
  call void Logging.File();
  after int Account.getId();
}
```

```
Rule: R5
Condition: {Log1, Log2}
Action: order (Log2, Log1);
```

Figure 15. A conflict in the same aspect.

d) The order of aspects in conflict depends on a condition of the system or of the context. This situation is easy to solve in Model of Associations, the developer has to specify an explicit rule where the action of resolution applies an optional category. For example in Figure 16, the statistic operation only is performed if the parameter of debit method is greater than value 100.

```
association LogAcc   {
  call void Logging.console();
  after void Account.debit(float);
}
```

```
association StatAcc   {
  call void Statistic.register($2);
  after void Account.debit(float);
}
```

```
Rule: R6
Condition: {LogAcc, StatAcc}
Action: if ($2 > 100)
           order (StatAcc, LogAcc);
        else excluded (StatAcc);
```

Figure. 16. The resolution of conflict depends of a condition.

e) The same conflicts require different policies of resolution in each application. This situation is easy to solve in Model of Associations. The developer must specify different rules for each application and it is not necessary to modify the aspects. In Figure 17 the same conflict is solved in different ways for each applications. The developer only has to re-define the rule *R5*. The developer does not have to re-define the aspects.

Application 1	Application 2	Application 3
Rule: R5	Rule: R5	Rule: R5
Condition: {LogAcc, StatAcc}	Condition:{LogAcc, StatAcc}	Condition: {LogAcc, StatAcc}
Action: Order(LogAcc,StatAcc)	Action: order(StatAcc,LogAcc)	Action: excluded(StatAcc)

Figure 17. Different policies of resolution for the same conflict.

f) If the developer needs to know the execution order of the aspects in conflict (in a deployabled application) he/she only must edit the rules or the linking class.

5. Related Work

Several works have been developed in order to detect and solve conflicting situations among aspects.

The first directly related work with the detection and resolution of conflicts seems to be [Douence R. et al. 2002]. The authors hold that the treatment of the conflicts among aspects should be carried out in a separated way from the aspects definition. The programmer solves the interactions using a dedicated composition language. The solution is based on a generic framework for AOP that is characterized by a very expressive language of crosscutting cuts, static conflicts analysis and a linguistic support for the resolution of conflicts.

An approach to detect and to analyze the conflicts caused by the capacities of AspectJ is presented in [Storzer M. and Krinkle J., 2003]. This approach provides mechanisms to modify the hierarchical structure of the classes (declaration declares parents) and to introduce new members to the classes (methods and attributes). This

work is based on traditional techniques of programs analysis and it is only led to the detection of a class of conflicts.

An analysis model to detect conflicts among crosscutting concerns is presented in [Tessier F. et al. 2004]. The purpose of the authors is to identify the interactions among aspects during the modelling, and to provide a formal method that allows developers to detect the conflicts by means of successive refinements. The main objective is to achieve the detection of conflicts as soon as possible (early detection of conflicts) and to offer certain level of prediction of the generated impact by the insert of new aspects. This work is restricted to the detection of the conflict, in spite of being carried out in early stages of the development.

A precedence model of AspectJ (sequential), used to establish the execution order of advice, when they are associated to the same join-point is improved and optimized in [Yu Y. and Kienzle J., 2004]. The representation of the model in a precedence graph, leads to a model of concurrent precedence. This work tries to improve the conflict resolution mechanism especially for AspectJ.

LogicAJ [ROOTS, 2005] provides interferences aspect-aspect analysis for AspectJ that includes capacities for: (a) identifying a well defined interferences class, (b) determining the execution order free of interference, (c) determining the weave algorithm more convenient for a group of given aspects. The analysis of interferences is independent from the base programs to those that the aspects are referred to and independent from the aspect analyzer's additional annotations.

Programme Slicing is a technique that aims to the extraction of program elements related to a computation in particular. This approach is proposed to analyze the interactions among aspects, since it can reduce the code parts that are needed to analyze in order to understand the effects of each aspect [Monga M. et al. 2003].

A very interesting work is Reflex [Tanter E. and Noye J., 2005], a tool that facilitates the implementation and composition of different aspect oriented languages. This work proposes a model which provides a high level of abstraction to implement the new aspect languages and to support the detection and resolution of conflicts. Reflex consists basically of a kernel with 3 layers architecture: (1) a layer of transformation in charge of the basic weave with support for the structural modification and of behavior of base programs; (2) a composition layer for the detection and resolution of interactions and (3) a language layer, for the definition of the aspect language modulation. The detection of interactions follows the outline proposed for [Douence R. et al. 2002] and it is a static approach. The interactions are not detected at execution time. There are two ways of solving an interaction: (1) to choose of the interactions the aspect that will be applied in the execution (2) to order and to nest the aspects for the execution. This work advances that an AOP tool should manage conflicts and consequently it provides a specific layer of the kernel for this purpose.

A way to validate the orders of precedence formally, (the orders between aspects that share join-points), is presented in [Pawlak R. et al. 2005]. This work introduces a small language, CompAr, in which the user expresses the effect of the advice that is important for aspect interaction, and properties that should be true after the execution of the advice. The CompAr compiler checks a given advice order, that does not invalidate a property of an advice.

An interaction analysis based on Composition Filters is proposed in [Durr P. et al. 2005]. In this work it is detected when one aspect prevents the execution of another, and can check that a specified trace property is ensured by an aspect.

The use of rules as strategy or mechanism for handling of conflicts has been proposed in several recent works. [Kessler B. and Tanter E., 2006] presents a logic-based initial exploration where facts and rules are defined for the detection of interactions in Reflex. In [Nagy I. et al. 2006], it is proposed a constraint-based, declarative approach to specify the composition of aspects which share join-points. The implementation of this model requires the extension of AOP Language in several aspects: join-point constructs, advice constructs, declarations statements, etc. The restrictions of AOP Languages, such as the precedence statement of AspectJ, are not overcome. Because of the resolutions are applied in the body of the aspects.

The Model of Interactions [Charfi A. et al. 2006] is similar to the Model of Associations. The authors proposed to represent the aspects such as components and the aspectual relationships in "Interaction" units. The Interactions are a mixture of associations and explicit rules. An Interaction Specification Language (ILS) is provided. This language provides a set of operators, which defines weaving rules ("merging"). The operators apply different resolutions to the conflicts: conditional execution, mutual exclusion and order, and the sequential or concurrent composition. All the interactions on the shared join-points are mixed in an only interaction (it unifies advice after and before). The interactions are defined in singular form, after an automatic mixture process generates the final interaction.

Another similar approach is Aspect-Markup Language (AML) [Lopes C. and Ngo T., 2004]. In AML, an aspect-oriented program consists of three elements: core modules, aspectual modules and aspect bindings. Core and aspectual modules are developed in the base language; these entities are similar to classes and aspects in MEDIATOR. The aspect bindings are specified in AML, and are similar to a set of associations. All aspectual relationship (pointcuts, advice, etc.) related with an aspectual module are encapsulated in an aspect binding. Aspect binding is XML-based binding specification. It is provided binding instructions that determine how core and aspectual modules are unambiguously composed to produce the final behaviour. In general, an aspect binding file contains a mixture of predefined XML elements called core elements and user-defined XML elements called custom elements. The authors do not highlight specific syntactic and semantic mechanisms for handling of conflict.

6. Conclusions and Future Work

In this article a novel approach to develop aspect-oriented application has been presented. The Model of Associations is based on following premises: (i) the crosscutting concerns logic is implemented in aspect units; (ii) the mechanics to relate the aspects with functional components (pointcuts, advise, join-point) are defined in associations; (iii) the conflict resolutions are declared in rules. These strategies allow us to improve the handling of conflicts and the aspects reuse. The weaving strategy makes the code of aspects and classes (source and compiled) remain intact. They are contaminated neither by the associations, nor by the conflicts resolution applied. These features make this approach flexible, effective and powerful in order to handle the conflicts.

In Section 2, it is identified several sceneries in which the AspectJ model presented severe restrictions to handle conflicts. In this work all these problems are solved using Model of Associations.

The implementation of Model of Associations supports the use of wildcards and the passing of the context in the associations. The main limitation of the implementation of our approach is that it does not support aspects instances.

The future work is addressed to apply the Model of Associations to real dynamic context. Now we are exploring the possibilities on dynamics language.

This work was partially supported by the Universidad Nacional de la Patagonia Austral, Santa Cruz, Argentina and Research Project PICT 32079 (ANPCYT).

References

Charfi A., Riveill M., Blay-Fornarino M., Pinna-Dery A. (2006) "Transparent and Dynamic Aspect Composition", Workshop on Software Engineering Properties of Languages and Aspects Technologies, VII AOSD – Germany – 2.006

Chiva S. (1.998) "Javassist – A Reflection – based Programming Wizard for Java". Proceeding of the ACM – OOSPLA. Workshop on Reflective Programming in C++ and Java. Canada.

Dijkstra E. W. (1.976) "A Discipline of Programming. Prentice Hall".

Douence R., Fradet P. and Südholt M. (2.002) "Detection and Resolution of Aspect Interactions". TR Nº4435, INRIA, ISSN 0249-6399, France.

Durr P., Staijen T., Bergmans L. and Aksit M. (2.005) "Reasoning about semantic conflicts between aspects". In K. Gybels, M. D'Hondt, I. Nagy, and R. Douence, editors, 2nd European Interactive Workshop on Aspects in Software.

Hursch W. and Lopes C. (1.995) "Separation of Concern". TR. NU-CCS-95-03, Northeastern University.

Kessler B. and. Tanter E. (2.006) "Analyzing Interactions of Structural Aspects". Workshop AID in 20th. European Conference on Object-Oriented Programming (ECOOP). France.

Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J. and Griswold W. (2.001) "An Overview of AspectJ". In J. L. Knudsen, editor, Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP), Num. 2072 in LNCS, pp 327–353, Springer-Verlag. Hungary.

Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J. and Irwin J. (1.997) "Aspect-Oriented Programming". In Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP), Finland.

Lopes C. and Ngo T. (2004) "The Aspect Oriented Markup Language and its Support of Aspect Plugings", Technical Report # UCI-ISR-04-08, Institute for Software Research – University of California.

Monga M., Beltagui F. and Blair L. (2.003) “Investigating Feature Interactions by Exploiting Aspect Oriented Programming”. TR N comp-002- Lancaster University, England.

Nagy I., Bergmans L. and Aksit M. (2.006) “Composing Aspects at Shared Join Point”. Workshop AID in 20th. European Conference on Object-Oriented Programming (ECOOP). France.

Pawlak R., Duchien L., and Seinturier L.(2.005) “CompAr: Ensuring safe around advice composition”. In FMOODS 2.005, Vol. 3535 of LNCS, pp 163–178.

Piveta E. and Zancanella L.(2.003) “Aspect Weaving Strategies”. Journal of Universal Computer Science, Vol.9, Num. 8.

Pryor J., Diaz Pace A. and Campo M. (2.002) Reflection on Separation of Concerns. RITA. Vol.9. Num.1

ROOTS (2.005) “LogicAJ – A Uniformly Generic and Interference-Aware Aspect Language”. http://roots.iai.uni-bonn.de/researh/logicaj/ .

Storzer M. and Krinkle J. (2.003) “Interference Analisys for AspectJ”. FOAL: Foundations of Aspect-Oriented Languages, USA.

Tanter E. and Noye J. (2.005) “A versatile kernel for multi-language AOP”. In Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering, Vol. 3676 of LNCS, pp 173–188, Springer-Verlag. Estonia.

Tessier F., Badri M. and Badri L. (2.004) “A Model-Based Detection of Conflicts Between Crosscutting Concern: Towards a Formal Approach”. International Workshop on Aspect – Oriented Software Development, China.

Yu Y. and Kienzle J. (2.004) “Towards an Efficient Aspect Precedence Model”. Proceeding of the 2.004 Dynamic Aspects Workshop, pp 156-167, England.

This work was partially supported by the Universidad Nacional de la Patagonia Austral, Santa Cruz, Argentina.