

# Towards a Formal Detection of Semantic Conflicts Between Aspects: A Model-Based Approach

Francis Tessier

Université du Québec à Trois-Rivières  
C.P. 500, Trois-Rivières, Québec,  
Canada G9A 5H7

Francis\_Tessier@uqtr.ca

Linda Badri

Université du Québec à Trois-Rivières  
C.P. 500, Trois-Rivières, Québec,  
Canada G9A 5H7

Linda\_Badri@uqtr.ca

Mourad Badri

Université du Québec à Trois-Rivières  
C.P. 500, Trois-Rivières, Québec,  
Canada G9A 5H7

Mourad\_Badri@uqtr.ca

## ABSTRACT

Aspect-Oriented Programming is a new promising software engineering paradigm. Aspects are well adapted to capture crosscutting concerns. The new mechanisms introduced by this paradigm allow weaving aspects with different join points in a program. Unfortunately, this flexibility can lead to many unsuspected conflicts between aspects. Moreover, the existing aspect-oriented tools do not detect these conflicts. They can have important consequences in terms of software quality. Managing these conflicts represents an important issue of the aspect-oriented development. In addition, their detection and resolution early in the development process offer many advantages. This paper proposes a formal way, based on model analysis, to detect semantic conflicts between aspects. We introduce two levels of conflicts: direct and indirect conflicts. This paper focuses on the first type of conflict and gives a brief summary on the adopted approach addressing the second one. We believe that the proposed solution could be considered for an interesting extension of the existing aspect-oriented tools.

## Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software and Program Verification – *formal methods, model checking, reliability.*

## General Terms

Algorithms, Design, Reliability, Theory, Verification.

## Keywords

Aspect-Oriented Software Development, Quality Assurance, Aspects, Interactions, Modeling, UML, Formal Description, Conflicts, Detection.

## 1. INTRODUCTION

Aspect-Oriented Programming (AOP) [8] is a new software engineering paradigm, which gains in popularity [10]. Aspects are considered as a component-based programming approach. The major objective of software components is to bring a new degree of separation of concerns. Aspects make it possible to develop modules reducing the dispersion of the code and expressing crosscutting concerns [9]. The basic idea consists of encapsulating these concerns in a new module called *Aspect*. AOP offers several interesting mechanisms, which allow a connection between aspects and classes. The concepts introduced by this paradigm make it possible to express more easily behaviors related to several modules. Aspects are, in fact, an elegant response to

object's limitation in expressing crosscutting concerns. They allow a better implementation of non-functional requirements reducing the dispersion of the code and creating more coherent components in a system [9].

The new mechanisms introduced by the aspect paradigm allow weaving aspects with different join points in a program. This result is a great flexibility in the interactions between aspects and classes, which can lead to many unsuspected conflicts between aspects. Moreover, this increases considerably the complexity of their modeling [6]. The improvement of modularity and concerns locality are in opposition as stated in [6] with the tendency of the aspects to destroy the locality of control flow execution. This generates a serious difficulty in the prediction of the interactions between the various modules of a program and their impacts. The integration of aspects in the code can involve some side effects and affect the internal control flow of a program as mentioned in [11]. These various questions show clearly that the development of effective mechanisms for the detection of semantic conflicts between aspects constitutes an important issue for the maturity of AOP. According to Hanneman [6], the detection and the resolution of the conflicts and the analysis of their impact are two key points for the development and the integration of AOP in the common development. In addition, an early detection of these conflicts would make it possible to reduce costs while preserving a high level of quality. The detection and resolution of these conflicts are not an easy task. Moreover, it's not easy to model and to integrate aspects in an object model. It is in this context and with the objective to ensure the quality of aspect-oriented programs that we have decided to deal with this problem. In this paper, we present a formal approach based on model analysis to detect semantic conflicts between aspects.

The rest of the paper is organized as follows: we give in section 2 a brief summary on the major conflicts, which may occur among aspects. Section 3 presents the introduced levels of conflicts. In section 4, we introduce the principal steps of our technique and illustrate its application on a concrete and simple example. Section 5 presents a short discussion on the adopted approach and the major approaches suggested in the literature in this field. Finally, section 6 gives some conclusions and future work directions.

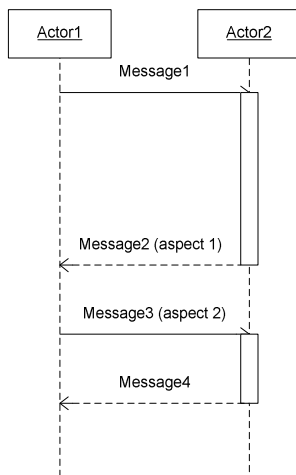
## 2. CONFLICTS BETWEEN ASPECTS

Several types of conflict may occur between aspects. Recent work of the workshop on the analysis of aspect-oriented software [6] allows identifying four principal categories of conflicts:

- **Crosscutting Specifications:** These conflicts are related to the way the aspects are integrated to the classes. The current use of join points can lead to two sub-types of problems: *Accidental join point* and *Accidental Recursion*.
- **Aspect–Aspect Conflicts:** This type of conflict arises when multiple aspects exist in the same system: *Conditional execution*, *Mutual exclusion*, *Ordering*, *Dynamic context dependant ordering* and *Tradeoffs and conflicts at requirements and architectural levels*.
- **Base–Aspect Conflicts:** This type of conflict leads to a circular dependence between basic classes and aspects because the base depends on or refers to an invasive aspect. The base needs to communicate with the aspect.
- **Concern–Concern Conflicts:** A concern may affect another one: *Change of functionality*, *Inconsistent behavior* and *Composition anomalies*.

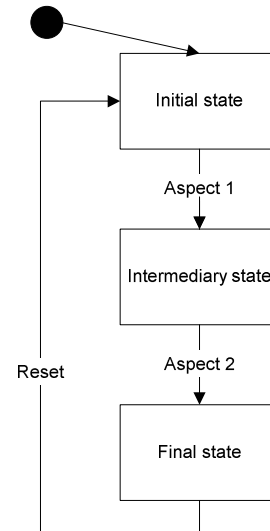
### 3. CONFLICTS LEVELS

In order to classify the conflicts, we introduce in this paper two different levels. The first level represents the direct conflicts. This type of conflict concerns basically aspects with direct connections generating conflicts. For example, two aspects sharing the same join point or an aspect is having a join point in another aspect. These types of conflict are easy to detect comparatively to the second type of conflict, which are the indirect conflicts. This



**Figure 1. Sequence chart.**

paper focuses on only the direct conflicts between aspects. The indirect relationship is not obvious to detect. Basically, to resolve the problem of the indirect level, we use several UML diagrams (sequence, activity, collaboration, state chart). The idea consists of using the state chart diagrams to determine the order of the execution of aspects. The sequence chart is also important to capture the information related to the behavior. Figures 1 and 2 represent an example of interactions between two aspects. The aspects don't share a common join point but we can see that the first aspect can have an impact on the behavior of the second. In the case where the first aspect is never reached, the second aspect



**Figure 2. State chart.**

will not be reached. A complete UML representation of the system is needed to determine all types of conflict. Using these diagrams, we generate the concept of connection tables that we introduce as illustrated in the rest of the paper for direct conflicts the connection tables. These tables will be taken as a basis for the detection of conflicts. In fact, connection tables will store information making it possible to perform behavioral analysis. This will evaluate the potential impact of interacting aspects based on, for example, previous operations and system states, e.g. dynamically, an aspect that alters an object state which is needed (some operations later) by another aspect. Is this case, the first aspect will have an indirect impact on the second one. The idea consists to identify if the execution of an aspect affects others aspects. The following sections focus on the illustration of the direct conflict level.

## 4. METHODOLOGY

### 4.1 Main Steps

Our main objective is to support the detection of conflicts as soon as possible and to cover the main first phases of the development process until the implementation phase. We consider also in our approach the insertion of new aspects in the code. The idea consists essentially to develop a tool managing interference between aspects and detecting their potential conflicts. We are interested in offering a certain level of prediction of the impact generated by aspects insertion. Moreover, our objective was also to develop a simple method as convenient as possible to facilitate its implementation and possibly it's integration in the current development environments. We have taken into account the recommendations illustrated in [6, 7] related to the facilitation of the integration of AOP. These recommendations are essentially related to the fact that AOP does not represent a solution to all problems. Aspects should be perceived as an extension to objects. It's thus necessary to establish the bridges with current technologies. Figure 3 shows the main phases of the adopted approach. The proposed solution allows managing aspects' interference and detecting potential conflicts during the development of aspect-oriented programs as well as their

evolution process. It takes into account the introduction of new aspects and captures their impact on the existing ones.

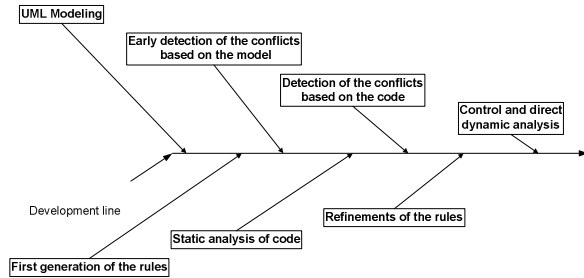


Figure 3. Approach and development process.

The modeling phase allows us managing, early in the development process, the potential conflicts generated by the interactions between aspects. The developed solution is based essentially on the introduced connection structures and relationships between aspects and classes as illustrated in detail in the following sections. The join points and pointcut structures are very important. They represent the basic constructions allowing the relationships between the different elements. The degree of separation of concerns depends largely on the types of available join points [2]. Our approach is based on these types. A complete description of join points and pointcuts is given in [9, 13].

#### 4.2 Aspects-Classes Connections Modeling

We use an extended UML class diagram model (figure 4). This extension allows the modeling of aspects and their connections to classes. Our main objective is not to support the modeling phase. There are several interesting works on this topic such as [1, 5]. We rather focused on supporting the detection of aspects conflicts as soon as possible by extracting a semantic synthesis from design models. We focused essentially on the basic and important elements (e.g. join point, pointcut, etc.) which are necessary to have early in the development process allowing the detection of connected aspects (related to the same classes). This modeling allows us managing, early in the development process, the potential conflicts generated by the aspects having common join points. Later in the process, a static analysis of the code will make it possible to refine the information collected initially from models, especially regarding the nature of the used methods (modification or access methods) and a possible execution order of aspects. Model analysis is a part of a global Framework composed of several tools. The framework is flexible and extensible. It is possible to integrate other models if necessary.

For the modeling phase, we were inspired by several approaches proposed in the field in the field of aspect modeling. In general, we can note that all authors introduce the notion of stereotype for the representation of aspects, point cuts, advices and introductions. To keep simple the representation, we summarize in our model these concepts. Moreover, we extended these concepts by introducing the concept of connection table, which is attached to each relationship between an aspect and a class. This table contains the principal information that we need to perform our approach.

Figure 4 gives an example of UML modeling integrating two aspects and two classes. This figure illustrates the fact that we

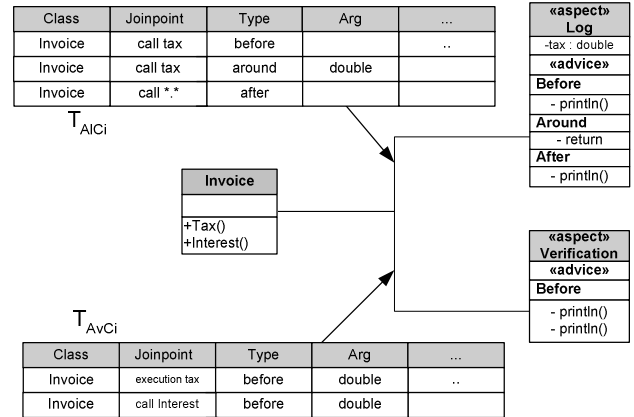


Figure 4. Relationships between aspects and classes.

privilege a relationship "aspect towards class" and not the opposite. Thus, aspects are connected to classes using unidirectional links. In addition, in order to facilitate modeling and to establish the bases of our technique for the detection of conflicts, we define the type of the connection while creating the link. Aspects are compartmentalized. The various compartments of an aspect are: aspect with attributes and the stereotype advice with methods. The adopted modeling takes into account the multiple links, which can exist between aspects and classes. Thus, an aspect  $A_i$  may be connected to several classes  $C_1, \dots, C_n$  simultaneously, which corresponds to the definition and the role of the aspects [9]. However, several aspects can be connected to the same classes (one or more). To connect aspects to classes, we create a connection link. This link represents the relationship  $R_{AC}$  that exists between the aspect  $A$  and the class  $C$ . While creating this link, we define the set  $S(R_{AC})$  that represents, in fact, what we called the semantics of the relation  $R_{AC}$ . It acts primarily of a collection of information characterizing the relationship between the aspect and the class. This information represents basic elements of the relation. These elements will be described in detail in section 4.3. They are gathered in several tables  $T_{AC}$  which we called connection tables. Each table  $T_{AC}$  characterizes the relationship that exists between an aspect  $A$  and a class  $C$ . For example, for the connection link of the aspect *verification* with the class *invoice* on the method *tax* can be described as illustrated by figure 5.

The connection tables support two levels of visualization. The first level relates to the representation aspect-classes ( $T_{A^*}$ ) as the example above. This level makes it possible visualizing all connections, which an aspect has with the classes of the model. The second level relates to the representation class-aspects ( $T_{*C}$ ). This level makes it possible visualizing all connections of a given class with all associated aspects. Thus, while creating the different links, we can define a precedence of the aspects during the design phase. For each aspect present in the model, all the information related to the relationships which it maintains with the classes of the model is presented at the two levels: Individual (Aspect-Class) and Collective (Aspect-Classes). As illustrated by figure 4, the  $T_{A1Ci}$  table represents the relationship  $R$  (Log: Invoice) and the tables  $T_{AvCi}$  represents the relationships  $R$  (Verification: Invoice). The content of the tables represents basic elements on which our approach is based. Each record of the table is associated to a pointcut synthesizing all information on a line. The columns of

Aspect	Class	Join point	Arg-ument	Type of join point	Advice	Pointcut	Type of pointcut	Precedence	Operation type (access or modification)
Verification	Invoice	Tax	Double	Method execution	Before	-	Execution of method	-	access

**Figure 5. Description of a connection link.**

the table represent the various elements of connection. These elements inform us, in a precise way, about the mode of connection of aspects to classes and their execution.

The analysis of models makes it possible to better understand the possible interactions between aspects and classes, and the conflicts that can result from this. This makes it possible apprehending the complexity of connections and to quickly perceive the competition caused by aspects pointing, for example, on common join points. At this stage, the adopted approach does not require the knowledge of the code. It's based on very simple mechanisms, which help the developer, early in the process, to make decisions regarding the resolution of the detected conflicts.

### 4.3 Relationships Between Aspects and Classes: Basic Elements

Table 1 gives the list of the basic elements that we consider. They represent a relevant set of information necessary to the management of conflicts. The idea consists in joining them together, for each relation between an aspect A and a class C, in an integrated way. To each relation  $R_{AC}$  between an aspect A and a class C corresponds a set of records  $S(R_{AC}) = \{(e_1, \dots, e_n), \dots\}$  representing the semantics of the relationship. Each  $e_i$  corresponds to one of the selected elements. The conflicts detection process is founded on the contents of these tables. Some information is not used for the detection of conflicts but is given to help the developer.

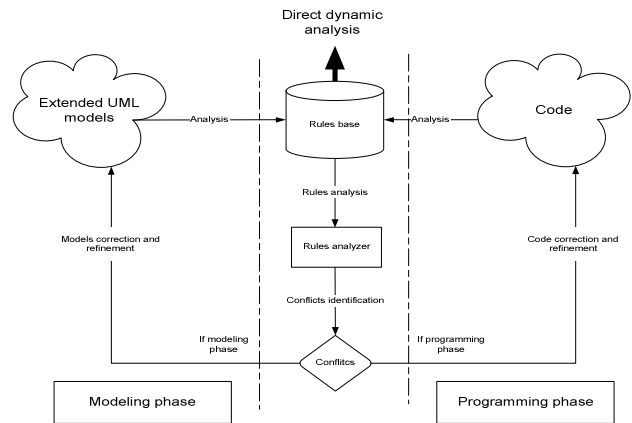
Name	Description
Aspect	(A <sub>i</sub> ) Aspect implied in the relationship
Class	(C <sub>j</sub> ) Class including the join point
Join point	(J <sub>i</sub> ) Join point implied, method on which the aspect is grafted, this allows managing the number of connections on a join point
Argument	(Arg <sub>i</sub> ) Arguments of the method, the type of argument is important for the overloaded methods
Type of join point	(TJ <sub>i</sub> ) Additional information allowing to qualify Join point, given on a purely informative basis
Advice	- The advices before, around and after and all the alternatives, this allows to know if two aspects enter in conflict on the same point
Pointcut	(P <sub>i</sub> ) Name of pointcut, indicates if it's a anonymous pointcut
Type of pointcut	(TP <sub>i</sub> ) Given on a purely informative basis
Precedence (order)	(O <sub>i</sub> ) Precedence order of the aspects while sharing the same join point

**Table 1. Basic elements.**

### 4.4 Inference Process

The information contained in the tables is collected during the development process, from the modeling phase to the implementation phase. This information is synthesized primarily for two principal actors: the developer and the algorithms supporting the detection of conflicts. In this way, the developer can make decisions and possibly re-examine the models. The adopted approach follows an iterative process. Its various stages are illustrated in figure 6. The process starts with models analysis

and continues until the code. The models are analyzed in order to extract a synthesis from the semantics of the relationships between aspects and classes. This synthesis is translated into formal rules. The analysis of the rules base enables us detecting the conflicts between aspects. This makes it possible to reconsider the models and to make the necessary corrections.



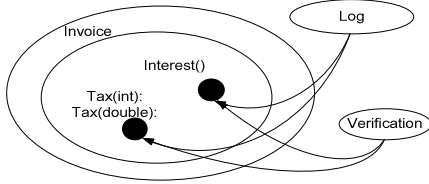
**Figure 6. Inference process.**

During the implementation phase, the static analysis of the code enables us refining the rules base. This refinement makes it possible to reinforce the inference process for the detection of conflicts present in the code. The various information collected in the rules may also be used to drive a dynamic analysis. A particular effort will be laid on certain parts of the code likely to be at the origin of problems generated by conflicts. In this way, the dynamic analysis would be complementary to the static analysis.

### 4.5 Formal Framework

The adopted approach is based on a formal description of the relationships that exist between aspects and classes. Formal description of aspects is a recent subject and slightly covered as mentioned in [4]. Conceptually, the structure of the relationship  $R_{AC}$  represents the join point of an aspect with a class. Aspects use join points to specify the place of the connection and pointcut to define how this connection is carried out. This implies that the join point is necessarily an element of the class to which the aspect is connected. That can be represented formally in the form:  $J_i \subseteq C_j$ , a join point  $i$  is included in the class  $j$ . For the representation of an aspect, we consider the total property according to which the aspects break the encapsulation and connect themselves to one or more classes. Aspects can be regarded as sets including one or more classes. The pointcuts specify the type of connection. Let us consider an aspect A connected to a method  $m$  defined in a class C. We represent this connection by: Connection (A, J, C) = ((Aspect (A)  $\cup$  Join point (J))  $\subseteq$  Class (C)). Figure 7 illustrates this connection. The

intersection of join point of an aspect creates a set, which is included in the class. For example, for a pointcut within, this is translated by the connection  $(A, \forall J, C) = ((\text{Aspect}(A) \cup \text{All Join point}(\forall J)) \subseteq \text{Class}(C))$ .



**Figure 7. Illustration of the connection.**

The adopted formal notation is at the bases of the rules construction process. We take into account all the basic elements described previously. This formalism makes it possible to determine the common elements shared by the aspects of a program. The larger the set of shared elements is, the more there are risks of conflict. We represent the connection of an aspect  $A_i$  with a join point  $J_k$  using the operator of intersection ( $\cap$ ). We obtain:  $(A_i \cap J_k) = L_{ik}$ . Let us now consider two aspects  $A_i$  and  $A_j$  that are related respectively to two join point  $J_k$  and  $J_m$  of the same class  $C_i$ . This situation is represented by the following expressions: Connection  $(A_i, J_k, C_i) = ((A_i \cap J_k) \subseteq C_i)$  and Connection  $(A_j, J_m, C_i) = ((A_j \cap J_m) \subseteq C_i)$ . Starting from these two connections, we can illustrate the various following cases:

- $(i \neq j)$  and  $(k = m)$  : there is a conflict between  $A_i$  and  $A_j$  because they are in relation with the same join point.
- $(i \neq j)$  and  $(k \neq m)$  : there is no direct conflict between  $A_i$  and  $A_j$ , they form part of the same class  $C_i$ .
- $(i = j)$  and  $(k \neq m)$  : only one aspect with two distinct join points.
- $(i = j)$  and  $(k = m)$  : only one aspect, which is in connection with one join point.

## 4.6 Case Study

We consider, in the following, a simple but concrete case study to illustrate our method. It will enable us to illustrate several types of conflicts (specification, aspect-aspect and concerns). The example contains an *Invoice* class that takes an amount in input, calculates the value of the tax and, if the transaction is with credit, adds the interest. The *Invoice* class defines two principal methods:

- *Tax()*: overloaded method that adds a tax to an amount.
- *Interest(double)*: increase of the amount if the case of credit.

This class is accompanied by two aspects:

- *Verification*: validate the input data
- *Log*: print messages, records in a file and manages the applicable tax.

Figure 7 illustrates this case in a graphic form. At first sight, we can note that possible conflicts can emerge because the two aspects share the same join points. In particular, the execution of the *Tax()* method can pose a problem. The advice *Around* can also cause a problem. Indeed, it can capture the context with the method *proceed()*. If the latter is not used, the execution will be different. UML modeling gives the model illustrated by figure 4. The connections tables, in this example, are simplified. Some elements are omitted deliberately. The UML model captures the various relationships that exist between the aspects and the

classes. The semantics of these relationships is described by several information joined together in the tables attached to the relationships. The analysis of the contents of these tables enables us generating the formal rules describing the relationships. Their evaluation, by the rules analyzer, allows detecting all the potential conflicts. It is clear that at this stage of the process, the effectiveness of the method depends closely on the information provided in the models. For simplification, we represent the aspects *Verification* and *Log* by  $A_V$  and  $A_L$ , the class *Invoice* is represented by  $C_i$  and join points *Tax* and *Interest* by  $J_t$  and  $J_i$ . The formal representation extracted from the UML model of figure 6 is given by the following expressions:

- **Connection 1**  $(A_V, J_t, C_i) = \text{Before}((\text{Aspect } \text{Verification} \cup \text{Execution } \text{Tax}(\text{double})) \subseteq \text{Invoice})$
- **Connection 2**  $(A_V, J_i, C_i) = \text{Before}((\text{Aspect } \text{Verification} \cup \text{Call } \text{Interest}(\text{double})) \subseteq \text{Invoice})$
- **Connection 3**  $(A_L, J_t, C_i) = \text{Before}((\text{Aspect } \text{Log} \cup \text{Call } \text{Tax}(\dots)) \subseteq \text{Invoice})$
- **Connection 4**  $(A_L, J_i, C_i) = \text{Around}((\text{Aspect } \text{Log} \cup \text{Call } \text{Tax}(\text{double})) \subseteq \text{Invoice})$
- **Connection 5**  $(A_L, J_*, C_*) = \text{After}((\text{Aspect } \text{Log} \cup \text{Call } *()) \subseteq *)$

## 4.7 Case Study Discussion

We can notice that connection 1 can enter in conflict with connection 3. They share the same join point, even if they are connected differently to the same class. It's a conflict related to the ordering of aspects. This ordering depends on the dynamic context. If there is a call of *Tax* with a data of a type other than *Double*, one of the aspects will not be executed. The connection 5 presents a case of conflicts related to transverse specifications. The use of the pointcut *\**, *\** brings an overlapping of the join points in many classes and the use of one function *println* brings an accidental recursion. A subtle case relates to connection 4. We know that depending on the executed code, this may affect the execution. The implementation of the *Around* method is very important. An around method with *proceed* has a behavior different than an around method without *proceed*. This case can be detected by code analysis and a dynamic execution will be necessary to capture all the context of this case.

The conflicts generated by the introduction of aspects are often implicit and difficult to capture especially when that influences the program control flow. Our approach is rather static. It enables us to extract a rich collection of information from models making it possible to detect a large number of conflicts. In future, we plan to use this information to drive a dynamic analysis that would be complementary to our approach. It's interesting to know, for example, how an aspect is linked to a method (call or execution). This process is very complex and requires a good management of scheduling in the execution. The proposed technique covers all the join points supported by AspectJ. Our technique allows to include *"\*"* in the relationships. This facilitates the detection of transverse specifications conflicts. The analysis of the pointcuts enables us to detect certain conflicts related to accidental recursion. This type of conflicts occurs when an aspect, attached to a join point, executes its method that is included in the pointcut of another aspect-class relationship. This represents a conditional execution that can lead to an inconsistency.

	Douence [4]	AJDT [4]	Trace analysis [11]	Interference analysis [12]	Our method
<b>Analysis</b>	Static	NA	Dynamic	Static	Static and dynamic opening
<b>Modeling</b>	NA	NA	NA	Class diagram	Class diagram and extended UML
<b>Code analysis</b>	NA	NA	Code execution	Extends and implements	Pointcut and join point
<b>Formal method</b>	Yes	No	No	Yes	Yes
<b>Supported Conflicts</b>	- Aspect-Aspect Interactions - Visibility	None but tool support	- Superim-position - Amputation - Change of behaviour	- Interference - Inheritance	- Aspect-Aspect conflicts - Programming help
<b>Simplicity</b>	Heavy and very theoretical method	Yes	Simple method but heavy to carry out	Yes	Yes
<b>Visual aspect</b>	No	Yes	No	Yes	Yes
<b>Tool Support</b>	No	Yes, pointcut et join point listing	No	No	Yes, information table
<b>Detection and/or resolution</b>	Detection and resolution	NA	Detection	Detection	- Early detection and resolution - Ordering
<b>Future research</b>	- Linguistic extension - Modeling of the code - Instantiation in AspectJ	- Pointcut wizard - Pointcut reader - Composition filter	- Use static methods	- Composition filter	- Trace analysis integration - Refinement of the tools and the method - Implementation

**Table 2. Comparison of methods.**

## 5. DISCUSSION

The detection of conflicts between aspects is a relatively recent subject and is poorly covered. This section presents a brief comparative evaluation between our method and the rare approaches proposed in the literature in this field. Douence et al. have elaborated a formal representation of the problem [4], Clement et al. proposed the development tool AJDT [3], and Storzer et al. proposed an approach based on execution trace analysis in [11] and interference analysis in [12]. Our approach is relatively earlier. It supports the prediction of conflicts based on model analysis. The comparison that we present in this section tries to highlight the characteristics of these approaches, their strong points and their weak points. A synthesis of the comparison is given in Table 2. Douence et al. propose in [4] an algebraic approach. It presents, in the manner of approaching the conflicts, several similarities with the approach that we adopted. The code-based technique suggested by these authors is based on the semantic of the aspect and a mathematical representation of the code. We limit ourselves to a simpler semantics to represent the same type of situations. Moreover, the solution that we propose is relatively earlier. It starts with the analysis of the design models and continues until the implementation phase. The suggested method remains very close to what is already supported by the object-oriented development environments. The concepts that we introduce are not completely unknown. They represent basically adaptations of largely tested concepts. During the modeling phase, aspects are treated like data bases tables having N-N relationships.

Connections tables are identical to those that we will find for an N-N relationship between two tables in the entity-relationship model. We believe that, using  $T_{AC}$  to generate the rules representing the relationships semantics, we offer a simpler and more visual method based on concepts that are easy to assimilate and to integrate in the current development tools.

## 6. CONCLUSIONS AND FUTURE WORK

We presented a new method for the detection of semantic conflicts between aspects. The proposed technique is simple to implement and could be integrated in existing environments. It offers the advantage of supporting an early detection of conflicts based on model analysis. Model analysis allows translating relationships between aspect and classes into formal rules. The evaluation of these rules, made by a rules analyzer, allows detecting eventual conflicts between aspects. The first simulation of the approach gives interesting results. The supported process is iterative. Presently, we are working on the implementation and the experimentation of the first stages of our approach, primarily, those related to model analysis. We plan to introduce the indirect conflict analysis.

## 7. ACKNOWLEDGMENTS

This work was supported by NSERC (Natural Sciences and Engineering Research Council of Canada) grant.

## 8. REFERENCES

- [1] Aldawud, O., Elrad, T. and Bader, A. *UML profile for aspect-oriented software development*, AOSD 2003, 3rd Aspect-Oriented Modeling Workshop, Boston, MA,
- [2] Chiba S. *What are the best join points?*, OOPSLA 2001
- [3] Clement A., Colyer A. & Kersten M. *Aspect-Oriented Programming with AJDT*, AAOS 2003, Darmstadt, Germany.
- [4] Douence R., Fradet P. & Südholt M., *Detection and resolution of aspect interactions*, INRIA, technical report, no. RR-4435, April 2002
- [5] Fayad, M.E. and Rangahat, A. *Modeling aspects using software stability and UML*, UML 2003, 4th Aspect-Oriented Modeling Workshop, San Francisco, CA.
- [6] Hanneman J., Chitchyan R. & Rashid A. *Analysis of aspect-oriented software*, AAOS 2003, Darmstadt, Germany
- [7] Kiczales G. (2003). The Server Side, *Interview with Gregor Kiczales*, [www.theserverside.com](http://www.theserverside.com)
- [8] Kiczales G., Lamping J., Mendhekar A. Maeda C., Lopes C. V., Loingtier J. & Irwin J., *Aspect-oriented programming*, ECOOP 97.
- [9] Laddad R. *I want my AOP!*, Java World, January 2002, [www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html](http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html)
- [10] Orr J. *Java tools reign supreme : Most Innovative Java Product or Technology: AspectJ 1.0.6*, Eclipse.org, June 9 2003 [http://www.javaworld.com/javaworld/jw-06-2003/jw-0609-eca\\_p.html](http://www.javaworld.com/javaworld/jw-06-2003/jw-0609-eca_p.html)
- [11] Störzer M., Krinke J. & Breu S. *Trace Analysis for Aspect Application*, AAOS 2003, Darmstadt, Germany.
- [12] Störzer M. & Krinke J. *Interference analysis for AspectJ*, AOSD 2003.
- [13] Xerox AspectJ Team *AspectJ programming guide*, [www.eclipse.org/aspectj/](http://www.eclipse.org/aspectj/)