Solving Conflicts in Aspect-Oriented Applications

Jane L. Pryor and Claudia Marcos

ISISTAN Research Institute
Facultad de Ciencias Exactas, UNICEN
Paraje Arroyo Seco, B7001BBO Tandil, Argentina Tel/Fax: + 54 – 2293 – 440362 / 3
http://www.exa.unicen.edu.ar/~isistan/
E-mail: {jpryor,cmarcos}@exa.unicen.edu.ar

Abstract. Aspect-Oriented Software Development is an approach which supports the modularisation of crosscutting concerns through the development phases of an application. One of the limitations in the main approaches to AOSD is that they do not provide support for the definition and handling of conflicts that may arise between multiple competing aspects. This paper describes how conflicts are managed by a framework that we have developed for the construction of aspect-oriented applications. In the first place, we describe the main characteristics of this reflective multi-level framework and the tool that instantiates it. We then present our categorisation of different conflicts and how they are defined and handled by our environment for the development of aspect-oriented software.

Keywords Aspect-oriented software development, crosscutting concerns, reflective architectures, aspect conflicts, frameworks.

1. Introduction

Throughout the history of software engineering, from the advent of structured programming through the different languages that provide constructs for the encapsulation of state and behaviour (procedures, classes, monitors, abstract data types, objects), the aim has been to organise and decompose software into primary and comprehensible components. Current methods, notations, and languages concentrate on finding and composing the functional units of an application. However, as non-functional requirements have also added to the scope and complexity of current applications, obtaining and maintaining a separation of concerns through all levels of software development continues to be a problem.

The object paradigm supports the decomposition of a system through modularization by classes or objects, but despite this asset, there is always a dominant decomposition concern when developing software [1]. This dominant concern is usually related to the innate functionality of an application. Consequently, the addition of non-functional features or aspects such as synchronisation, tracing, error handling, persistence, resource sharing, performance optimisation, etc., often leads to invasive changes that do not align with the functional components of the system [2]. As a result, the new feature code is scattered across multiple classes and leads to tangled code, and the desired encapsulation and separation of concerns is lost.

Aspect-Oriented Software Development (AOSD) [3] is a widespread and experimented approach to the separation of concerns [4]. The goal of aspect-oriented software development is to provide explicit support for modularising the crosscutting concerns of a system. In AOSD these overlapping or crosscutting concerns are encapsulated in separate modules called aspects, whose code is woven into the functional components of the system at predetermined join-points. Current approaches and techniques for Aspect Oriented Software Development [5][6][7][8] differ on many issues. Some of these issues include the manner and timing for the composition of aspects with other components, whether aspects may be composed with other aspects, how to improve aspect reusability, and how conflicts among multiple competing aspects are solved.

Computational reflection is a technique that permits a system to observe and modify the properties of its own behavior. It is a solution to the problem of creating applications capable of maintaining, using or changing the representation of their own designs [9]. It seems natural therefore, to consider reflection as an adequate technique for the implementation of systems with crosscutting concerns. By its very own definition, the solving of associations between crosscutting and basic concerns may be handled dynamically, as a system observes its behavior and consequently modifies it at runtime.

Our work on reflective architectures in different problem domains including AOSD [10][11] has led to the development of a framework for the construction of aspect-oriented software. This framework was designed to include the characteristics for aspect-oriented applications that we considered would increment the adaptability, flexibility, and reuse of the resulting software.

We have also constructed a tool to facilitate the instantiation of the framework. This tool, called Alpheus, provides an environment which allows the developer to define all the components of the application and their associations, and to visualize them graphically in UML based notation [12]. From these specifications, the tool generates the Java code of the application.

One of the problems of the current approaches and techniques for AOSD is the lack of support for the definition and handling of conflicts between aspects. When more than one aspect is associated to the same object and these aspects are not totally independent, the system´s behaviour may be unpredictable.

In order to explore a solution to these situations in AOSD, we studied the different dependencies that may arise between competing aspects, and defined categories of conflicts. Our objective has been to permit the developer to define different types of conflicts between components of the software, and to include the treatment of conflicts in our environment for AOSD, including the above-mentioned framework and Alpheus, the tool for specifying and generating the application.

The following section presents the multi-level reflective framework. The categorization of conflicts and how they are defined and handled by the framework is described in Section 3. Section 4 describes Alpheus, the AOSD tool that instantiates the framework. Section 5 presents an example of a Hospital Resource Allocation system, and lastly we present our conclusions.

2. A Framework for the Construction of Aspect-Oriented Software

In a typical object-oriented reflective architecture, the components of the system reside on two different levels: the *base level* which contains the objects that solve a problem and return information about the domain of the application, and the *metalevel* which contains objects that carry out computation about the system itself. The objects on the metalevel are called metaobjects.

The functionality of the metalevel is orthogonal to the base level - i.e., it crosscuts the basic functionality of the system. An association is therefore defined between the base and the metalevel in order to determine the points at which these levels interact with each other. This association is established between the corresponding metaobject of the metalevel and one or more components (classes, methods and/or objects) at the base level. At runtime, when an object at the base level receives a message, the reflection mechanism redirects the control to the associated metaobject.

This two-level hierarchy can be generalised to include many metalevels, becoming an N-level architecture. In this type of architecture, the metaobjects on each level carry out computation on the components of the previous level.

We have specialized this reflective architecture in order to develop a framework for the construction of aspect-oriented applications. The base level contains the components which deal with the basic functionality of the system, and the metalevels contain the components which crosscut those on the inferior levels. The metaobjects implement the aspects of the application, and each aspect is associated to the base components with which it interacts in an orthogonal way.

The reflective framework is an N-level architecture which not only supports the association and composition of aspects with the basic components of the application, but it also supports the association and composition of aspects with other aspects. This permits a greater flexibility than the most widespread aspect-oriented approaches such as the AspectJ platform, which only associate aspects to the basic functional components.

The framework was designed to support the aspect-oriented development of software applications, enhancing fundamental software properties such as flexibility and reuse. The main characteristics provided by this reflective framework are the following:

- encapsulation and facilities for the reuse of groups of aspects with similar or related functionality;
- flexible strategies for the association and activation of aspects;
- support for the definition and solving of conflicts between simultaneously active and competing aspects.

The following sub-sections describe the first two of these characteristics. Our work on the runtime solving of conflicts between aspects is presented in section 3.

2.1. Encapsulation of Aspects

When using aspect-orientation for the development of applications, there are many different concerns which crosscut the dominant functionality of the system and which are implemented as aspects. As mentioned earlier, it is very common that these aspects are related to the so-called non-functional components of an application, such as synchronisation, tracing, error handling, persistence, resource sharing, performance optimisation, etc. It is also often convenient to implement other types of components as aspects in order to enhance the flexibility of the application. This is because the dynamic composition of aspects with other components permits the user to change functionality by simply modifying the association rule between an aspect and a component.

In order to improve the modularity, comprehension, and reuse of the software, we have developed a means to support the encapsulation of aspects. A *plane* is a set of crosscutting components which carry out a specific functionality, as for example the logging of the system. Planes will ease the handling of these groups of components, their reuse, their interaction with the rest of the system, their maintainability, the definition of conflicts, and so on. In general terms, it may be said that the use of planes offers a greater level of abstraction in the handling of those components which have a specific functionality.

Planes have been introduced in order to obtain a clearer separation of concerns, primarily at the aspect level. Their use simplifies the association and composition between crosscutting concerns, facilitating their reuse and improving the overall flexibility of the application.

2.2. Strategies for the Association and Activation of Aspects

In the applications built with the reflective framework, each component may be associated and then composed with many aspects. This implies that when that basic component is invoked, the associated aspects are activated.

It is often the case that what is to be associated to an aspect is not an object with all its methods; it may be all the objects of a class, or a specific instance, or a specific method. For example, if the aspect to be associated to an object is related to the storing of specific data, one would only want to associate that aspect to the methods that deal with the storing of the data. Apart from avoiding unnecessary overhead by reflecting all the methods, the cohesion and comprehensibility of the application is better.

For these reasons, four different reflective strategies which make up a *reflection taxonomy* [13] have been identified:

- *Class reflection*, used when all instances of a class interact with an aspect;
- *Method reflection*, when only one method of a class needs to incorporate an aspect´s functionality;
- *Object reflection*, applied when only a particular object of a class is associated to an aspect;
- *Method-object reflection*, a combination of method and object reflection, used when reflective behavior is added to a particular method of a specific object.

Additionally, the reflection mechanism of the framework permits the developer to determine when and how the meta level is invoked for each association: before the intercepted object, after the intercepted object, before and after, and omitting (or not) the invocation of the originally intercepted method.

3. The Specification and Handling of Conflicts between Aspects

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 which updates a system log and also to an aspect which defines a specific function for the data-base administrator. When a message is passed to this object and a method which stores data in the data-base is invoked , the two associated aspects will be activated. The questions are: Can we predict the resulting behaviour? Should both aspects be invoked? Does the order in which the two aspects are invoked matter? 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 behaviour is dependent on the context.

Conflicts may occur if two or more aspects compete for activation. There may be different types of hidden dependencies or conflicts between aspects, and each will require a different solution in order to avoid undesired or unpredictable behaviour [14][15].

As our framework for the construction of aspect-oriented applications composes aspects dynamically, it was feasible to include the runtime handling of conflicts between simultaneously activated aspects. In the first place we studied the different types of situations in which competing aspects present a conflict, and defined a categorization of types and granularities of conflicts in aspect-oriented applications. We subsequently developed a conflict manager and included it in the reflective framework.

The following sub-sections describe the categories of conflicts, the levels of granularity, and the functionality of the conflict manager component in the framework.

3.1. Conflict Categories

It may be the case that a specific aspect should be executed before others, or that the execution of two aspects may produce an inconsistency that would be avoided if only one of them was executed. In other cases, the corresponding activation of the competing aspects may depend on the current runtime context of the system, as, for example, the value of some state variable or the current thread of control.

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.

In order to incorporate conflict handling in our framework, we first identified and classified the most common types of conflicts. The conflicts may be classified as *static* or *dynamic*. *Static conflicts* can be seen as system restrictions, as they are detected when the user declares the association between the crosscutting concerns. This would be the case when the same aspect is associated twice to the same base object, or when the user wishes to declare an association between two objects of the same plane (i.e. both having similar functionality).

The conflicts which are detected at runtime are called d*ynamic conflicts*, and the developer must specify what actions are to be undertaken when a conflict arises. It is also possible that some situations are not problematic at certain moments of execution, but may be at others. For example, a conflict may be defined between two aspects that are not yet associated to the same base object; thus they do not yet present a conflictive situation, but may do so in the future if they are associated to the same object.

We have identified and implemented different categories of conflict activation policies:

- *InOrder*: the aspects are activated in the order specified by the developer;
- *ReverseOrder*: the aspects are activated in the reverse order to that established by the developer;
- *Optional*: the system itself decides which aspect to activate, according to some pre-established system of priorities or in a random fashion;
- *Exclusive*: only one conflicting aspect is executed;
- *Null*: neither one of the aspects is activated;
- *Context dependent*: the developer adds the code that specifies the activation policy of the aspects.

Although we believe that the above categories of conflicts cover most situations, the framework has been designed to permit the incorporation of other categories. However, this should be unnecessary as context dependent conflicts are coded by the developer and ensure flexibility .

3.2. Levels of Granularity for the Definition of Conflicts

In small systems the declaration and handling of conflicts between specific aspects may be sufficient. However, in more complex and large systems this may become very tedious and difficult to maintain. It is therefore important to abstract the concept of conflicts to a higher level of granularity, permitting their declaration at levels of functionality and not only between specific aspects. With the introduction of planes, the reflective framework supports a more flexible handling of conflicts, easier to define and maintain.

The following levels of granularity between conflicts are supported by the framework:

- *Aspect – aspect*: determines a conflict between two specific aspects.
- *Aspect - plane*: establishes a conflict between an aspect and a plane (all the aspects belonging to that plane).
- *Plane - plane*: specifies a conflict between all the aspects of one plane with respect to all the aspects of the other.
- *Aspect - all*: permits the specification of conflicts "one to many", where a specific aspect has a conflict with all other aspects.

3.3. The Conflict Manager Component

The conflict manager is the component which deals with the detection and solving of conflicts in the reflective framework. In the first place, the user defines the potential conflicts between aspects, specifying the activation policy to be followed if the conflict is detected.

This policy is specified simply by identifying the components involved (e.g. aspect-aspect, aspect-plane), the type of conflict (e.g. inorder) and, in the case of context dependent conficts, the code which specifies the action to be taken.

This specification is supported by Alpheus, the tool which has been built to allow the user to specify all the application components and their associations, and which then instantiates the framework based on these specifications, generating the application code.

The Conflict Manager is the component which is responsible for the runtime handling of the conflicts between aspects. This manager stores instances of the different conflicts specified by the user, and when required, it returns the order of invocation of the aspects to be activated

The Reflection Manager is the component which detects the aspects associated to a method (or object or class) when that object method is invoked, and it passes the list of aspects to be activated to the Conflict Manager. The Conflict Manager checks the existence of conflicts on that list of aspects, and according to the types of conflicts detected and their activation policy, it modifies the list and returns it to the Reflection Manager. This list indicates which aspects are to be activated and the order in which this is to be carried out by the Reflection Manager.

The Conflict Manager has been implemented as a directed graph, where the vertices are represented by the aspects, or aspect pairs, and the conflicts are represented by arcs. The graph is traversed in order to determine the activation order, though only those aspects which are to be activated are taken into account.

It is important to realise that the developer of an application is not aware of the manager components and how they carry out their functionality, as this is inherent to the framework. All the developer must do is specify the components (planes, objects and aspects), their associations and the conflicts. Only those methods which are specific to the application are to be coded The actual application code is automatically generated by Alpheus, the instantiation tool.

4. Alpheus: the Specification and Instantiation Tool

Alpheus is a visual tool designed to aid users in the development of aspect-oriented applications. The tool supports the following tasks:

- the specification of all the components of an application (planes, basic objects, and aspects), and the specification of the associations and conflicts between the components;

- the visualisation of these specifications and components by means of different diagrams and levels of abstraction;
- the generation of the corresponding application code.

The tool allows developers to define the classes of the base application and the aspects using pre-existing ones or creating new ones. A developer may use Alpheus to add aspects to a standard application without modifying its classes, by defining the aspect classes to be added to the application, and by specifying the associations and conflicts between the basic application components and the aspects.

As every aspect with similar functionality belongs to the same plane. It may be possible to specify which particular policy or aspect is to be used in an application by defining the corresponding association, offering great flexibility when implementing a specific feature.

For an aspect to be composed (or activated) at runtime with a component of the application (class, object, or method), the corresponding association between that aspect and the reflected component must be specified with Alpheus. The developer must also specify if the aspect has to be activated *before*, *after*, or *before* and *after*, the invocation of the reflected element.

Alpheus also supports the specification of conflicts between aspects. The developer specifies the pair of conflicting aspects (or planes), and then specifies how the conflict has to be solved by identifying the type of conflict (in order, optional, exclusive, null, or context dependent). In the case of a context dependent conflict, the developer adds the code which is executed when the specified conflict is detected.

In order to facilitate the design and specification of an application´s components and their associations and conflicts, Alpheus provides different views of the system. It is possible to identify two main groups of diagrams: firstly, those related to planes, conflicts and association levels; secondly, those related to the UML diagrams [12].

Within the first group the designer can visualise the different planes, their aspects, the associations between aspects and planes, the declared conflicts, etc. In the second group the tool uses the specifications to generate the component UML diagram with the corresponding interfaces. The tool also generates sequence diagrams showing how the system will behave, including cases when conflicts are detected and indicating the order in which the aspects will be activated. Some consistencies are checked before generating these visualisations; for example, the tool does not generate the sequence diagrams if the developer has omitted a conflict between competing aspects.

These visualisations help developers to check whether the application specification and the aspects and their associations have been well defined, before generating the application code. Once the specification phase has been completed, Alpheus generates the corresponding code plus a list of tasks for the developer to complete when the tool finds that some information is missing for the full generation of the application code. The programmer will only have to implement some application-specific characteristics.

Although this tool for the instantiation of the framework currently generates Java code, it has been designed so that it may be extended in order to generate the application code in other programming languages.

5. A Hospital Resource Allocation System

In this section a Hospital Resource Allocation System is described in order to illustrate the proposed framework and Alpheus, the tool used to develop applications.

The application consists of the allocation of resources in a hospital, with respect to patient admission and surgery scheduling. When a patient is admitted to hospital he/she is assigned to a ward. There are several wards per floor, and nurses are assigned to each floor. A patient has his/her doctor assigned according to his/her illness.

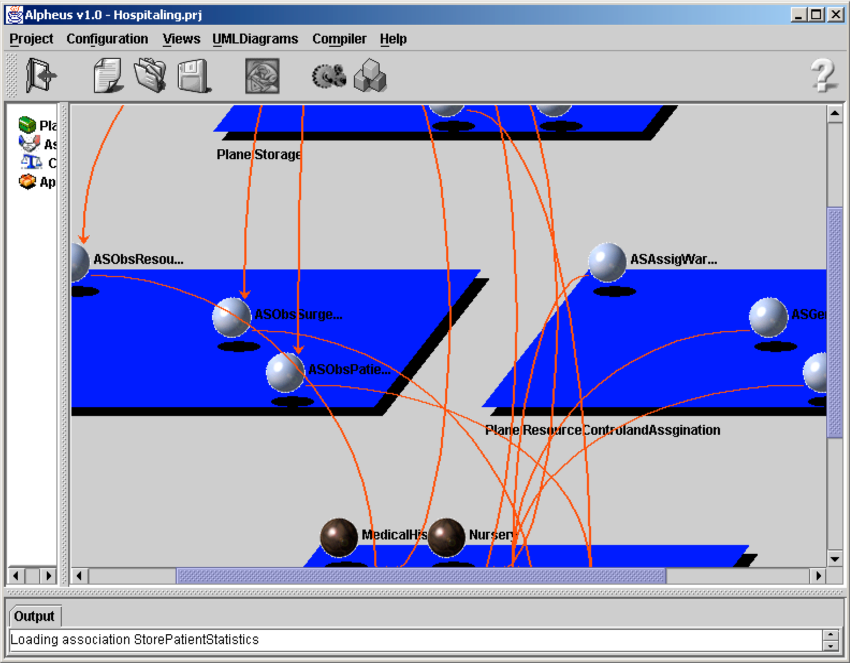

Fig. 1: Planes and Associations of the Hospital Resource Allocation System

When surgery is to be carried out on a patient, both human and non-human resources are allocated and a priority is established according to the urgency. With this information the surgery is scheduled and the resources are reserved. If the required resources have already been reserved for another surgery, the priorities are compared and the surgery with less priority has its resources reassigned.

Information about surgeries, resources, and patients is maintained in order to carry out statistical analysis of the establishment. The statistics include the incidence of different illnesses of the population, how many days a patient is in hospital, the most frequent surgeries, and so on.

Firstly, the components of the application are specified. In this example four planes are identified: *Base*, *Storage*, *Statistics,* and *Resource Control and Allocation* (Fig. 1). The Base plane contains the hospital system, *Storage* maintains the database, *Statistics* generates the statistics of the establishment, and *Resource Control and Allocation* assigns resources to patients and surgeries. The aspects and base classes of all the planes - with their objects and methods – are also specified with Alpheus.

Some conflictive situations can be identified in this application. The first occurs when a patient arrives and is admitted. The method invoked to add a patient is AdmittoHosp of the HospitalSystem class. This is a reflected method which is associated to the ASAssignWard and ASStMedicalHistory aspects, which assign a ward to the patient and open his medical history. The allocation of resources has to be done before the patient medical history is opened and updated, so an order conflict is declared between the ASAssignWard and ASStMedicalHistory aspects (Fig. 2).

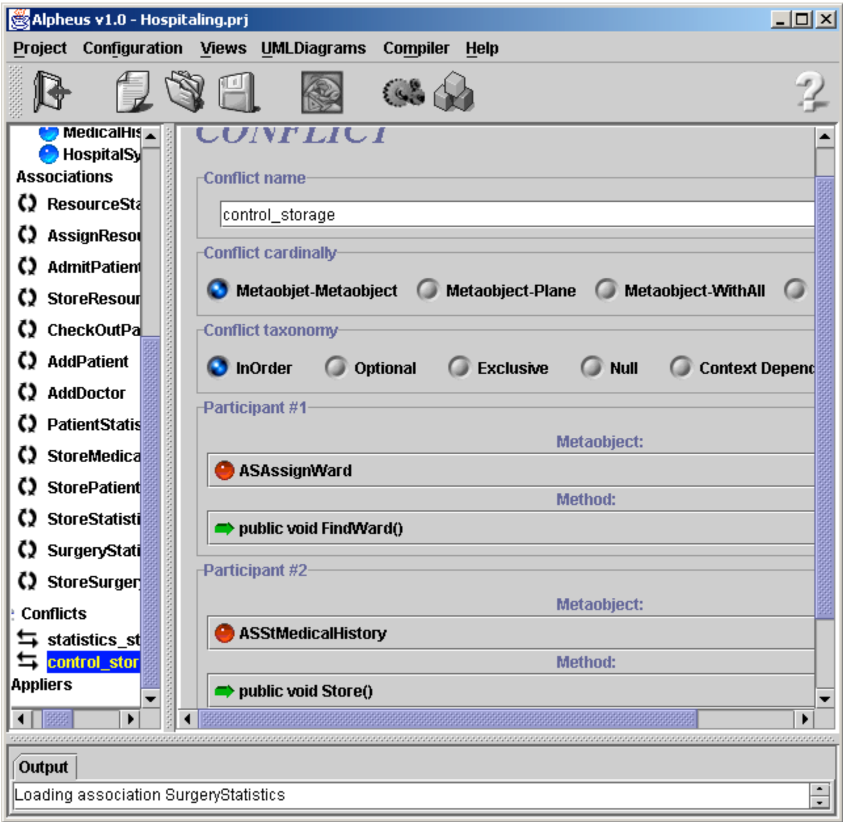

Fig. 2: Definition of a conflict using Alpheus

Another conflictive situation occurs when a laboratory analysis is asked for a patient. The Hospital is divided in two sections, paediatry and adults; both use the same laboratory for their analyses. The difference in the system (the software) used in the two sections, is that paediatry uses an Oracle database and the adults section uses an Informix one. Therefore, when a laboratory analysis is developed, the laboratory has to store the information in the corresponding database according to the section which originated the analysis. In this case, two aspects belonging to the Storage plane compete for activation, the ASStOracle and the ASStInformix, but only one has to be activated according to the section. In this case we have a context-dependent conflict that has to be solved, that is, coded, by the developer. The developer specifies the conflict in a similar fashion to that in Figure 2, and then in the implementation phase, the programmer has to code the way in which the competing aspects should be activated. In this case, the programmer would have to specify that if the laboratory analysis was asked for by the paediatry section, the ASStOracle aspect has to be activated, and otherwise, the ASStInformix aspect should be activated.

An exclusive conflict is also specified. This conflict occurs when the surgery statistics are calculated. We may identify two dependent statistics: one related to patients and the second one to surgeries. In the first one, the objective is to track the patients during the time they are in hospital; the corresponding aspect is ASObsPatient. In the second statistic, the information of the different surgeries, such as the pre- and post-surgery information, is stored for future analysis. The aspect which has this responsibility is called ASObsSurgery. It is possible to see that the information about a patient having surgery is the same as that of the surgery itself, so an exclusive conflict is defined in order to avoid the two aspects being activated simultaneously. In this case, if a base object has the two aspects associated to it, the ASObsSurgery aspect will be activated but not the ASObsPatient. This is because the ASObsSurgery aspect will store the same information as the ASObsPatient aspect, plus some additional details about the surgery.

6. Conclusions

This paper describes the multi-level reflective framework which supports the development of aspect-oriented applications, and Alpheus, the tool which facilitates the specification of all the components and their associations and conflicts. This tool also generates the Java code of the application, and provides the visualization of different UML based diagrams to aid the development process. The resulting aspect-oriented applications are easy to specify and implement.

We have also described the problem when a component in an aspect-oriented system has more than one associated crosscutting aspect, and the conflicts that may occur if these competing aspects are not totally independent. A categorisation of types of conflicts has also been presented. The framework includes a component, the Conflict Manager, which manages and solves the conflicts at runtime according to those specified by the application developer.

Compared to other AOSD techniques, our approach handles conflicts between competing aspects at runtime and in a more comprehensible fashion and with great ease for the developer. An example application has been described in order to illustrate the framework and instantiation tool, and the handling of conficts between aspects.

This dynamic approach to the handling of crosscutting concerns which includes the solving of conflicts between interdependent, simultaneously activated components, is currently being used to develop an extensive application in order to compare it to other approaches and tools.

7. References

[1] Tarr P., Ossher H., Harrison W. and Sutton Jr. S.M. "N Degrees of Separation: Multi-Dimensional Separation of Concerns". Proceedings of ICSE'99.

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

[3] AOSD 2002. 1st. International Conference on Aspect-Oriented Software Development. Gregor Kiczales (Ed.). ACM Press. The Netherlands. April 2002.

[4] Dijkstra E.W. "A Discipline of Programming". Prentice Hall, 1976.

[5] Aksit, M., Bergmans, L., and Vural, S. "An object-oriented language-database integration model: The composition filters approach". In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 1992.

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

[7] Homepage of Hyper/JTM, IBM Thomas J. Watson Research Center. http://www.alphaworks.ibm.com/tech/hyperj.

[8] Pawlak, R., Seinturier, L., Duchien, L., Florin, G. "JAC: A Flexible Framework for AOP in Java". Reflection'01, The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns Kyoto, Japan. September 25-28, 2001, Kyoto, Japan.

[9] Maes, P. "Concepts and Experiments in Computational Reflection". In N.K. Meyrowitz, pages 147-155.

[10] Pryor, J., and Bastán, N. "A Reflective Architecture for the Support of Aspect-Oriented Programming in Smalltalk". Position paper at the Workshop on Aspect Oriented Programming of the European Conference on Object-Oriented Programming (ECOOP), 1999.

[11] Valentino, F., Ramos, A., Marcos, C., and Pryor, J. "A Framework for the Development of Multi-Level Reflective Applications". In Proceedings of the Second Argentine Symposium on Software Engineering (ASSE). Argentina, 2001.

[12] Booch, G., Rumbaugh, J., and Jacobson, I. "The Unified Modeling Language". User Guide. Addison-Wesley, 1999.

[13] Marcos, C. “Patrones de Diseño como Entidades de Primera Clase”. PhD. Degree Dissertation. Universidad Nacional del Centro de la Provincia de Buenos Aires (UNICEN), Facultad de Ciencias Exactas, ISISTAN Research Institute, April 2001.

[14] Nuseibeh, B., Kramer, J. and Finkelstein, A. “A Framework for Expressing the Relationships between Multiple Views in Requirements Specification”. In IEEE Transactions on Software Engineering, October 1994.

[15] Truyen, E., Vanhate, B., Joosen, W., Verbaeten, P., Nørregaard Jørgensen. “Dynamic and Selective Combination of Extensions in Component-based Applications”, in Proceedings of the 23rd International Conference on Software Engineering (ICSE'2001), May 2001, Toronto, Canada.