Associations in Conflict

SANDRA I. CASAS[1]
J. BALTASAR GARCÍA PEREZ-SCHOFIELD[2]
CLAUDIA A. MARCOS[3]

[1]Unidad Académica Río Gallegos, Universidad Nacional de la Patagonia Austral,
Río Gallegos, Argentina, 9400
lis@uarg.unpa.edu.ar
[2]Departamento de Informática, Universidad deVigo,
Orense, España, 32004
jbgarcia@uvigo.es
[3]Instituto de Sistemas de Tandil, Universidad Nacional del Centro
Tandil, Argentina, 7000
cmarcos@exa.unicen.edu.ar

Abstract. Aspect-Oriented Programming (AOP) is an emergent technology for the modularization of crosscutting concern. The most used approach to support the AOP paradigm is based on the aspect notion, as the basic unit that contains the crosscutting concern logic and a method that relates it to the functional components (pointcuts, advices, join-points). This mechanism, popularized by tools like AspectJ, restricts and limits the handling of conflicts among aspects. In this work a flexible, wider and powerful approach is presented. This strategy is based on two main mechanisms: definition of associations and explicit rules. The approach presented is implemented in a prototype denominated MEDIATOR.

Keywords: AOP, Conflicts, Associations, Rules, AspectJ.
(Received September 30, 2006 / Accepted January 03, 2007)

1 Introduction

The term *concern* generally refers to concepts within an application that are relevant to that Application. This basically means that, in every applications, there is a core part (the business logic), and many accessory parts (concerns) which deal with the user interface, security, and other concerns. However, the code related to the latter is mixed with the bussiness logic of the application. The Separation of Concern (SoC) [9] principle states that a given problem involves different kinds of concerns, which should be identified and separated to successfully cope with complexity, and to achieve the required engineering quality factors such as robustness, adaptability, maintainability and reusability [20].

Therefore, two concern types are identified: the core concerns and the crosscutting concerns. Generally, the core concerns represent the basic or primary functionality of a system. The crosscutting concerns represent the secondary or peripheral functionality. For example Logging, Persistence, Security, Authentication, Synchronization, Error Handling, are crosscutting concerns.

The conventional programming techniques give appropriate support for the implementation of the core concern. However, the implementation of the crosscutting concern using these techniques generates the code for these concerns mixed and scatted through the core concerns code.

An Aspect-Oriented Programming (AOP) [23] tool provides mechanisms to encapsulate the crosscutting concern and to integrate the aspects without modifying the components of basic functionality. Most of the AOP tools are extensions of conventional languages. These extensions incorporate mechanisms to implement the cuts and add constructs that describe the semantics and the aspects behavior. After aspects codification, a weaving process integrates the aspects with the components of basic functionality, generating the final application [28]. An Aspect is a unit of code and it is composed of different constructions like methods, attributes, introductions, declarations, cuts, etc. The cuts give sense to the aspects, since they are the elements that link the aspects with components of basic functionality. The fundamental devices of a cut are: join-point, primitive cuts and advices. Although the device characteristic varies among these tools, their purpose is essentially the same one. This model has been imposed by AspectJ [22], the most diffused, popular and used AOP tool. The AspectJ model has also been replicated by many other AOP tools [13][14][15][19][27].

It is possible that two or more aspects can cause a conflict, during the weaving process. *Conflicts* may occur if two or more aspects compete for activation [29]. The activation of certain conflicts can cause undesired, unpredictable and inconsistence behavior. Handling of conflicts requires solutions for two problems: conflict detection and resolution. However, in most of the AOP tools the conflicts detection is a manual task and the resolution possibilities are very poor and restricted [4].

In this work an approach to conflict solving among aspects is presented. The solution provided involves detection and resolutions of these conflicts, while based on two mechanisms: the definition of associations and explicit rules. These strategies make the approach very flexible, effective and powerful. Other objectives, such as aspect reuse, are also achieved. These strategies have been implemented in a prototype denominated MEDIATOR.

This work is structured as follows: section 2 is dedicated to the analysis of the causes and consequences of the aspect conflicts, while in sections 3, 4 and 5 the proposed solutions is deeply discussed. Section 6 exposes the related works, while section 7 presents the final conclusions and future work.

2 Conflicts: Causes and Consequences.

Conflicts may occur if two or more aspects compete for activation [29], this phenomenon is also known as interaction [10] or interference [31]. There may be different types of hidden dependencies or conflicts between aspects, and each one will require a different solution in order to avoid undesired or unpredictable behaviour.

In aspect-oriented applications, the same component may be associated to more than one aspect. This would be the case, for example, if an object is associated to an aspect that updates a system log and also to an aspect that defines an specific function for the data-base administrator. When a message is sent to this object and a method that stores data in the data-base is invoked, the two associated aspects will be activated. These situations arise questions: such as the possibility of being able to predict the resulting behavior when both aspects are invoked without any control. For example, a given order in which aspects are invoked could result in undefined behavior, while another order could result in correct behavior without further need of control. It may even be the case that the aspect related to the data-base administrator should be specialized according to the actual data-base being updated at that particular moment; that is, the behavior is dependent on the context.

In these and other cases, it would be desirable for the developer to specify the type of conflict between competing aspects, and to describe the actions to be carried out, determining the priorities and activation policy of the conflicting aspects.

However, the handling of aspect conflicts are hard and complex with the current aspects tools, because of two reasons: first of all, the identification process of conflicts is a manual task, and secondly the possibilities of resolution of conflicts are very restricted.

AspectJ [22][16] is a good candidate to be analyze as for the support for handling conflicts. In this tool, aspects are programming constructs that crosscut the modularity of the basic functional classes of the application in predetermined ways.

As for the devices that AspectJ provides for the handling conflicts it is observed that, in particular AspectJ lacks mechanisms to detect possible conflicts among aspects. The weaver-compiler "ajc" does not inform the possible conflicting situations automatically and it always proceeds in the same way, no matter whether conflicts exist or not.

The responsibility of this task is for developer, needing to take control of code weaving and track generated conflicts. The detection of conflicts can be a simple task if the application manages a reduced quantity of units (classes and aspects), but the complexity of the task grows squarely, while the components of the application increase. Also, there are other factors that even make the detection more difficult: (a) the extension of aspects can introduce potential conflicting situations that can be unobserved and which identification can be difficult to be carried out; (b) the tasks of applications maintenance requires the addition, removal and modification of components introducing new conflicts, as requiring certain policies to be disabled before the elimination of nonexistent conflicts; (c) the conflicts identification is less readable due to the usage of certain constructions (for example the wildcard); (d) when the system is developed by a group of programmers, the detection of the conflicts should be made at the end.

As well as for detection of conflicts, the mechanism for resolution, provided by AspectJ consists of 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 *declare precedence* statement. The semantics is that if an aspect A precedes aspect B, then the advices of aspect A have priority and they are executed before the advices of aspect B.

declare precedence: A, B;

The declare precedence statement presents limitations in the following scenarios:

a) The aspects outline more than a conflict (Figure 2). Each conflict requires different order policies.
In this case, it is necessary that the advice associated to the pointcut A1 of aspect A be executed before that the advice associated to the pointcut B1 of aspect B. Once the declaration "declare precedence: A, B" is added to the source code, it is impossible for the advice associated to pointcut B2 of aspect B to be executed before the advice associated to pointcut A2 of aspect A. This situation is not possible to solve through a mechanism of precedence declarations, because it is related to the aspects and not to the advices or pointcuts.

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

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

b) The order in that aspects should be executed depends on a condition of the system or of the context. In Figure 2 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.

```
aspect A
 {
 declare precedence: A, B if (cond)
 pointcut A1(): call(void CX.met());
 before(): A1()
   { ........ }
 }
aspect B
 {
 declare precedence: B,A if (!cond)
 pointcut B1(): call(void CX.met());
 before(): B1()
   { ........ }
 }
```

Figure 2: Two different declare precedence statements for aspects A and B.

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

c) Two or more aspects outline a conflict that are solved in a certain order in a certain application. If these same aspects are used in another application in which should be executed in a different order, the previous declare precedence statement is not valid. In this situation the aspect that contains the declare precedence should be modified.

In spite of these restrictions, the precedence scheme is the only possibility of conflicts resolution that AOP tools offer to the programmer. Even many other AOP tools with limited conflict detection and solving are available. For example AspectR [17] and phpAspect [18] just do not have a similar mechanism to the one of precedence discussed above for AspectJ. The dynamic weavers µDyner [5] and microDyner [32] do not admit that a join-point is associated to more than an advice, pointcut or aspect.

3 Crosscutting Concern: Between Associations and Aspects

To solve the inconveniences discussed in the previous section, a different approach is adopted for crosscutting concerns implementation. The aspect is an independent unit composed by a group of methods and attributes and encapsulates specific crosscutting concerning logic. Associations are entities defined in a separated way, instead of being tied to aspects, linking aspects with classes. That is to say, an association describes a relationship between an aspect and a class.

For example, in Figure 3 the association LogAcc is defined, relating the Logging aspect with the Account class. It establishes that every time that the setBalance(float amount) method of Account class is invoked, the loogedOperations() method of Logging aspect is executed immediately afterwards.

```
association LogAcc
{
 call void Logging.loogedOperations();
 after void Account.setBalance(float amount);
}
```

Figure 3: Association among aspect Logging and class Account

Looging.loogedOpearations() can be related to other functional components and Account.setBalance() can be related to other aspects, other associations will be defined when needed. An association is always a one-to-one relationship.

This approach allows the aspects to be independent of the systems in which they are used and they can be more reusable. But more importantly, the approach will facilitate the handling of associations in an isolated particular way. From within this mechanism, a conflict does only happen when two or more associations define the same relationship type for the same functional component. Conflicts handling should be applied over associations.

4 Explicit Rules to Rescue Conflicts

The conflicts identification process analyzes the application associations evaluating the functional component that they affect and the type of relationship. When two or more associations coincide in these elements a conflict is created. The detection process generates a group of conflicts and it is automatic. A conflict (K) is a group of n associations (A):

$$K = (A_1, A_2, \ldots A_n) \qquad n > 1$$

An association can participate of a unique conflict.

The definition rule allows to specify a particular resolution for a conflict. In this way a rule is another mechanism of the whole approach for develop the AOP applications, as it involves classes, aspects and associations.

A rule establishes a condition and an action. Each conflict is presented as condition of the rule and a resolution category is indicated as action. The rule format is indicated in figure 4.

Rule:	Id_rule
Condition:	$(A_1, A_2, ..., A_n)$
Action:	$R(A_1, A_2, ..., A_n)$

Figure 4: Rule Format.

4.1 Wider Resolution of Conflicts

A wide strategy of resolution of conflicts means that multiple and varied methods are available in order to solve a conflict. The resolution categories that can be applied are classified in basic and combined. In Table 1, the basic categories are described and exampled (In the example column, A and B represent associations).

Table 1: Basic Resolution Categories.

Basic Categories	Action	Example
Order	Defining an execution order for associations in conflict.	order (A, B);
Inverse Order	Defining an execution reverse order for associations in conflict.	inverseOrder (A, B);
Optional	Defining a optional condition execution over associations in conflict. This condition can be a system or context policie.	if (cond) A; else B;
Exclusion	Defining execution exclusion (removing) over some associations in conflicts.	excluded (A);
Nullity	Defining nullity execution (removing) over all associations in conflicts.	anulled (A, B);

These categories of resolution of conflicts are inspired by a taxonomy proposed in Alpheus [30] and used in Astor [3]. New categories are defined starting from the combination of basic categories. In this way more resolution possibilities are available. In Table 2, the combined resolution categories are indicated and exampled (In the example column, A1, A2 and A3 represent associations).

Table 2: Combined Resolution Categories

Combination	Example
Order-Nullity	order (A1, A2), anulled (A3),
Optional-Order	if (cond) order (A1, A2); else order (A2, A1);
Optional-Order-Exclusion	if (cond) order (A1, A2); else excluded (A2, A1);
Optional – Order - Nullity	if (cond) order (A1, A2, A3); else

	order (A2, A1);
	anulled (A3);

Optional - Order - Nullity - Exclusion	if (cond)
	excluded (A1, A2, A3);
	else
	order (A2, A1);
	anulled (A3);

The capacity to combine resolution categories overcomes the conventional based-precedence schemes. These new procedures to solve conflicts represent a very powerful and flexible mechanism. The definition of rules impacts in the weaving process in a decisive way. This process will be explained here below.

5 The Weaving Process.

The weaving process integrates the aspects with the classes to build the final application [28]. The objective is to maintain the associations and the logic of resolution of conflicts (rules) as separated and isolated as possible from classes and aspects. This is the objective to which the design of the weaving strategy was aimed.

The proposed strategy requires classes, associations and rules to participate in the weaving process, apart from aspects, which are explicitly not involved in this process. 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 5 the *link_Met1()* linking method is shown, generated from the LoggAcc association (relates Account class to Logging aspect).

```
association  LoggAcc
{
 call void Logging.loogedOperations();
 after void Account.debit(float amount);
}
--------------------------------------------
class Link_Class1
{
  -----
 void static link_Met1()
 {
   Logging.loogedOperations();
 }
 -----
} // end linking class
```

Figure 5: Linking method generated from association LoggAcc

The *link_Met1()* method invokes the execution of the aspect method, defined in the LoggAcc association.

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 relationship type. This process has been implemented by means of the Javassist API [6][7][8]. Figure 6 (according to LoggAcc association) illustrates how debit() method would be linked to the Link_Class1.link_Met1() linking method, when the Account class is loaded by the JVM (Java Virtual Machine). Thus, the weaving process does not modify the source code or the bytecode of classes.

```
class  Account {
  ---
  void debit()
  {
   ---
   Link_Class1.link_Met1();
  }
}
```

Figure 6: Modification of bytecode on-the-fly

The previous example is validated for associations free of conflicts. When the associations to weave results in a conflicts, it is basically proceeded in the same way. In the first phase the rules are required besides the associations. The linking method concentrates the logic of resolution of conflict. For example, in Figure 7 the LoogAcc and StatisAcc associations have been defined (Logging and Statistic are aspects, Account is a class), along with the rule R, which will be used to try to solve any possible conflict. The LoogAcc and StatisAcc associations are in conflict. The Rule R has been defined to solve this conflict applying a category combined optional-order.

```
association  LoggAcc
{
 call void  Logging.loogedOperations();
 after void Account.debit();
 }
--------------------------------------------
association  StatisAcc
{
 call void Statistc.register();
 after void Account.debit();
}
--------------------------------------------
Rule R
Condition: LoggAcc, StatisAcc;
Action:
   if (n)
     order (LoggAcc, StatisAcc);
   else
     order (StatisAcc, LoggAcc);
```

Figure 7: Conflict between LoggAcc and StatisAcc associations and Explicit Rule R.

In the compilation phase, both associations are merged in a unique linking method. The method encapsulates the logic of resolution of the conflict. This logic comes from the category resolution of conflict in the defined rule R. In Figure 8, the *link_Met2()* linking method has been automatically generated starting from the rule R.

```
class Link_Class1 {
 -----
 void static link_Met2(boolean n) {
    if (n() )   {
      Logging.loogedOperations();
      Statistic.register();
    }
    else   {
      Statistic.register();
      Logging.loogedOperations();
    }
 }
 -----
```

Figure 8: Linking method of associations in conflicts.

In summary, the functional components affected by some association are inserted a call to a linked method, according to the type of relationship of the association. The linking method, consists of the invoking of 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) classes do not have any knowledge about what aspect cuts them; (ii) aspects preserve their original state and they can be associated to any other functional component (iii) conflict 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 compilate the other units (classes and aspects).

The first phase, the creation of the linking class, could be carried out in load-time. However, for efficiency reasons it is more convenient to be carried out as part of the compilation process.

5.1 Performance

Performance is a decisive factor in certain applications. For that reason, it is considered important to know and to evaluate the impact of the weaving strategy. An experiment allowed to measure and to compare the times of execution of an application developed under our approach was therefore very important.

The basic application of the experiment is composed of two classes: AccountManager and Account. AccountManager manages a collection of Account object, to update the balance of the same ones. Figure 9 represents the application of the experiment.

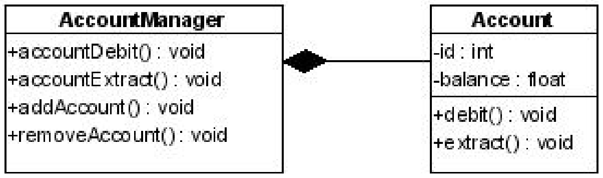

Figure 9: Experiment Basic Application.

Two non-functional requirements are now added: Logging and Statistic. Both crosscutting concerns register different information after debiting and extracting operations are executed.

The application was implemented in 3 different versions: (1) implementation in Java; (2) implementation in AspectJ; (3) implementation in MEDIATOR. The execution of each version was carried out in 3 tests: (A) 10,000 instances of Account, 100,000 aleatory calls to the debit() method and 100,000 aleatory calls to the extract() method; (B) 50,000 instances of Account, 500,000 aleatory calls to the debit() method and 500,000 aleatory calls to the extract() method; (C) 100,000 instances of Account, 1,000,000 aleatory calls to the debit() method and 1,000,000 aleatory calls to the extract() method.

Timing was taken using the standard System.currentTimeMillis() method, which has a resolution of 10ms or less, depending on the operating system. The environment of the experiment was the following:

- Machine: Intel Celeron CPU 1.80 GHz. 248 MB RAM
- Operating System: Windows XP
- JVM: Java 1.5.0 (J2SE 5.0)
- AspectJ 1.5
- Javassist 3.2

The experiments results are showed in Figure 10.

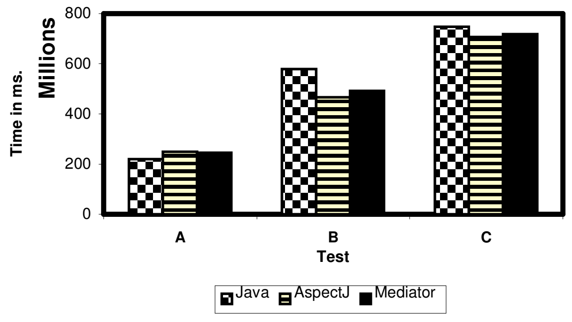

Figure 10: Chart of Test A, B and C over 3 different implementations.

The best performance in test A was obtained by Java version. However, the difference with other versions was less than 13,5%. Tests B and C present a few surprises because it shows that the Java version spends more time than AspectJ and MEDIATOR versions. Also in these tests AspectJ version is faster than MEDIATOR version. In test B AspectJ spent 6% less and in test C 2% less. The cause of this minimun advantage in favor of AspectJ, it can be that the AspectJ weaver inserts hook in the classes to the aspects. The cuts are explicit and direct. In MEDIATOR, the used method is less direct and obvious, since the classes are linked to the methods of the linking class, and these are linked to the aspects. Another factor to keep in mind is the association aspect-object coded in AspectJ. In the AspectJ version of Test A, B and C, the aspects use the default association. By default, only one instance of an aspect exists in a virtual machine (VM)-much like a singleton class. If another association type is applied, such as perthis, pertarget or perflow, which consume more resources, the AspectJ performance can be less.

We can not compare the execution when a resolution combined category has been applied, because AspectJ do not support it. For these reason we consider that this study is not complete. Even so, it guides us about MEDIATOR performance. The preliminary observation is that MEDIATOR is competitive, but consistently slower than AspectJ.

6 Related Works

Several works have been developed in order to detect and solve conflicts situations among aspects. Each of them presents their own classification and the strategies to solve the conflicts. The resolution of the conflicts in most of the cases is the developer's responsibility, which means that, the developer has to analyze the situation and decide the best resolution strategy.

The first directly related work with the detection and resolution of conflicts has seemingly been [10] [11]. The authors hold that the treatment of the conflicts among aspects should be carried out in a separated form from the aspects definition. A model of three-phases intends for the programming of multiple aspects: (i) Programming: The aspects that are part of an application are written independently and possibly for different programmers; (ii) Analysis of conflicts: An automatic tool detects the interactions among aspects and it returns the results to the programmer; (iii) Resolution of conflicts: The programmer solves the interactions using a dedicated composition language. The result of this phase can be again checked in phase (ii). 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 interferences (conflicts) caused by the capacities that AspectJ is presented in [33]. 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 concern is presented in [35]. The purpose of the authors is to identify the interactions among aspects in 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 impact generated by the insert of new aspects. This work is limited to the detection of the conflict, although in early stages of the development.

A precedence model of AspectJ (sequential), used to establish the execution order of advices, when they are associated to the same join-point is improved and optimized in [36]. The representation of the model in a precedence graph, leads to a model of concurrent precedence. This work tries to improvement the resolution of conflicts mechanism especially for AspectJ.

LogicAJ [31] 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 (only the aspects are necessary for the analysis) 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 [24]

A very interesting work is Reflex [34], a tool that facilitates the implementation and composition of different aspects oriented languages. This work proposes a model which provides a high level of abstraction to implement the new languages of aspects and to support the detection and resolution of conflicts. Reflex consists basically on a kernel which a 3 layers architecture: (1) a layer of transformation in charge of the basic weave with support for the structural modification and of behaviour of base programs; (2) a composition layer for the detection and resolution of interactions and (3) a language layer, for the definition of the language aspects modulation. The detection of interactions follows the outline proposed for [10] and it is limited to a static approach of the interaction of aspects. 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 a AOP tool should manage conflicts and consequently it provides a specific layer of the kernel for this purpose, but it imposes very restricted resolution methods.

A way to formally validate precedence orderings between aspects that share join-points is presented in [26]. 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 can then check that a given advice ordering does not invalidate a property of an advice.

An interaction analysis for Composition Filters is proposed in [12]. In this work 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 conflicts has been proposed in several recent works. [21] presents a logic-based initial exploration where facts and rules are defined for the detection of interactions in Reflex [34]. In [25], it is proposed a constraint-based, declarative approach to specify the composition of aspects at shared join-points. The ordering constrains and control constrains are similar to the basic categories of resolution of our approach. The ordering and control constraint can not be combined. The implementation of this model requires the extension of AOP Language in several aspects: join-point constructs, advices constructs, declarations statements, etc. The restrictions of AOP Languages, like the limitations of precedence statement (section 2) are not overcome, because the resolutions are applied in the aspects body.

7. Conclusions and Future Work

In this article a complete approach to detect and to solve conflicts among aspects has been presented. The main strategies applied are: (i) Implementation separated from the crosscutting concern in aspects and associations. These mechanisms allow us to isolate and to individualize the resolution of conflicts; (ii) the precedence scheme is overcome by the explicit rules definition. The possibility to apply several categories resolution (basic and combined), are a unique solution in AOP context; (iii) the weaving strategy makes the aspects remain intact. They are not contaminated of the associations, by no one of the conflict resolution methods applied.

These characteristics make this approach very flexible, effective and powerful in order to handle conflicts. The SoC principle also stays. Additional benefits are achieved as the reusability because the aspects do not contain any information about the functional components that they cuts.

The strategies describe in sections 3, 4 and 5 have been implemented in a research prototype denominated MEDIATOR. MEDIATOR allows implementation of AOP application in Java. MEDIATOR is simple and easy. It extends Java with two new units: associations and explicit rules. The MEDIATOR relationships are still a reduced group (call-after, call-before, set-after and get-after). We are working in the extension of this group and also allowing the use of some wildcards.

The disadvantage of our approach arises in very complex scenarios in which dozens of conflicts can be activated and they require similar solutions. The explicit rules definition can be a tedious, repetitive and a task prone to errors. In these cases, the best strategy will be to use symbolic and general rule definition. In this way, each symbolic rule can solve subsets of conflicts. We are working on the implementation of rule expert system embedded in MEDIATOR. The rule expert system allows us to detect and to solve conflicts by means of the definition of symbolic rules [1][2].

Acknowledgements

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

References

[1] Casas S., García Perez-Schofield B., Marcos C. *Detección y Resolución de Conflictos entre Aspectos basado en un Sistema Experto de Reglas.* IX Workshop Iberoamericano de Ingeniería de Requerimientos y Ambientes de Software, Pp 509-512. ISBN 978-950-34-0360-0, Argentina, 2.006.

[2] Casas S., García Perez-Schofield B., Marcos C. *Gestión de Conflictos entre Aspectos mediante un Sistema Experto de Reglas.* XXXII Conferencia Latinoamericana de Informática, Chile, 2.006.

[3] Casas S., Marcos C., Vanoli V., Reinaga H., Saldivia C., Pryor J., Sierpe L. *ASTOR: Un Prototipo para la Administración de Conflictos en AspectJ.* XIII Encuentro Chileno de Computación, Jornadas Chilenas de Computación, Chile, 2.005.

[4] Casas S., Reinaga H., Sierpe L., Vanoli V., Saldivia C., Pryor J. *Clasificación y Resolución de Conflictos entre Aspectos.* VII Workshop de Investigadores en Ciencias de la Computación, Argentina, 2.005.

[5] Chen Y. *Aspect – Oriented Programming (AOP): Dinamic Weaving for C++.* Master thesis, Vrije

Universiteit Brussel and Ecole des Mines de Nantes, France, 2.003.

[6] Chiva S. *Javassist – A Reflection – based Programming Wizard for Java.* In Proceeding of the ACM OOSPLA´98 Workshop on Reflective Programming in C++ and Java.1998.

[7] Chiva S. *Load-time Structural Reflection in Java.* ECOOP 2.000 – Object-Oriented Programming, LNCS 1850 - Springer Verlag - pp 313-336, 2.000.

[8] Chiva S., Nishizawa M. *An Easy-to-Use Toolkit for Efficient Java Bytecode Translation*. Proceeding 2nd International Conference of Generative Programming and Components Engineering, LNCS 2830 pp 364-376 – Springer Verlag, 2.003.

[9] Dijkstra E.W. *A Discipline of Programming.* Prentice Hall, 1976.

[10] Duoence R., Fradet P., Südholt M. *Detection and Resolution of Aspect Interactions.* TR N°4435, INRIA, ISSN 0249-6399, France, 2.002.

[11] Duoence R., Fradet P., Südholt M. *A Framework for the Detection and Resolution of Aspect Interaction.* In Proceeding of GPCE 2.002, vol. 2487 of LNCS, Springer Verlag, pp 173-188, USA, 2.002

[12] Durr P., Staijen T., Bergmans L., Aksit M. *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, 2005.

[13] Gal A., Scroder-Preikschat W., Spinczyk O. *AspectC++: Language Proposal and Prototype Implementation.* ACM International Conference Proceeding Series Proceedings of the Fortieth International Conference on Tools Pacific. Vol.10. Australia, 2.002.

[14] Homepage of AspectC: http://www.cs.ubc.ca/labs/spl/projects/aspectc.html

[15] Hirschfeld R.: AspectS - AOP with Squeak. In Proceedings of OOPSLA. Workshop on Advanced Separation of Concerns in Object-Oriented System. USA, 2.001.

[16] Homepage of AspectJ™, Xerox Palo Alto Research Center (Xerox Parc), Palo Alto, California. http://aspectj.org.

[17] Homepage of AspectR: http://aspectr.sourceforge.net/

[18] Homepage of phpAspect: http://phpaspect.org/wiki/doku.php

[19] Homepage of Pythius: http://sourceforge.net/projects/pythius/

[20] Hursch W., Lopes C., "Separation of Concern". TR. NU-CCS-95-03, Northeastern University, 1.995.

[21] Kessler B, Tanter E. *Analyzing Interactions of Structural Aspects.* Workshop AID in 20th.ECOOP. France, 2.006.

[22] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W. *An Overview of AspectJ.* In J. L. Knudsen, editor, Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001), number 2072 in Lecture Notes in Computer Science, pp 327–353, Budapest, Hungary, 2001. Springer-Verlag.

[23] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J. and Irwin J. *Aspect-Oriented Programming.* In Proceedings of ECOOP'97 Finland, 1.997.

[24] Monga M., Beltagui F., Blair L. *Investigating Feature Interactions by Exploiting Aspect Oriented Programming*. TR N comp-002-2.003, Lancaster University, England, 2.003. http://www.com.lancs.ac.uk/computing/aop/Publications.php

[25] Nagy I., Bergmans L., Aksit M. *Composing Aspects at Shared Join Point.* Workshop AID in 20th. ECOOP. France, 2.006.

[26] Pawlak R., Duchien L., and Seinturier L. *CompAr: Ensuring safe around advice composition.* In FMOODS 2005, vol. 3535 of LNCS, pages 163–178, 2005.

[27] Piveta E., Zancanela L. *Aurelia: Aspect oriented programming using reflective approach.* Workshop on Advanced Separation of Concers ECOOP 2.001.

[28] Piveta E., Zancanella L. *Aspect Weaving Strategies*. Journal of Universal Computer Science, vol.9, num. 8, 2003.

[29] Pryor J., Diaz Pace A., Campo M. *Reflection on Separation of Concerns.* RITA. Vol.9. Num.1 2.002.

[30] Pryor J., Marcos C. *Solving Conflicts in Aspect-Oriented Applications*. Proceedings of the Fourth ASSE. 32 JAIIO. Argentina. 2.003.

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

[32] Segura-Devillechaise M., Meneaud J. *microDyner: efficient dynamic weaving of aspects in native running processs.* Languages et Modeles a Objets, pp. 119-133, France, 2.003.

[33] Storzer M., Krinkle J. *Interference Analisys for AspectJ*. FOAL: Foundations of Aspect-Oriented Languages, USA, 2.003.

[34] Tanter E., Noye J. *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, pages 173–188, Tallinn, Estonia, 2005. Springer-Verlag.

[35] Tessier F., Badri M., Badri L. *A Model-Based Detection of Conflicts Between Crosscutting Concern: Towards a Formal Approach.* International Workshop on Aspect – Oriented Software Development, China, 2.004.

[36] Yu Y., Kienzle J. *Towards an Efficient Aspect Precedence Model.* Proceeding of the 2.004 Dynamic Aspects Workshop, pp 156-167, England, 2.004.