Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

On using pre and postconditions to tackle the aspect scheduling problem by rewriting systems: a design-level approach

Toufik Benouhiba[a] Amina Boudjedir[a]

a. LISCO Laboratory, Computer Science Department, Badji Mokhtar University, Annaba, Algeria

Abstract The aspect-oriented paradigm promises separation of crosscutting concerns by modularizing them as aspects. This paradigm allows then weaving aspects upon some points in the base system. Unfortunately, the interaction of aspects may have an undesirable effect on each other and/or on the base system if they are executed in any order. Many works tried to solve this problem but the proposed solutions were either limited to some simple models of aspects or used to check if a set of temporal properties is preserved when aspects are introduced. In this paper, we propose a framework based on extended UML diagrams (class and state/transition diagrams) in order to make the detection of undesirable interaction more powerful and realistic. This framework relies on finite state automata (FSA); it transforms the interaction problem into a reachability issue. In fact, bad interaction is expressed as a generic LTL property which is independent of the system to be verified. This property can be checked using any model-checker like Maude. To concretize the proposed framework, we propose a rewriting system that allows an implicit construction of the FSA of the base system and the aspects in addition to the aspects composition and the weaving mechanism. Therefore, the proposed system defines a translation scheme of UML models into rewriting logic specifications written in Maude language. Thanks to the advances of the *on-the-fly* technique in Maude, the use of its LTL model-checker avoids a systematic exploration of all possible combinations of the aspects. The approach will be illustrated by a case study in order to explain how it works.

Keywords Aspect interaction; finite state automata; pre/postconditions; scheduling; UML class and state/transition diagrams; LTL; rewriting systems.

1 Introduction

The object-oriented paradigm has been proposed for building *easy-to-maintain* understandable programs. However, this paradigm is particularly limited in the expression of crosscutting features, i.e., those expressing concerns that affects several classes or modules of the application. Indeed, in at least two cases (the tangling and scattering code [SP05]) the object-oriented paradigm does not provide a satisfactory solution to achieve clear and well-structured programs.

To overcome these problems, the advanced separation of concern approaches proposes separating concerns and core functionalities of a system in distinctive structures. Among these approaches, the aspect-oriented paradigm (AO) [FECA04] allows engineers to design and implement software's core functionalities and crosscutting concerns, as a *base system* and separate modules, called *aspects*, respectively. These aspects include *pointcut* declarations which provide logical definitions for selecting the *join points* where a piece of *advice* is applied Before execution, AO' *weaving* mechanism integrates those aspects into the base system on the selected join points. This new paradigm helps to create a more coherent system since it avoids scattering and tangling the code of concerns everywhere in a system.

In spite of the improvement in modularity, reusability and maintainability of the system, AO paradigm faces an important problem concerning the difficulty to reason about the behavior of the whole system. Even when the base system and the aspects are well defined, the weaving mechanism may lead to a bad interaction between all parts. This is one kind of what is commonly known as **aspect interaction problem** [SP05, TCZY09].

Let us note in passing that since aspects may modify any variable's value of the base system, the semantic of this latter can be drastically changed. Because of the declarative nature of the aspects and the relationship between pointcuts and advices, AO paradigm allows aspects to be woven upon the base system on many join points. Hence aspects may interact with other aspects in the case of weaving multiple aspects on the same join point of the system. Their interaction may have some undesirable effects on each other and/or on the base system if they are executed in any order.

Many works have attempted to detect conflicts between aspects but the proposed solutions were either limited to some simple aspect models or used to check whether a set of temporal properties is preserved when aspects are introduced. Furthermore, other approaches have tried to detect conflicts between aspects using the *source-code*. These approaches cannot tackle powerfully the interaction problem because of the complexity of the *source-code*. Hence, we argue that aspect interaction problem has to be detected and fixed in early development stages in order to minimize maintenance costs. Thus, we focus, in this article, on the detection of bad interactions between aspects in general and aspect scheduling in particular at the model level. Models are more abstract than code; therefore fixing errors in the design is cheaper then fixing errors in the code.

In this paper, we present a rigorous framework for aspect-oriented modeling and verification based on finite state automata (FSA) [Hop79]. In this framework, we use a combination of the Aspect-UML class diagram [MV07] and High-Level Aspects (HILA) [Zha10] UML states machines. First, classes and aspects are defined in the Aspect-UML in order to be able to specify, among others, pre and postconditions of the pertinent elements such as: methods, join points and advices. The behavior of these elements is then specified in state/transition diagrams. Then, an FSA representing the woven system is built. We will show that the scheduling problem can then be transformed into a verification of a generic LTL [MP92] property.

In order to implement the proposed approach, we use the rewriting systems which are interesting to model and verify aspect interaction problem. This is because rewriting systems turned out to be very powerful in modeling and verifying the complex systems [DJ90]. In fact, they allow an implicit construction of the FSA of the base system and the aspects as well as aspect composition and weaving. In addition, they facilitate the expression of the pre and postconditions by using rewrite rules. Therefore, we propose a rewriting system to model the base system as well as the aspects. This rewriting system is built upon the Maude tool [CDE$^+$11] where each UML diagram is transformed into Maude constructions. The Maude LTL model-checker is then used in order to detect possible conceptual errors concerning aspect interactions. Maude has the advantage from some of the advances in *on-the-fly* [EMS03] technique which lead to a non-systematic exploration of all possible combinations of the verified aspects.

In the remainder of this article, we present in Section 2 the aspect-oriented paradigm and the problems arising from the interaction of aspects. In section 3, we give the different extended UML models used in our approach. We present in section 4 an overview of the proposed approach. We outline, in section 5, the main phases of the framework and argue that the interaction problem can be transformed into a reachability issue. In order to concretize the proposed framework, we present in section 6 how UML models are translated into Maude specifications. We illustrate in, section 7, how LTL model-checker can be used to detect bad (or a correct) aspect interactions in the case study. The results of interaction are discussed in this section. Section 8, summarizes some related works. The conclusion of the paper and some of its perspectives are given in section 9.

2 AO paradigm and the aspect interaction problem

AO paradigm is a methodology for separating transversal preoccupations or concerns that crosscut the core functionalities of the base system by modularizing them into reusable modules. The final system is built by *inserting* these modules within the base system. The basic concept in AO paradigm is the aspect which is for AO paradigm what a class is for object oriented programming: the modularization unit. During the design or programming, any concern that crosscut several classes is considered as an *aspect*; this one consists of two parts: *pointcut* and *advice*. A *pointcut* is a set of many *join points* that indicate some action or a state of the system where one or several aspects apply. An *advice* is the behavior of an aspect, it can be executed *before*, *after* or *around* the join point that has been selected by a pointcut. The *around* advice contains a *proceed* instruction to execute the invoked join point. The concepts introduced by this paradigm allow core functionalities (i.e., base system) and crosscutting concerns (i.e., aspects) to be specified independently in separate modules.

The above mentioned concepts require that when running the base system, aspects must be able to control the execution (intercept it, stop it, restart it, etc.) by interleaving the execution of their advices. To achieve this, two approaches exist. The first consists of statically re-mixing the aspects and business functionalities (i.e. during or after the compilation of the code) with an operation called *weaving*. This latter consists mainly in converting aspects into classes, advices into methods and inserting calls to these methods in business functionalities where indicated by the pointcuts. So the code that will finally run is merely an object-oriented code. The second approach achieves a dynamic weaving (i.e. during the execution). This requires a platform that is capable of stopping the execution business functionalities to give control to aspects timely.

Despite of the improvement in modularity, reusability and maintainability of the system, AO paradigm faces an important problem: it is difficult to reason about the behaviors of the whole system obtained by weaving aspects onto to base system. Since the base system is modified by weaving, unexpected results can emerge. In AO paradigm, this issue is commonly known as the aspect interaction problem [SP05, TCZY09].

In fact, aspect interaction problem is derived from the interaction between aspect and base system, which name is *Aspect/Base-system* interference, and the interaction between the aspects which name is *Aspect/Aspect* interference. The former interaction arises from the declarative nature of the concepts of the AO paradigm. Some aspects can be woven into the base system at many join point. As a result, aspects may modify any variable's value and change arbitrarily the control flow of the base system. Thus, the semantic of this system can be drastically changed. The second interaction, which is *Aspect/Aspect* interference, occurs when multiple aspects are woven at the same join point of the system. Their interaction may have some undesirable effect on each other and/or on the base system if they are executed in any order. Some of these orders can be used in a harmful way that invalidates the desired properties of the system. Actually, there are many kinds of Aspect/Aspect interference [SP05]:

- Dependence: each time an aspect is executed, another one should be also executed.
- Redundancy: two aspects or more have the same effect.
- Scheduling: many independent aspects are concerned with the same joint point but executing them in any order may violate the properties of aspects or the properties of the base system.

The two former problems should be fixed in design level because the weaver cannot find out the relationship between aspects and hence cannot determine whether a given aspect should be executed or not. These problems should hence be resolved by the aspect developer [SP05].

The scheduling problem, which is the subject of this paper, depends on both design and implementation levels (the weaver's rules). In fact, this problem may occur when many aspects concern the same joint point. In such situation, the weaver has to decide how to execute them. Parallel execution is possible but can lead to race condition that makes the analysis of the system harder (especially if aspects use shared variables). AspectJ weaver [KHH+01] which is one of the most popular frameworks, usually executes them sequentially which means that a problem will arise: should advices be executed in any order? And if not, what orders should be avoided? For instance, logging and authentication aspects are an obvious example of this problem. The combination of these aspects illustrates the scheduling problem. It is obvious that if the former aspect is executed before the last one, the authentication event will not be recorded.

An interesting solution of the scheduling problem may involve the use of Aspect-UML [Mos08]; a UML profile that allows modeling interactions between the base system and aspects by extending the classic UML use case and class diagrams. The new use case diagram allows integrating aspects as extensions of the base system use cases. Meanwhile the new class diagram consists of adding aspects as a kind of new classes and using OCL constraints to annotate both methods and advices (*before* and *after* types). The annotations consist of the pre and postconditions of every action. The scheduling problem can be solved by defining a precedence relationship (as it is the case in AspectJ) between the conflicting aspects (aspects that share the same join point) and representing it on Aspect-UML diagrams. This relationship is used to define the sequential composition of the conflicting aspects. However, if it is not always obvious to find the precedence relationship (especially in case of a big number of aspects or if aspect interaction cannot be avoided as for aspects that modify variable's values of the base system) then the result can be disastrous.

At this level, the use of formal approaches can provide a good solution to this problem by modeling base system/aspects interactions with Aspect-UML class diagrams. In this paper, we argue that the use of pre and postconditions in Aspect-UML class diagram is useful for detecting bad aspect interaction. Thanks to these ones (considered in our work as assumptions and not guards that have to be verified in order to execute a given task), Aspect-UML models provide additional information to analyze aspect interaction from a semantic point of view. The aim behind the use of the pre and postconditions is to get some independence between the aspect interaction problem and the properties to be verified (and preserved). Thus, the aspect interaction problem can be defined or transformed into the verification of a generic property. The definition of a set of global temporal properties that should to be preserved is harder than defining simple conditions. In addition, it is not always obvious to define and reason on global temporal properties since this requires reasoning about the whole system. Pre and postconditions have the merit to operate on a local level which makes them easier to define.

However, the use of Aspect-UML class diagram annotated with the pre and postconditions is still insufficient to tackle the scheduling problem because we lack information about the behavior of the system. It is hence judicious to combine this diagram with state/transition diagrams as proposed by [Zha10] so that richer execution scenarios could be considered.

In this work, we propose a rigorous framework for aspect-oriented modeling and verification based on pre and postconditions and automata. We consider here bad interaction between aspects and the base system as well as the scheduling problem. In the proposed framework, classes and aspects are first defined in the Aspect-UML in order to be able to define, among others, pre and postconditions of methods and advices. The behavior is then specified in state/transition diagrams so that a finite state automaton representing the woven system is built. We will show that aspect interaction problem can then be transformed into a verification of a generic LTL property which is independent of the system to be verified and hence enables the use of any model-checker like Maude to verify bad interaction. We also propose a rewriting system built with respect to the Maude tool [CDE$^+$11] to model the whole system. The use of rewriting systems seems to be a good idea to model the system since they are suitable for the representation of the pre and postconditions of the methods and advices as rewrite rules. Moreover, the use of rewriting system is also motivated by the fact that interaction problem is equivalent to check the confluence of the aspects execution orders.

3 Modeling aspect-oriented systems

To model aspect-oriented systems at an early stage of the development, we have used the Aspect-UML profile [Mos08] which introduces the fundamental concepts of AO systems and permits to define methods' pre and postconditions as OCL constraints. However, this profile cannot define the dynamic behavior of the base system and the aspects. In this case, the use of UML state/transition diagrams [OMG09] can be of great help through modeling the behavior of the different objects and the advices instead of just declaring them as OCL constraints. Hence, we have used High-Level Aspect (HILA) [Zha10], an aspect-oriented extension of UML state machines, which makes it possible to define rigorously the behavior of advices.

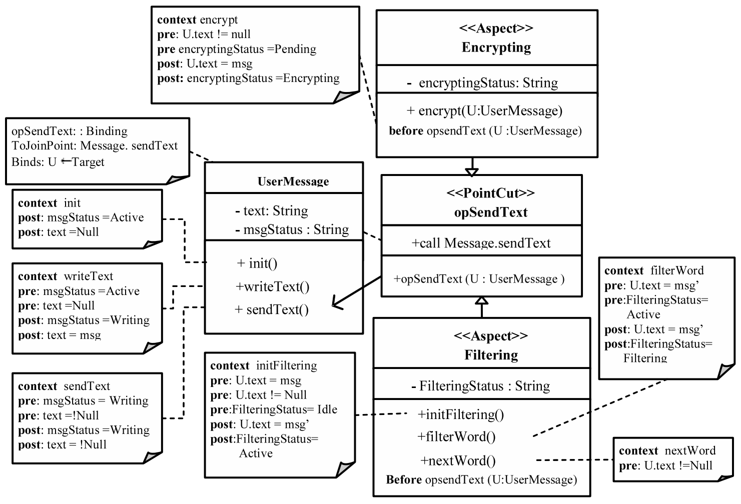


Figure 1 – A part of the class diagram of secure messaging system

3.1 Class diagram

The proposed framework will be validated through a case inspired from [ARS09] that helps us to explain the different key points of our contribution. This example describes a simulation of a secure messaging system in which the users can write, send and receive messages. To these base functionalities, the system administrator decides to add two aspects in order to improve the management quality. These aspects are described below:

- The *encrypting* aspect encodes the written message in order to secure its transmission in the system.
- The *filtering* aspect removes inappropriate words of the message in order to alleviate the transmission of data (message).

Figure 1 shows the integration of the *encrypting* and *filtering* features in the UML class diagram, using Aspect-UML notation. These crosscutting are depicted as UML classifiers decorated with stereotype « *Aspect*». These aspects crosscut the base system through the pointcut *opSendText()* which is modeled as a special interface stereotyped with «*PointCut*». Both the *encrypting* and *filtering* aspects implement the *opSendText()* pointcut and hence provide an implementation to the *before* advices. These two aspects modify the same variable. Encrypting the text after filtering inappropriate words is the obviously desired behavior. In reverse order, the *filtering* aspect will be applied to an encrypted text and will not be able to filter this text since the message is modified. This is a typical situation of aspects conflict because two operations will be added before the join point and executed in a given order.

3.2 State/transition diagrams

UML state/transition diagrams describe the internal behavior (state changes) of objects. It specifies the possible sequences of states and actions that an object can handle during its life in response to events. Different types of events are defined by UML (signals, method call). We will only focus on the events of the call type. Aspect behavior is also modeled by state/transition diagrams as in HILA. Each advice is represented by its own state/transition diagram in order to improve modularity. However, since pointcuts definitions are given in the extended class diagram, they do not figure in state/transition diagram. In addition, we suppose that there is neither parallel region nor synchronization in order to avoid resumption conflict [ZH12]. By synchronization, we mean barriers which are used by many processes to implement a kind of rendez-vous. This can be seen as a future extension of our work. It is noteworthy that the user is not meant to model either weaving or aspect composition in state/transition diagrams since these activities are generic and should be model-independent. Figure 2 shows the state diagram of the *UserMessage* class and the *filtering* aspect.

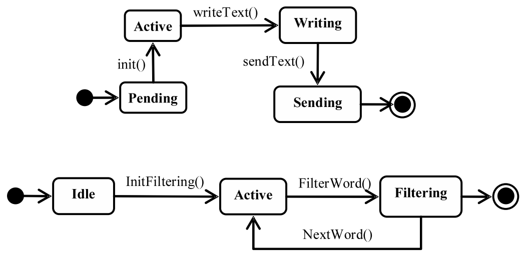


Figure 2 – State/transition diagrams of the UserMessage (on the top) and Filtering aspect (at the bottom)

Despite the fact that UML is a language endowed with widely used notation for modeling complex systems, this language is still semi-formal. UML diagrams need to be more formalized in order to ensure a correct behavior of a system. Hence, UML has known considerable efforts to formalize its diagrams, among which the formalization of the state-transition diagrams [CD05] can be cited. These diagrams were the most used by the fact of their large application in the modeling of the internal behavior of objects. In this work, we propose to model state/transition diagrams as finite state automata (FSA) in order to formalize state/transition diagrams and provide a formal model that describes advices composition and weaving. This formal representation helps us later to verify the aspect interactions and advices scheduling problem.

4 An Overview of the proposed approach

In this section, we explain the different activities involved in using our approach. The first step of this approach (marked '1' the figure 3) represents the modeling of the UML diagrams. The user models the Aspect-UML class diagrams: he defines each classes and aspects and specifies the pre and postconditions of each method and advice. Once Aspect-UML class diagram is modeled, the user proceeds to specify the behavior of objects by using HILA-UML states-transition diagrams.

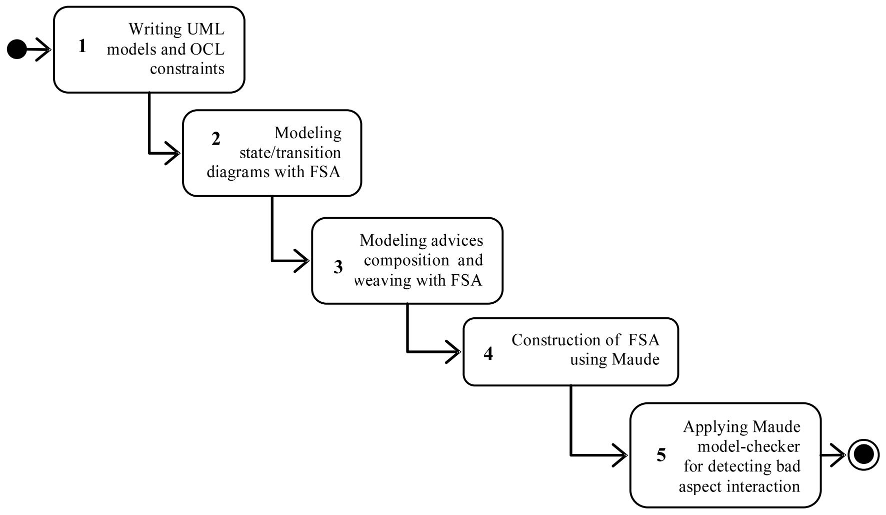

1 Writing UML models and OCL constraints
2 Modeling state/transition diagrams with FSA
3 Modeling advices composition and weaving with FSA
4 Construction of FSA using Maude
5 Applying Maude model-checker for detecting bad aspect interaction

Figure 3 – An Overview of the proposed approach

The next step, marked '2' the figure 3, allows representing each state-transition diagram with FSA. The aim of this representation is to define a framework in which every entity (both the base system and the different advices) is processed in the same manner as FSA. It is noteworthy that this work deals only with the state-transition diagrams of the *before* and *after* advices. The reason of this is that state/transition diagrams become simpler since *proceed* instruction of the *around* advice does not appear in a *before* or an *after* advice (state/transition diagrams will look like any ordinary object oriented system). In fact, the around advice requires storing their context when the *proceed* instruction is encountered (a stack is often necessary to store this context). The *before* and *after* advices don't need this context because there is no need to save anything after the execution of such advices. Their composition is then easier and the complexity of the verification is greatly relaxed.

Once state-transition diagrams are represented with FSA, we proceed to step 3 (marked '3' the figure 3) to represent the composition and weaving of advices with FSA. In the composition of advices of the same type, we merge the different FSA of each advice into one single FSA that will be woven on the base system depending on its nature. Note that we don't consider terminating aspects: those that can terminate the base system by executing a *halt* action. This is because the existence of two terminating aspects means that at least one of them may not be executed, thus we fail to reach the final state of the base system. Although this will not affect deeply our modeling, we suppose that the considered aspects are not terminating. Moreover, since defining pointcuts on aspects may lead to a non-terminating composition even for an observer aspect (see [DDF12] for more details), we consider only pointcuts that concern the base system. We will show in this step that aspect interaction problem can be transformed into a verification of a generic LTL property. All the details of the steps 2 and 3 are presented in section 5.

In Step 4 (marked '4' the figure 3), we propose a rewriting system that allows an implicit construction of the FSA of the base system, the aspects, the aspects composition and the weaving mechanism. This rewriting system is built with respect to the Maude tool [CDE+11] to model the whole system. Thanks to the concepts of the rewriting systems, it was possible to specify the pre and postconditions of the different methods and advices with rewrite rules. In addition, the use of rewriting system is also motivated by the fact that interaction problem is equivalent to check the confluence of aspects execution orders. Section 6 presents the proposed rewriting system in details.

Using the results from the previous step, the last step (marked '5' the diagram) allows us to apply Maude model-checker for detecting bad aspect interaction. This is discussed in section 7.

5 A formal framework for aspect-oriented modeling and verification

5.1 Modeling state/transition diagrams

A state/transition diagram is considered as an FSA defined by $(A,Q,init,F,\delta)$ such that A is set of actions, Q is the set of possible states, *init* is the initial state, F is the set of final states (those corresponding to the end of the execution of an advice or the destruction of an object) and δ: $A \times Q \rightarrow Q$ is the transition function (it describes how the state changes). In order to obtain a generic LTL property to verify, we require that each FSA contains one single final state. This can be easily done by transforming the state/transition diagram into a new FSA ($A^{'}$, $Q^{'}$, *init*,$\{f\}$, $\delta^{'}$) such that:

- $A^{'}= A \cup \{\tau\}$ where τ is a silent action (an action that has no effect). Each action may have a precondition denoted by *pre(a)* and postcondition denoted by *post(a)*. Whenever a pre or a postcondition is not defined, it is assumed to be equal to *true*.
- If F is a singleton then $Q^{'} = Q$, else $Q^{'} = Q \cup \{\text{f}\}$.
- If $\delta(q_i,\text{a})=q_j$ (such that q_i, $q_j \in Q$, $a \in A$) then $\delta^{'}(q_i,\text{a})=q_j$. In addition, if F is not a singleton, we add the following transitions (the goal is to have one final state): $\delta^{'}(q,\tau)= f$ (such that q $\in F$).

Example 1. We present here the transformation of the state/transition diagram of the *filtering* advice (figure 2) to FSA:$(A_{Fil}, Q_{Fil}, Idle, \{endf\}, \delta_{Fil})$ such that:

- $A_{Fil}= \{$*initFiltering(), filterWord(), nextWord()*, $\tau\}$
- $Q_{Fil}= \{$*Idle, Active, Filtering, endf*$\}$
- $\delta_{Fil} : Q_{Fil} \times A_{Fil} \rightarrow Q_{Fil}$ with:
  - δ_{Fil} (*Idle, initFiltering()*)= *Active*
  - δ_{Fil} (*Active, filterWord()*)= *Filtering*

- δ_{Fil} (*Filtering, nextWord()*)= *Active*
- δ_{Fil} (*Filtering*, τ)= *endf*

The *filtering* advice is represented as an FSA with a set of states Q_{Fil} and actions A_{Fil}. The transition function δ_{Fil} describes how these states change from the *Idle* state to final state *endf*.

5.2 Advice composition

Through the functioning of the rewriting systems, it is not compulsory to build all possible orders when many advices of the same type (*before* or *after*) apply to the same point. In fact, the verification will not check all advices order unless no order preserves the properties of the system. However, in order to clarify the composition of the advices, we consider in this modeling all execution orders, i.e. each time the weaver has to choose an advice to execute; the choice is made non-deterministically. That is why we call it non-deterministic composition. Throughout this, if the verification succeeds, one can be sure that any execution order is possible.

Let a and b be two advices of the same type that apply on the same joint point. Their non-deterministic composition is denoted by $a \sqcup b$. There are two possibilities to define this operation: the first one consists of keeping some composition information in the automaton of $a \sqcup b$ so that the operator will be associative, the second possibility consists of giving a n-ary definition for it since a binary definition without keeping any information on composition means that $\sqcup$ is not associative but commutative. The former solution is suitable for dynamic weaving (i.e. situations where we do not know in advance how many aspects to combine) but requires a more a complex definition. The latter solution is meanwhile suitable for static weaving and simpler to define. In this paper, we used the second definition since a safe design would not generally require a dynamic weaving.

Let's consider now n advices of the same type that apply on the same join point. Each advice is represented by an FSM as a_i=(A_i, P_i, Q_i, $init_i$, $\{f_i\}$, δ_i) such that i = 1..n. We first define the set Φ_n as the set of one-to-one mapping from {1..n} to {1..n} or, equivalently, all permutations of values from 1 to n (intuitively, an element of Φ_n represents an execution order of advices). We consider the set $\Phi_{i,1}$= $\{p \in \Phi_n \mid p(1)=i\}$ (intuitively, $\Phi_{i,1}$ corresponds to compositions in which the first element is the advice i). The non-deterministic composition $a_1 \sqcup a_2 ... \sqcup a_n$is given by ($A_{ND}$, Q_{ND}, int_{ND}, $\{f\}$, δ_{ND}):

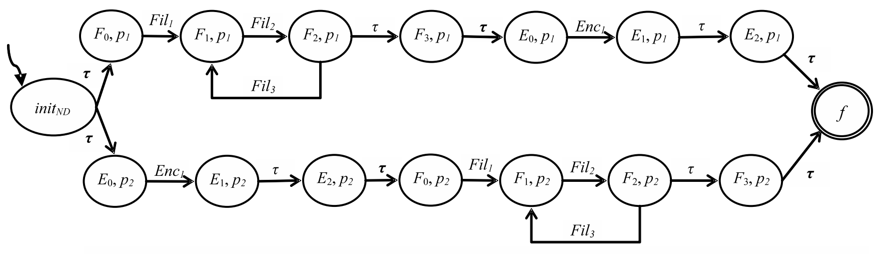
- $A_{ND} = \cup_{i=1}^{n} A_i \cup \{\tau\}$
- Q_{ND} =(($\cup_{i=1}^{n} Q_i$) $\times$ Φ_n) $\cup$ $\{int_{ND}, f\}$
- int_{ND} is the initial sate of the non-deterministic composition
- δ_{ND} is the new transition function defined by:

```
- for i = 1..n
    - $\delta_{ND}(int_{ND}, \tau)$= $\delta_{ND}(int_i, p_j)$ for each $p_j \in \Phi_{i,1}$(non deterministically
      pick one execution order in which advice $i$ is the first one to execute) —1
    - for each $q \in Q_i$
        - if $q \in Q_i - \{f_i\}$ then
            • $\delta_{ND}((q,p), x)$=($q^{'}$, $p$) such that $\delta_i(q,x)$=$q^{'}$   —2
        - else
            • if $p(i) < n$ then $\delta_{ND}((q,p),\tau)$=($\delta_{p(i+1)}$,$p$)   —3
```

- **else** $\delta_{ND}((q,p),\tau)=f$ —4

From the initial state $init_{ND}$ of the non-deterministic composition and through the transition functionδ_{ND}, the FSA branches to the first advice of each possible permutation (marked 1 in the algorithm). Taking the selected advice of one permutation, all its states are taken into account (marked 2 in the algorithm) in the FSA until a final state is reached. If there is another advice to be composed, the final state of the previous advice is linked to the initial state of the new advice (marked 3 in the algorithm). Otherwise, the final state is related to the final state f of the FSA through the action τ (marked 4 in the algorithm).

Example 2. Let's consider the *filtering* and *encrypting* aspects. The set of permutations is Φ_2=p_1, p_2 such that $p_1 \equiv [1,2]$ and $p_2 \equiv [2,1]$. The result of their non-deterministic composition is given in figure 4.



Such that:

- *F_0...F_3* specify respectively the states of the *filtering* advice: *idle, Active, Filtering, endf*
- *E_0...E_2* specify respectively the states of the *encrypting* advice: *Pending, Encrypting, endE*
- *Fil_1...Fil_3* specify respectively the actions of the *filtering* advice: *initFiltering(), filterWord(), nextWord()*
- *Enc_1* specify the action *Encrypt()* of the *encrypting* advice

Figure 4 – Non-deterministic composition of the encrypting and filtering advices

From the initial state $init_{ND}$ of the non-deterministic composition of the *filtering* and *encrypting* advices, the FSA branches to the first advice of each possible permutation with the action τ. In permutation p_1 we find the *filtering* advice as a first one followed by the *encrypting* advice whereas in permutation p_2 we find the opposite. By taking the *filtering* advice of the first permutation p_1 (the same thing can be said about p_2), all its states are taken into account in the FSA until a final state is reached. At this time, the final state of the *filtering* advice is linked with the initial state of the *encrypting* advice with the action τ . All the states of this *encrypting* advice are considered on the FSA until a final state is reached. This later will be related to the final state f of the FSA with the action τ in order to have one final state.

5.3 Weaving

Since the scheduling problem depend on two levels: design level and implementation level (the weaver's rules), we chose, in this work, to be independent of the weaver's rules since these rules can vary from one weaver to another. Thus, the proposed solution should resolve the problem on the design level. The only rule that should be kept is that the execution of *before* advices should be prior to the execution of *after* advices. Hence, we will not refer to the precedence of aspects nor advices anymore; we rather prefer to use the expression: execution order. This means that all advices will be processed independently of the aspects to which they belong.

When many advices of the same type are defined over the same join point, we initially apply non-deterministic composition in order to have at most one advice of each type (since the weaver considers first the execution of *before* advices then *after* advices). We consider now a base system given by a set of FSA: (A_b, Q_b, int_b, $\{f_b\}$, δ_b). The FSA: (A_{Bc}, Q_{Bc}, int_{Bc}, $\{f_{Bc}\}$, δ_{Bc}) and (A_{Ac}, Q_{Ac}, int_{Ac}, $\{f_{Ac}\}$, δ_{Ac}) represent respectively the non-deterministic composition of *before* and *after* advices. If we suppose that all advices concern the join point j (an action of the base system), this means that the base system includes at least one occurrence of the instruction j otherwise the weaving result is equal to the base system. The weaving of those advices on the base system with respect to the join point j is defined by the FSA: $weaving^j$ (A^j_w, Q^j_w, $init$, $\{f_b\}$, δ^j_w) such that:

- $A^j_w = A_b \cup A_{Bc} \cup A_{Ac} \cup \{\tau\}$
- $Q^j_w = Q_b \cup ((Q_{Bc} \cup Q_{Ac}) \times Q_b)$
- δ^j_w the new transition function is defined by:

– $\delta^j_w(q,x)=q^{'}$ with $\delta_b(q,x)=q^{'}$, $x \neq j$ and $q,q^{'} \in Q_b$ —1
– $\delta^j_w(q,\tau)=(q^{'}, int_{Bc})$ with $\delta_b(q,j)=q^{'}$, and $q,q^{'} \in Q_b$ —2
– $\delta^j_w((q,q^{'}),x)=(q,q^{''})$ with $\delta_{Bc}(q^{'},x)= q^{''}$,$q \in Q_b$, $q^{'} \in Q_{Bc}$–$\{f_{Bc}\}$ and $q^{''} \in Q_{Bc}$ —3
– $\delta^j_w((q,f_{Bc}),j)=(q, int_{Ac})$ with $q \in Q_b$ —4
– $\delta^j_w((q,q^{'}),x)=(q,q^{''})$ with δ_{Ac} $(q^{'},x)= q^{''}$,$q \in Q_b$, $q^{'} \in Q_{Ac}$–$\{f_{Ac}\}$ and $q^{''} \in Q_{Ac}$ —5
– $\delta^j_w((q,f_{Ac}),\tau)= q$ with $q \in Q_b$ —6

According to the above algorithm, the base system changes its states via the transition function δ^j_w (marked 1 in the algorithm) until a join point j is detected (marked 2 in the algorithm). Once a joint point is detected, we first keep the state q in order to have the return state after weaving. After that, the control flow is changed to weave the *before* advices (marked 3 in the algorithm). At this level, the FSA of the non-deterministic composition of the *before* advices is weaved onto the base system. The FSA $weaving^j$ follows the control flow until a final state of the composed *before* advices (marked 4 in the algorithm) is reached. Once the final state of the *before* advices is reached, the control flow is passed to the invoked join point (marked 4 in the algorithm) by using the kept state q. After that, the control flow is passed to the FSA of the non-deterministic composition of the *after* advices. At this time, this later FSA is weaved to the base system and follows the control flow until the final state of the composed *after* advices (marked 6 in the algorithm) is reached. Once reached, the control flow is passed to the base system to continue the sequel of the system.

It is worth noting that this algorithm is slightly changed when only *before* advices (resp. *after* advices) are defined like in the following example.

Example 3. Let's compute the weaving of advices *filtering* and *encrypting* on the base system. Those advices crosscut the base system through the join point *sendText()*. The weaving result is given by the FSA $weaving^{sendtext()}$ ($A^{sendtext()}_w$, $Q^{sendtext()}_w$, $init_b$, $\{S_4\}$, $\delta^{sendtext()}_w$) defined in figure 5.

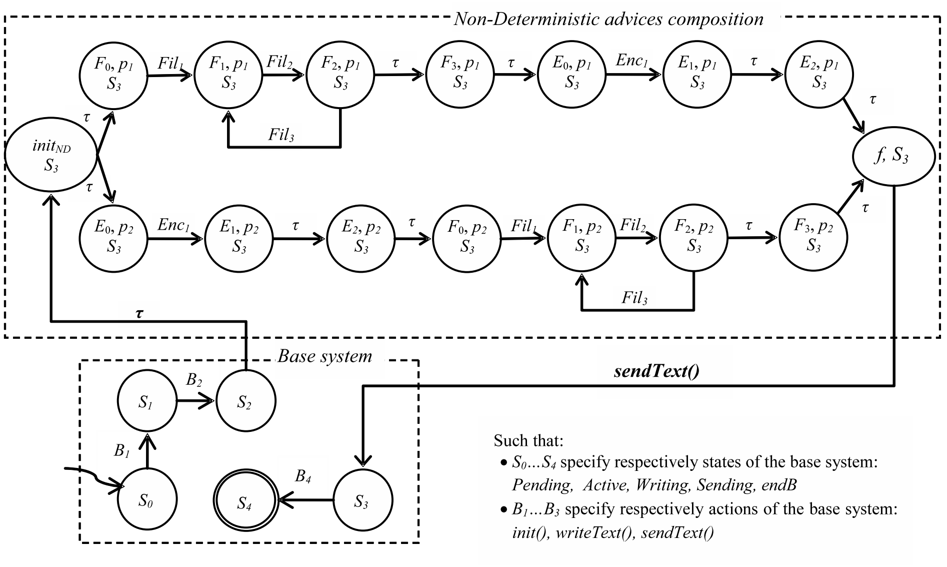


Figure 5 – Graphical representation of the weaving of advices

From the initial state S_0, the base system changes its states until a join point *sendText()* is detected. Once this is done, we first keep the state S_3 in order to have the return state after weaving. After that, the control flow is changed to weave the *before* advices. At this level, the FSA of the non-deterministic composition of the *before* advices is weaved onto the base system. Afterward, the FSA follows the control flow until a final state of the composed *before* advices is reached. Then, the control flow is passed to the invoked join point *sendText()* by using the kept state S_3 which in turn passes the control flow to the base system to continue the sequel of the

5.4 Interaction detection and verification

For the purpose of verification, we give here the definition of paths. Let FSA $=(A, Q, init, \{f\}, \delta)$ be a finite state automaton constructed from a state/transition diagram.

Definition 1. A *path* is a finite or infinite word $q_0\ q_1$ ... such that its alphabet is Q, $q_0 = init$ and $\forall\ i\ \exists\ a$: $\delta(q_i,a)= q_{i+1}$.

When the woven system is produced, we can decide whether the woven system preserves the base system's properties. For instance, we can verify if a given base system temporal property expressed in LTL is still preserved. Let's consider an LTL property ϕ verified by the base system. If ϕ is not verified by the woven system then a counter-example can be generated. This one has the form of uv^ω if we consider infinite executions or uv^* if finite executions are rather considered (u and v are finite paths). Actually, the woven system fails to preserve the base system properties if one of the two problems occurs: either the composition cannot terminate which means that the final state of the base system will not be reached (as explained in [DDF12]), or a precondition of a given action is not verified by the current state of the system.

In the first problem, a counter-example corresponds to a path uv^ω such that v is exclusively composed of states of the form $((q, q^{'}), p)$ such that q is a base system state, $q^{'}$ is an aspect state and p is an execution order. In the second problem, which corresponds to the violation of a precondition, the counter-example is a finite path u. Since the base system is deadlock free and u contains at least one state of the form $((q, q^{'}), p)$.

In both cases, we conclude that the execution orders that figure in u (or possibly v) are bad and should be avoided. Unfortunately, it is not obvious to decide which scheduling order is responsible of the problem if there is more than one conflict.

We notice, however, that this reasoning requires the user to define labels (atomic propositions) for every state in the diagram otherwise classical verification approaches would not be usable. This is not an obvious task since our modeling is based on properties of actions and not on properties of states. In particular, one has to define a Kripke structure in order to be able to apply, among others, the LTL verification procedure (based on Büchi automata [Büc62]). Our aim here is not to verify if some temporal properties are preserved by the woven system. Instead, we are interested in the verification of violation of pre and/or postcondition.

We present two approaches for verifying the non-violation of pre and postconditions. The first one is somehow limited because it assumes some precedence relation between actions. The second one is more general and transforms the non-violation property into a reachability issue.

5.4.1 Precedence based verification

In this approach, we build for each action a the set of all actions that can be executed after a. Formally, we define this set as $\xi_a = \{c \mid post(a) \rightarrow pre(c)\}$ (postcondition of a is compatible with the precondition of c). Obviously, since τ can be always executed, it belongs to any ξ_a.

Once all sets ξ_a are computed, we will be interested by event-based temporal properties. For instance, we consider that writing a means that action a can be executed in the current step, $\mathbf{X}a$ means that a can be executed in the next step, $\mathbf{G}a$ means that a can always be executed, etc.

Now, let P be the system precondition and Q be the system postcondition. We define two special actions in order to consider P and Q. The first action is act_p for which the precondition is *true* and the postconditionis P. The second action is act_Q for which the precondition is Q and the postcondition is *true*. We hence transform every FSA $(A, Q, init, \{f\}, \delta)$ into a Kripke structure $(Q \cup \{ init^{'}, f^{'}\}, R, init^{'}, L)$ such that:

- $(q, q^{'}) \in R$ if $\delta(q,a) = q^{'}$
- $(init^{'},init) \in R$ and $(f,f^{'}) \in R$
- $L(q)= \{a \mid \exists\, q^{'} \in Q\text{: } \delta(q,a) = q^{'} \}$ such that $q \in Q$
- $L(init^{'})= \{act_p \}$ and $L(f)= \{act_q\}$

The preservation problem is transformed into the verification of the following LTL property:

$$(\wedge_{a\in A}\ \mathbf{G}(a \rightarrow \vee_{c\in\xi_a}\ (\mathbf{X}c)) \wedge (\mathbf{F}\ act_Q) \qquad (1)$$

This property means that whenever an action a is executed, the next one must be an action whose precondition is implied by the postcondition of a. For the sake of verification, we assume here that for every state q in the finite state automaton, the transition system δ is defined in such way that $\delta(q,\tau) = q$.

5.4.2 LTL-based verification

The previous analysis relies on operations order; we qualified it as naive since it only considers binary precedence. However, it can be enhanced by rigorously defining precedence between operations but this is a complex task that requires a deep analysis of pre and postconditions.

Instead of basing the reasoning on operation precedence, we can use pre and postconditions in order to label the states of the systems. The labeling algorithm we are presenting depends on substitutions over predicates. We adopt here a definition of pre and postconditions similar to those used in [NKK97].

A precondition (resp. postcondition) is an AND-conjunction of predicates $\psi(x_1,...,x_n)$ (such that $x_1,...,x_n$ are variables). A precondition is allowed to have negations $\neg\psi(x_1,...,x_n)$ meaning that a predicate does not hold. An action a with a precondition $pre(a)$ and postcondition $post(a)$ can change a state s into a state $s^{'}$ with respect to a global property *prop* (an AND-conjunction of predicates) representing the state of the whole system. The execution of a is possible if each predicate in $pre(a)$ is satisfied by *prop*, i.e., $prop \rightarrow \psi(x_1,...,x_n)$. Once executed, all predicates of $pre(a)$ are retired from *prop* and restated by those defined by $post(a)$. We would notice here that this modeling is somehow influenced by our choice to use rewriting systems as a support for model checking (replacing preconditions with postconditions is seen as a rewriting operation). The operation of replacing predicates of $pre(a)$ by those of $post(a)$ in *prop* is written as $prop[pre(a) \setminus post(a)]$.

In order to apply classical LTL verification for example, we construct a Kripke structure. First, a new finite state automaton is built as follows. Suppose $(A,\ Q,\ init,\ \{f\},\ \delta)$ a finite state automaton for which pre and postconditions belong to a set Ω (with respect to variables s and $s^{'}$). Suppose that the precondition of the base system is P and that the postcondition is Q. The new finite state automaton is given by $(A,\ Q \times \Omega \cup\{end\},(init,\ P),\{end\},\delta)$. A state (q,ϕ) figures in this new automaton if ϕ is not *false* in q. The new transition function is given by:

$$\delta^{'}((q,\phi),\ m)=(q^{'},\ \phi\ [pre(m) \setminus post(m)]\ ) \qquad (2)$$

Such that $\delta(q,\ m)=\ q^{'}$ and $\phi \rightarrow pre(m)$. The following transitions are also added to ensure reaching the final state:

$$\delta^{'}((f,\phi),\ \tau)=\ end \qquad (3)$$

Such that $\phi \rightarrow\ Q$.

With this formalization, respecting all pre and postconditions means simply reaching the final state **end**, i.e., it is sufficient to test whether the LTL property **F end** is verified. In other words, we can say that the different advices orders are confluent, i.e, we get the same final result whatever the execution order is.

It is also possible to deal with the problem in another way. Actually, if there are lots of advices to be woven, then exploring all execution orders may be time consuming. We can hence be interested in finding one or some orders that do not violate pre and postconditions (good execution orders). It is sufficient to test whether the LTL property **G** $\neg$ **end** is verified.

6 The rewriting system and its underlying formalism

The use of rewriting systems is suitable for the modeling and the verification of aspect interaction problem by the fact that interaction problem is equivalent to check the confluence of the execution orders. Thus, in order to concretize the proposed framework, we propose a rewriting system to model the base system and aspects. This rewriting system is built with respect to the Maude tool. In the following of these sub sections, we will first give an overview of rewriting logic and Maude tool. We then explain how the UML models are translated into Maude constructions so we can detect undesirable aspect interactions.

6.1 Overview of rewriting logic and Maude

Rewriting logic [CDE+02] is a flexible logical framework for expressing a very wide range of concurrency models and distributed systems. Many rewriting based languages have been proposed; in this paper we use the Maude language [CDE+11]. It is a specification and programming language based on rewriting logic. Data types can be defined as well as their properties by giving signatures, and equations. The behavior is expressed by rules. These rules are of the form "t $\Rightarrow$ t' if C" which indicates that "term t becomes t' if a certain C is verified". The condition C is optional, so rules can be unconditional. Maude also supports object oriented programming and it integrates an LTL model-checker that can be used to verify the required properties.

6.2 Translation of UML models into Maude constructions

This step begins by the translation of the Aspect-UML class and state/transition diagrams into Maude specifications. After that, the weaving process which consists of the detection of the join point, the non-deterministic composition of advices and integration of these advices at the corresponding join point is represented. Note that this implementation is partially based on [BBM10] in which the authors dealt only with Aspect-UML class diagram.

6.2.1 Structural specification rewrite theory (SSRT)

The aim of the SSRT is to model the set of all structural elements (i.e. classes, aspects, attributes, methods) of the class diagram. Our solution is partially inspired by Maude object oriented programming [CDE+11]. Each UML class is specified with a Base-SSRT represented as a module. Each module defines an operator of sort *Cid* (class identifier) to represent the name of UML class. The attributes and methods of classes are respectively represented by operators of sorts *Attribute* and *Msg.* If an object *Obj* of sort *Oid* (object identifier) has a method with no parameter, this method will be represented by an operator such in mark 1. Otherwise, it will be represented by an operator such in mark 2. These operators are represented as:

```
op  ClassName : -> Cid.
op  AttributeClassName : -> Attribute.
op  MethodClassName: Oid -> Msg.        --1
op  MethodClassName: OidParms -> Msg.   --2
```

Aspects are modeled in the same way. Each aspect is represented with an Aspect-SSRT (*Aspect structural specification rewrite theory*). Concretely, this corresponds to a system module. This one defines the name of the aspect with an operator of sort *Aid* (a subsort of the sort *Cid*). The different attributes (resp. methods mark 2 and 3) of an aspect are represented by means of operators of sort *AspAttribute* in mark 1 (resp. *AspMsg*) which is a subsort of the sort *Attribute* (resp. *Msg*). These operators are represented as:

```
sort Aid .          subsort Aid < Cid .
sort AttributeAsp.  subsort AttributeAsp < Attribute .
sort AspMsg .       subsort AspMsg < Msg .
op AspectName: -> Aid .
op AttributeAspectName:-> AspAttribute . --1
op MethodAspectName: Oid -> AspMsg .     --2
op MethodAspectName: Oid Parms -> AspMsg . --3
```

Moreover, every aspect should have an operator DefAdv that defines the advice name, its type (*before* or *after*) and the join point. This will be done thanks to a membership axiom. These operators are represented as:

```
sorts  AdviceName Advice.
ops Before After: -> AdviceType.
op  DefAdv(_,_,_): AdviceName AdviceType Msg -> Advice.
```

6.2.2 Behavior specification rewrite theory (BSRT)

The UML state/transition diagrams describe the behavior of the base system objects and the advices of each aspect. The representation of these diagrams consists of associating a BSRT to each diagram. Each BSRT defines the following configurations *(A,S,T)* where:

- A is the set of possible actions (methods). In Maude, these actions are represented as messages of sort *Msg* (or *AspMsg*).
- S is the set of possible states (of the object or the advice). These states are represented through a set of sorts and operators as:

  $$\texttt{ops } \texttt{s}_1\ \texttt{s}_2 \texttt{...} \texttt{s}_n\ \texttt{:}\ \ \texttt{-> State .}$$

  The initial and final states are represented respectively by operators of sort *InitState* and *FinalState* (subsorts of *State*).
- The set of transitions T specifies how the state changes from a current state S_i to a new one S_j with respect of an executed action *ActionName()* as:

  $$\texttt{S}_\texttt{i} \times \texttt{ActionName() => S}_\texttt{j}$$

  In Maude, each object *Obj* (resp. instance of aspect *AspIns*) of class *Cl* (resp. of an aspect class *Asp*) with the actual state S_i and n attributes is represented by the following syntax (this corresponds to the state of a system):

  $$\texttt{<Obj:Cl|}\ \langle\ \texttt{ObjState:S}_\texttt{i}\ \rangle,\ \langle\texttt{att}_\texttt{1}\texttt{:val}_\texttt{1}\rangle\texttt{,...,}\langle\ \texttt{att}_\texttt{n}\texttt{:val}_\texttt{n}\rangle\texttt{>}$$

  As described in the OCL constraints of the class diagrams, each method of the object *Obj* (resp. *AspIns*) has pre and postconditions. If a method introduces a modification on one or many variables of the object (resp. the aspect) then the state should be expressed by using rewriting variables in order to specify the action effect. Furthermore, if pre/postconditions require some values for the variables, then the required values are directly specified within the object (resp. the aspect) state. By using the new syntax of each object (resp. the different advices of each aspect), a transition is represented as:

```
rl[Object/Advice]:  ActionName()
  <Obj:Cl/ AspIns:Asp |
     ⟨ ObjState/AdvState:Si ⟩, ⟨ att1:val1 ⟩,..., ⟨ attn:valn ⟩>
⇒
  <Obj:Cl/ AspIns:Asp |
     ⟨ ObjState/AdvState:Sj ⟩, ⟨ att1:val1' ⟩,..., ⟨ attn:valn' ⟩>
```

Where $val_1^{'}$,...,$val_n^{'}$ expresses precondition and variables'values. The rule can also be conditional if there are some other conditions that cannot be directly represented in the rule (for example, the state S_i is not equal to certain values $S_i \neq \mathit{ValueState}$).

We notice, however, that this translation does not cope really with the modeling. In fact, since each rule should specify the method's name to be executed, we have to specify every time which methods to be executed next. Formally, consider a set of states Q, a transition $\delta(q_i,m)=q_j$ and $A=\{a_1,..., a_n\}$ a set of methods such that $\forall$ a $\in A$, $\exists\ q_a \in Q$:$\delta(q_j,a)=q_a$ (intuitively, A is the set of methods that can be executed starting from state q_j). Independently of pre and postconditions, the transition $\delta(q_i,m)=q_j$ is translated as follows:

```
rl[Object/Advice]:
      <Obj:Cl | ⟨ ObjState/AdvState:qi ⟩...>
             Active(BeginMethod, m(), BaseSystem/Aspect)
 ⇒
     <Obj:Cl| ⟨ ObjState/AdvState:qj ⟩...>
             Active(EndMethod, m(), BaseSystem/Aspect)
             GetMessage(a1(Obj) a2(Obj)...an(Obj))
```

The predicate (term) *Active(...)* indicates how the system is executed. It is used in order to pass the control to the next method, the first *before* advice or the join point. In this rule, it is used in order to define which method is being executed (when used with the parameters (*BeginMethod*,*m()* and *BaseSystem* or *Aspect*)). The parameters (*EndMethod*,*m()*,*BaseSystem*) mean that the method *m()* has been successfully executed within the base system and hence next methods could be executed (when the last parameter equals *Aspect* this means that the execution is done within an aspect). This is done by building a list $\mathit{GetMessage}(a_1(\mathit{Obj})\ a_2\ (\mathit{Obj})...a_n(\mathit{Obj}))$ which corresponds to methods that can be executed next. A non-deterministic rewriting rule is then used in order to explore all possibilities (X is a variable that indicates which part of the whole system is being executed: the base system or the aspect):

```
rl[NonDeterministicNextMethod]:
    GetMessage(List1 m1:Msg List2) Active(EndMethod, m2, X)
 ⇒
    Active(BeginMethod, m1, X)
```

In order to indicate the methods to be executed in the initial state and to switch between the different systems, we have used the flags *Active(Trigger)* for the base system and *Active(TriggerAdvice)* for the advices. This is done in the following structure:

```
eq Active(Trigger/TriggerAdvice)
=
  <Obj/AspIns:  Cl/Asp | ⟨ ObjState/AdvState:  InitState ⟩>
  Active(EndMethod, Tau, BaseSystem/Aspect)
  GetMessage(a1(...)  a2(...)...an(...))
```

Where *Tau* is a silent message (corresponding to the action τ) used to switch to a given transition system.

6.3 Weaving

6.3.1 Detection of the join point and collection of advices that share this point

This step consists of detecting the joint point specified by the different aspects. The detection is carried out by using a set of conditional equations the goal of which is to monitor the base system messages corresponding to one or more pointcuts. Once a join point is detected, two lists are built: the first one contains the *before* advices (*BeforeList*) while the second one contains the *after* advices (*AfterList*).

6.3.2 Non-Deterministic Composition and Integration of advices

The purpose of the proposed approach is the detection of eventual errors in aspect scheduling when composing conflicting advices. Hence, the weaving has to consider all possible orders. We have used a set of equations and rewrite rules in order to ensure, on the one hand, a non-deterministic composition of the conflicting advices and, on the other hand, the integration of this composition into the base system. We have also used a set of flags (via *Active(..)*) to switch between the base system and the advices when necessary. When a join point is detected, the term *Active(BeginMethod,m(),BaseSystem)* is rewritten as *Active(WeavingBefore)* in order to indicate that control passes to the *before* advices.

The weaver considers a non-deterministic composition of the conflicting *before* (or *after*) advices. This is achieved thanks to Maude rewriting mechanism. In fact, whenever a rule contains variables, Maude considers all their possible instantiations. Hence, it is possible to extract non-deterministically one element from a list. By repeating this process, all permutations of a list can be easily generated. The following rewrite rule illustrates this fact (a similar rule is used in order to process the *after* advices).

```
rl[NonDeterministicBeforeComposition]:
   BeforeList(befAdv Adv befAdv', JoinPoint)
⇒
   BeforeList(befAdv befAdv',JoinPoint) Adv Active(TriggerAdvice)
```

Maude chooses non-deterministically the advice *Adv*. By using the trigger flag *Active(TriggerAdvice)*, this advice *Adv* begins the execution of the rules that reflect its behavior until it achieves its final state. Once the selected advice reaches the final state, Maude proceeds with the composition of the remaining list of the *before* advices (it re-executes the previous non-deterministic before composition rule). Maude will then choose another advice, executes its behavior and so on until it remains no *before* advices to be composed. At this level, the control should be past to execute the invoked join point by using the *Active(BeginMethod,JoinPoint,BaseSystem)* flag which triggers the execution of the invoked join point method. To achieve this, we use the following equation (*noneAdvice* means that the list of advices is empty):

```
eq BeforeList(noneAdvice, JoinPoint)
=
Active(BeginMethod, JoinPoint, BaseSystem)
```

Once the invoked join point is executed, another equation is then used to switch back to the *after* advices:

```
eq Active(EndMethod, M, BaseSystem) = Active(WeavingAfter)
```

The execution of the *after* advices follows the same principle adopted in the execution of the *before* advices.

7 Implementation of the proposed approach with Maude

The implementation of each step explained above is given here.

7.1 Representation of the base system

We present below a part of the representation of the *UserMessage* class with a Base-SSRT and Base-BSRT. These later illustrate respectively the structure and the behavior specification of the *UserMessage* class.

```
mod USER-MESSAGE is
  pr CONFIGURATION .
  sort M-SateValues
  --- The different States
  ops Pending Active Writing Sending -> M-SateValues.
  --- ClassName
  op UserMessage : -> Cid .                           ---1
  ---Attributes
  op Text :       String  -> Attribute .              ---2
  --- Methods
  op SendText :       Oid -> Msg .                    ---3
  ...
  crl[send] : SendText(M)                             ---4
     < M: UserMessage | MState: State, Text : mssg >
   =>
     < M: UserMessage | MState: Sending, Text : mssg >
  if State == Writing.    ...
endm
```

In the above structure, the *UserMessage* class is represented with a system module. This module imports the *Configuration* Maude module in order to represent the essential concepts of the object-oriented systems. The name of the *UserMessage* class is represented by an operator in mark 1. The attributes of this class are represented with operator of sort *Attribute* (mark 2). The methods (we take only one method) are also represented by operators as shown in the above structure (mark 3). The behavior of each method is represented by conditional rewrite rule (mark 4). This rule describes the transition between the *writing* and *sending* states. The term *sendText()* means a message is sent to the object M asking for the execution of *SendText()* method. The execution of this method is not realized unless the preconditions are verified (State = Writing). Once these preconditions are fulfilled, the right-hand side of the rule is reduced and the postconditions of the method *sendText()* replace the preconditions.

7.2 Representation of the aspects

The different aspects are represented with Aspect-SSRT and Aspect-BSRT. These later illustrate respectively the structure and the behavior specification of the aspects. We give here as an example, a part of the translation of the *encrypting* aspect.

```
mod ENCRYPTING is
  pr User-MESSAGE .
  --- The different States
  sort E-StateValues .
  ops Pending Encrypting : -> E-StateValues .
  --- AspectName
  op Encrypting : -> Aid .                                ---1
  ---Methods
  op Encrypt : Oid String -> AspMsg .                     ---2
  ---AdviceName
  op EncryptAdvice: ->  AdvName .
  mb DefAdv(EncryptAdvice,Before,sendText(M): Advice.      ---3

  var M : Oid .
  crl[Encrypt]: Encrypt(M)                                ---4
    < EncryptAdvice: Encrypting | AdvState : Pending >
    < M: Message | MState: State, Text : mssg >
  =>
    < EncryptAdvice: Encrypting | AdvState: Encrypting >
    < M: Message | MState: State, Text : Encrypt(mssg)>
  if mssg =/= null .
endm
```

In the above structure, a part of the representation of the *encrypting* aspect of the figure 1 is illustrated. This aspect is represented by a system module which defines an operator to represent the name of aspect (as it is shown in mark 1). The different methods of this aspect are represented in the mark 2. The name, the type of the advice and the invoked join point are represented with the term *DefAdv* in mark 3. The behavior of each method is represented by conditional rewrite rule (mark 4). The pre and postconditions of this rule are respectively represented in the condition of the rule and the left-hand side of this rule.

7.3 Results and discussion

Once the translation of the different diagrams (Aspect-UML and state/transition) and the representation of the composition and weaving process are done, we proceed to the verification of the aspect interaction problem. The verification consists of exploring all possible aspects orders and verifying if all pre and postconditions are respected. To obtain that, Maude LTL model-checker can be used to verify an LTL property (reachability property). This property is expressed as <> *FinalState* (<> corresponds to ***F***). Hence, the command to be executed is:

```
red  modelCheck (InitialState, <>FinalState)
```

Where:

- *InitialState* represents an initial configuration of the system where the verification should be start. This initial state takes into consideration the base system objects and the archiving and notification aspect which are initially supposed correct.
- *FinalState* represents the final configuration of the system. This final state defines the necessary operators used in the definition of the reachability property.

Using this command, Maude model-checker starts the verification from the defined *InitialState* and test whether all possible orders of advices can lead to the *FinalState* of the base system. Let us use this command with the *encrypting* and the *filtering* aspects. When both aspects are woven to the system, the Maude model-checker reports a counter-example (as it is presented in the following result). The existence of a counter-example means that the base system and the advices behave badly or that the scheduling order is bad. The above counter-example represents the deadlock of the system which means that the reachability of the final state is no longer possible. In general way (as it is explained in section 5.4), a counter-example is a pair (u,v) consisting of two lists of transitions, where the first one u corresponds to a finite path beginning in the initial state of the system until an acceptor or reachable state of the system is found. The second list of transitions v describes a loop which is a cycle on this acceptor state. Noting that each transition is represented as a pair *SystemState*, *RL*, consisting of a state of the system and the label of the rule applied to reach the next transition.

```
reduce in WEAVER-CHECK: modelCheck(InitialState, <> FinalState (A))
result ModelCheckResult:
counterexample (
{ < M: UserMessage | MState: Pending, Text: null>, 'init}
{ < M: UserMessage | MState: Active, Text: null >, 'writeText }
{ < M: UserMessage | MState: Writing, Text: msg >
                        AdvicesOrder(Null), ‘BeforComposition }
{ < M: UserMessage | MState: Writing,  Text: msg >
  < EncryptAdvice: Encrypting | AdvState : InitialState >
              AdvicesOrder(EncryptAdvice), 'TriggerEncrypting }
{ < M: UserMessage | MState: Writing , Text: msg >
  < EncryptAdvice: Encrypting | AdvState : Pending >
                       AdvicesOrder(EncryptAdvice) ,  'Encrypt}
{ < M: UserMessage | MState: Writing, Text: msg' >
   < EncryptAdvice: Encrypting | AdvState : FinalState >
                    AdvicesOrder(EncryptAdvice) ,'FinalEncrypt}
{ < M: UserMessage | MState:Writing, Text: msg' >
               AdvicesOrder(EncryptAdvice) , BeforComposition }
11.    { < M: UserMessage | MState: Writing , Text: msg' >
         < FilterAdvice: Filtering | AdvState : InitialState >
       AdvicesOrder(EncryptAdvice, FilterAdvice) ,  'TriggerFiltering}
12.,
13.    { < M: UserMessage | MState: Writing, Text: msg' >
         < FilterAdvice: Filtering | AdvState : InitialState >
                AdvicesOrder(EncryptAdvice, FilterAdvice) ,  deadlock}
)
```

The first list of transitions u (starts from line 4 to line 12) is composed of a set of pairs. Each pair can be represented as *{(BSState),RL}* if there is no interaction between aspect and base system. Otherwise, the pair is *{(BSState,AState,AdvicesOrder), RL}* such that: *BSState* is a base system state, *AState* is an aspect state which is represented respectively by means of an object and aspect instance, and *AdvicesOrder* is an execution order of advices. The second list of transition v (shown line 13) contains one transition which is represented as a deadlock situation. This situation is due to (note that only the relevant part of the counter-example is shown in the above result): before executing the join point *sendText()*, the system starts the composition of advices (line 6) by choosing the *encrypting* advice as the first advice to be executed (line 7). We have used the *AdvicesOrder* predicate (list of advices) in order to show the advice order composition during the execution. So, the chosen *encrypting* advice intervenes before sending message by trying to encrypt the written message. Once the *encrypting* advice ends, the control flow is passed to the *filtering* advice as it is shown in line 11. At that time, a deadlock (line 13) situation is encountered by the fact that the preconditions of the *filtering* advice are not verified (pre: M.text = msg, see figure 1) and the *filtering* aspect will not be able to filter this text since the message is modified. Consequently, the execution of the *encrypting* advice before the *filtering* advice led to a deadlock situation which blocked the execution of the *filtering* advice and thereafter the execution of the base system. The execution of the *filtering* advice should hence be considered first.

7.3.1 Discussion

The proposed approach allowed us to show how the integration of aspects (as it is the case of the *encrypting* and *filtering* aspects) that share the same join point (*sendText()*) in the base system can violate some properties and hence interact badly with the base system. In general way, the woven system fails to preserve the base system properties if one of two problems occurs: either the composition cannot terminate which means that the final state of the base system will not be reached, or a precondition (or a postcondition) of a given action is not verified by the current state of the system (see section 5.4 for more details). We have therefore tested whether all possible orders of advices can lead to the final state of the system and we have obtained a counterexample. This one indicates that the preconditions of the notification advice are violated when executing the archiving advice which has led to the deadlock of system. Note that the violation of the pre and/or postconditions does not mean necessarily that the base system will be halted but we can say that the whole system would be in an incoherent status, which makes it impossible to predict its future states.

However, although the approach resolves the problem of interaction in design level (the complexity of the problem will be less important than working on the source code level), the verification of all possible orders of advices can be hard to do especially when the number of aspects (advices) is important (the number of all possible orders is as *n!* where n is the number of the advices).

To make our approach scalable, the verification of all possible orders of advices can be avoided in many ways:

1. **Defining a deterministic composition**: In some cases aspects can be independent of each other. This is particularly the case for aspects that do not change the base system variables and do not share any variable (for example, a logging aspect and a tracing aspect). In such case, the composition becomes deterministic and it is sufficient to check one execution order. In Maude constructions this is equivalent to define the composition operator as commutative leading hence to consider all combinations as equivalent.

2. **Defining groups**: On the design level the designer can conceptually define several independent groups of dependent aspects. The idea is to put together aspects that may interact since they use shared variables. The verification does not need to explore all execution orders of all aspects, it is sufficient to verify each group independently of the others. Suppose, for simplicity, p the number of groups and q the number of aspects per group. Instead of exploring *(pq)!* orders, the model-checker will only explore *p(q)!* possible orders.

3. **Looking for the good order**: If no reduction is possible and the number of advices is not important, we can be rather interested in looking for one or a limited number of good execution orders (as explained in section 5.4). In Maude construction this is done by the following command:

   ```
   red  modelCheck (InitialState, []~ FinalState)
   ```

   It is noteworthy that, Maude does not build the entire system, because this one is rather built on *demand* (or *on-the-fly*). In fact, Maude starts the verification of the advices order by the *pairwise* checks between the advices. Suppose that the execution order to be explored is the following:

   $$(((a_1, a_2), a_3)..., a_n)$$

   If the verification of a pair of the advices (for instance a_1 and a_2) fails, then the whole execution order is aborted and a counter-example is generated. This avoids the verification of the remaining order once a problem is detected in the current pair order, and thus minimizes the number of the checked states (the same behavior of Maude is done if there is a problem between the base system and the advices). When the *pairwise* verification of the advices succeeds, it passes to another pair of the advices (for instance the result of a_1, a_2 with a_3) until a correct order is found.

All these propositions reduce the state-space size of the verification and the simulation time. In addition, the state-space size can be reduced by the nature of Maude that can detect isomorphic states in branches and can merge them into one branch. This means that, during the rewriting mechanism, if Maude finds states that have the same result then it can merge them into a single state. This mechanism can be done when Maude starts the verification of the commutative aspects, thus the result of each order of these aspects is the same. Consequently, Maude merges all the branches of these orders into on branch.

8 Related Works

Several works were dedicated to detect and verify aspect interaction. We can classify these works into two categories depending on what level they actually act: *source-code* level and *design model* level.

In the former category, the authors of [ZZ07] presented six bug patterns in AspectJ which provide to both researchers and programmers a clear view of what kind of bugs may happen in AspectJ programs and how to detect them. The study of bug patterns mainly focused on the aspects of bug pattern symptoms, cause root, cures and preventions. This work has been extended in [SZZ+8] with more bug patterns in AspectJ. Thus, the authors of [SZZ$^+$08] presented XFindBugs, an eX-tended FindBugs for AspectJ, to help programmers to find potential bugs in AspectJ applications through static analysis. XFindBugs supports 17 bug patterns to cover common error-prone features in an aspect-oriented system, and integrated the corresponding bug detectors into the FindBugs framework. The authors evaluated XFindBugs on a number of large-scale open source AspectJ projects. Beside the fact that this approach is based on a source code, we think that the detection of aspect interaction problem at the code level may even become unnecessary and more expensive at the same time.

In general, the *source-code* category of works can be divided in its turn into two subcategories: *syntactic* and *semantic* approaches. The focus of *syntactic*-based approaches [DFS02, SK03b, SdM03] is limited to analyze aspects sharing join points and updating common variables of base systems in order to detect aspect conflicts. For instance, Douence et al. [DFS02] presented a framework characterized by a very expressive crosscut language, static conflict analyses and linguistic support for conflict resolution. The authors proposed to model the program as an observable execution trace and aspect as rules specifying instructions to insert at an execution state. The weaver process is modeled by analyzing aspects rules and determining the set of rules applicable to the current join point. In this paper, the notion of aspects independence is a sufficient condition to ensure that weaving is well defined. Thus, two general independence properties (*strong independence* and *independence w.r.t a program*) have been presented. For each case, an algorithm is provided to check whether two aspects are independent. Finally, the authors proposed some commands for conflict resolution. This framework is later extended to other type of aspects (stateful aspects) [DFS04] that deal with inter-crosscut variables. Beside the fact that these approaches are based on source code, the authors concentrate their analysis on the detection of conflicts revealed by the syntax which is limited regarding coverage and precision.

To increase the accuracy and the significance of the verification process, *semantic* interference analysis should rather be considered. In this area, many approaches have been proposed. For example, Xu et al. [XR07] extended a regression test selection technique for Java to take aspects into account. The authors proposed a source-code-based control-flow representation of AspectJ programs, referred to as the AspectJ Inter-module Graph (AJIG). An AJIG includes (1) Control-Flow graphs (CFGs) that model the control flow within Java classes, within aspects, and across boundaries between aspects and classes through non-advice method calls, and (2) interaction graphs that model the interactions between methods and advices at certain join points. An AJIG captures the semantic intricacies of an AspectJ program without introducing extra nodes and edges to represent the low-level details of compiler-specific code. The authors defined a two phase graph traversal algorithm that identifies differences between two versions of an AspectJ program. The algorithm is specifically designed to take into account the interactions between methods and advices at join points, which is allowed by our approach too, but unlike our work they consider the precedence between the set of advices that shared the same join point. The authors implemented the regression test selection technique and performed an experimental evaluation of its precision and cost.

In a series of papers [KS03, SK03a, GK06, KK08], Katz and his group addressed various issues of model checking aspect-oriented code. In [KK08], the authors proposed the use of LTL formulas to define a set of assume-guarantee properties of aspects. The assume properties are general properties which any base program should satisfy otherwise the aspect cannot be woven to that program. The guarantee properties are satisfied by the program after weaving the aspect. A pairwise check is performed by the approach: two aspects are interference-free if when they are woven to a base program satisfying their assumption properties (the guarantee properties should be satisfied after weaving). Maven model-checker is used for automatic verification of properties. The proposed approach is interesting because it relies on the semantic interference among aspects, but, in contrast to our work, the authors did not consider the aspect scheduling problem in first preoccupation since they focus on the verification of LTL properties. In an extended work with Goldman [GKK10], state machines are used to model the base program, the aspects, and the woven system. The weaving process is implemented by integrating the aspect state machine directly to the base system state machine. However, the approach only focuses on one aspect at a time and they only consider weakly invasive aspects. Moreover, when interference is detected (i.e. a property is not satisfied) the programmer is responsible to fix it (they don't provide composition operators as it was presented in our approach) and thus the approach does not provide any solution. In fact, this work is proposed to cope with the limitations of Krishnamurthi et al. [KF07] proposition, where a state machine is defined for each advice, and focused on treating aspects not modifying data variables of base systems. The approach is interesting because it can simplify the verification of the system, but it can only be used if the aspects are independent, unlike our approach.

We can also cite, in this subcategory (*semantic*-based approaches), the work of [ARS09] in which the authors proposed an approach to detect aspect interference at shared join points using the aspect-oriented language: Composition Filters (CF) [AT98]. First, the authors specified the semantic of each syntax element of the language. After that, the graph of the join point model is generated from the source code of the program. The execution of the system is then simulated by considering all advice orders. This simulation checks whether the execution orders are confluent (i.e. the same final result whatever the execution order is) in which case we can conclude that the system is interference free. Beside the fact that this approach is based on source code, it lacks to tell why the advices behave badly with each other and/or with the base system.

The authors of the *design model* category of works tried to integrate aspects within abstract models to ensure early detection of interaction problem. For instance, the works of [MV07, Mos08], which is one of the works that inspired our contribution, introduce a new UML profile named Aspect-UML as an extension of the classic UML use case and class diagrams. The new use case diagram allows integrating aspects as extensions of the base system use cases. Meanwhile, the new class diagram consists of adding aspects as a kind of new classes and using OCL to annotate both methods and advices. The annotations consist of defining the pre and postconditions of every action. The system can be verified by using use-case scenarios in order to check whether the system violates pre and/or postconditions. Several methods were hence applied to make that such as: software abstraction [MV07], colored Petri Nets [Mos08] and rewriting systems [BBM10]. Compared to Alloy, considered as a simple checker in [MV07], Maude is less restricted. If in Alloy, we should set a limit on the size of the instances to generate (*small scope hypothesis*), Maude does not have this limit except that we must fix the maximum tree depth rewriting if the number of states is infinite. According to the designers of Maude [CDE+11], we can solve this problem by using a more abstract modeling of the problem.

[XAXW09] presented an approach for modeling and verifying an aspect-oriented system in a general way. The authors defined class and aspect models with state machines. These models are then composed and weaved in a final model that will be verified against desired system properties. However, the weaving process was not rigorously defined and the authors did not consider the scheduling problem since they supposed a predefined execution order. In the same axis, the authors of [CHA+10] proposed a technique and tool for the detection of conflicts between aspects at the model level. The proposed approach works on models defined in UML with an extension for modeling pointcuts and advice. The authors have defined graph-transformation rules that constitute an operational semantics for aspect-oriented models. By implementing this operational semantics with a model-checker GROOVE [KR06], they can automatically detect violations of invariants at the modeling level. Beside the fact that this approach relies on the detection of the semantic interference among aspects after weaving, aspect composition and the weaving process was not formally defined. In addition, our aim in this work is to show that the interaction problem can be transformed into a generic property which is independent of the system to be verified and hence enables the use of any model-checker to verify bad interaction. However, the verification of invariants changes depending on the studied system (this is also the case of the verified properties of [XAXW09]).

In addition, the authors of [CYD10] considered an aspect model in which every action (method or advice) is described by a pre and a postcondition. By using the weakest precondition concept, the authors defined many conditions that should be fulfilled in order to avoid bad interaction. Unlike the work of Chen et al [CYD10], our underlying formal foundations are different. Their work is built on a denotational semantics and theorem proving, while our work is based on finite state automata and the interaction problem is transformed into a generic LTL property which is independent of the system to be verified. Hence, it enables the use of any model-checker like Maude to verify bad interaction.

The aspect problem has also been considered in approaches that combine components and aspects. [HDA11] proposed a model of aspectualized components and used UPPAAL [BLL+95] to verify if the introduction of a set of aspects does not affect some properties of the system (expressed as a CTL [BMP81] property).

We can also cite in the *design model* category, the work of Zhang [Zha10] which is one of the inspirations for this work. The author proposed the HILA diagrams, which are an aspect-oriented extension of state/transition diagrams, in order to model aspect behavior while considering the parallelism case on the aspects. The author proposed a weaving algorithm in order to produce a new diagram that represents the woven system. An interaction problem rises then when some states of the base system or the aspects could not be reached. However, this work considered only guards: conditions that have to be verified in order to execute a given advice. Zhang had also defined a criterion for the existence of conflicts but does not propose a verification method to detect them. We can also notice that the aspect model considered by Zhang is not close to the one defined by AspectJ as in our case. In addition, Zhang et al [ZH12] presented the weaving algorithms of aspects in HILA diagrams. In particular, they show how the weaving process detects potential conflicts between different aspects. Thus, the authors have been used the model-checker Hugo/RT in order to check some global properties which is not always obvious to define them since this requires reasoning about the whole system. In addition, the verification of these properties depends on the studied system which isn't the case of our work.

9 Conclusion

In this paper, we have investigated the aspect interaction problem in general and aspect scheduling in particular. We have presented a framework that uses many UML diagrams (Aspect-UML class and state/transition diagrams) and transforms them into FSA in order to discover bad execution orders and/or bad aspect interaction with the base system. In the proposed framework, classes and aspects are first defined in Aspect-UML class diagram which makes it possible to define, among others, pre and postconditions of methods. The behavior is then specified in state/transition diagrams. After that, an FSA that represents the woven system is built. We have shown that scheduling problem can be transformed into a reachability issue and thus could be expressed as a generic temporal property which is independent of the system to be verified.

In order to concretize our work, we have proposed a rewriting system to model the base system as well as the aspects. Firstly, the Aspect-UML class diagram that describes the structural elements is transformed into functional modules by defining sorts, operations and membership predicates. Then, all state/transition diagrams are transformed into system modules that define behavior as rewrite rules which are suitable to the representation of the pre and postconditions. Finally, the third step takes up the implementation of the weaving process. Afterward, we have verified the generated specifications with Maude LTL model-checker that benefits from some of the most recent advances in *on-the-fly* technique which lead to a non-systematic exploration of all possible combinations of the verified aspects.

The proposed approach can be enriched in many ways. For instance, only Aspect-UML class and state/transition diagrams have been used. It would be interesting to extend our approach by considering other UML diagrams (such as collaboration diagrams). Furthermore, the current work deals only with *before* and *after* advices. Moreover, it relies on the fact that pointcuts correspond only to the base system. It can be improved in different ways. The presented framework can be extended by integrating the *around* advices and considering more general kinds of pointcuts by defining them on aspects. It can be also interesting to investigate the relationship between the proposed framework and special aspect classes (like those proposed by [Kat06], [DDF12]). It would be interesting to have a better characterization of the non-violation of pre and postconditions depending on the classes of advices. For instance, consider two observing aspects [DDF12] (i.e. aspects that can introduce new instructions and a new local state but do not modify the base system' state and control-flow) which are independent (i.e. an aspect cannot modify the local state of another aspect). If there is a violation of pre or postconditions when weaving these independent aspects on the base system, we can conclude that the problem is due to a bad interaction between the aspects and the base system and not between the two aspects since we have assumed that they cannot modify the local states of each other.

Finally, the aim of the presented implementation is to show the feasibility of the proposed approach and to offer a preliminary support of aspect programming in Maude. The proposed implementation respects our modeling but contains some details in code the goal of which is to ensure the composition of advices and weaving. Although, this code is independent of the system, it is preferable to hide all these details. In addition, it seems to be interesting to integrate the aspect-oriented paradigm as an attractive support in rewriting systems (as it is the case of object-oriented paradigm in Maude [CDE+11]). Thus, we think that it will be better to design an extension to Maude by using meta-rewriting for example in order to realize this goal.

References

[ARS09] Mehmet Aksit, Arend Rensink, and Tom Staijen. A graph- transformation based simulation approach for analysing aspect interference on shared join points. In *Preceedings of the 8th ACM international conference on Aspect-Oriented Software Development*, pages 39–50, Virginia, USA, March 2009. `doi:10.1145/1509239.1509247`.

[AT98] Mehmet Aksit and Bedir Tekinerdogan. Aspect-oriented programming using composition filters. In *Preceedings of the European Conference on Object-Oriented Programming*, Germany, July 1998. `doi:10.1007/3-540-49255-0_132`.

[BBM10] Amina Boudjedir, Toufik Benouhiba, and Djamel Meslati. Verification of aspect composition and integration using rewriting systems. In *Preceedings of the First International Symposium on Modeling and Implementing Complex Systems*, pages 138–148, Algeria, May 2010.

[BLL+95] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal- a tool suite for automatic verification of real-time systems. In *Preceedings of the 4th DIMACS Workshop on Verification and Control of Hybrid Systems*, pages 232–243, Heidelberg, June 1995. `doi:10.1007/BFb0020949`.

[BMP81] Mordechai BenAri, Zohar Manna, and Amir Pnueli. The temporal logic of branching time. In *Preceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 164–176, New York, 1981. `doi:10.1145/567532.567551`.

[Büc62] J.Richard Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the International Congress in Logic, Method and Philosophical Sciences*, pages 1–11, California, 1962.

[CD05] Michelle L. Crane and Juergen Dingel. On the semantics of uml state machines: Categorization and comparison. Technical report, School of Computing, Queen's University, Canada, 2005.

[CDE+02] Manuel Clavel, Francisco Duran, Steven Eker, Patrick Lincoln, Narciso Marti-Oliet, José Meseguer, and Carolyn Talcott. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, August 2002. `doi:10.1016/S0304-3975(01)00359-0`.

[CDE+11] Manuel Clavel, Francisco Duran, Steven Eker, Patrick Lincoln, Narciso Marti-Oliet, José Meseguer, and Carolyn Talcott. *Maude Man-*

ual (version 2.6), January 2011. http://maude.cs.uiuc.edu/maude2-manual/maude-manual.pdf.

[CHA$^+$10] Selim Ciraci, Wilke Havinga, Mehmet Aksit, Christoph Bockisch, and Pim van den Broek. *A graph-based aspect interference detection approach for UML-based aspect-oriented models*, volume 6210, page 321–374. Springer, 2010. `doi:10.1007/978-3-642-16086-8_9`.

[CYD10] Xin Chen, Nan Ye, and Wenxu Ding. *A Formal Approach to Analyzing Interference Problems in Aspect-Oriented Designs*, volume 6445, pages 157–171. Springer, 2010. `doi:10.1007/978-3-642-16690-7_7`.

[DDF12] Simplice Djoko Djoko, Rémi Douence, and Pascal Fradet. Aspects preserving properties. *Science of Computer Programming*, 77:393–422, March 2012. `doi:10.1016/j.scico.2011.10.010`.

[DFS02] Rémi Douence, Pascal Fradet, and Mario Südhot. *A Framework for the Detection and the Resolution of Aspect Interaction*, volume 2487, page 173–188. Springer, September 2002. `doi:10.1007/3-540-45821-2_11`.

[DFS04] Rémi Douence, Pascal Fradet, and Mario Südhot. Composition, reuse and interact analysis of stateful aspect. In *Preceedings of the 3rd International Conference Aspect-Oriented Software Development*, pages 141–150, 2004. `doi:10.1145/976270.976288`.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. *Rewrite Systems*, volume B: Formal Models and Semantics, chapter 6 of Handbook of Theoretical Computer Science, pages 243–320. ACM, Amsterdam, 1990.

[EMS03] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. *The Maude LTL Model Checker and its Implementation*, volume 2487 of *Lecture Notes in Computer Science*, page 230–234. Springer, Berlin, May 2003. `doi:10.1007/3-540-44829-2_16`.

[FECA04] Robert Filman, Tzilla Elrad, Siobhan Clarke, and Mehmet Aksit. *Aspect-Oriented Software Development.* Addison Wesley Professional, 2004.

[GK06] Max Goldman and Shmuel Katz. Modular generic verification of ltl properties for aspects. In *Preceedings of the Foundations of Aspect Languages Workshop*, Germany, 2006.

[GKK10] Max Goldman, Emilia Katz, and Shmuel Katz. Maven: modular aspect verification and interference analysis. *Formal Methods System Designs*, 37:61–92, November 2010. `doi:10.1007/s10703-010-0101-1`.

[HDA11] Abdelhakim Hannousse, Rémi Douence, and Gilles Ardourel. Static analysis of aspect interaction and composition in component models. In *Proceedings of the 10th ACM international conference on Generative programming and component engineering*, pages 43–52. ACM, 2011. `doi:10.1145/2047862.2047871`.

[Hop79] John E. Hopcroft. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 1979.

[Kat06] Shmuel Katz. *Aspect categories and classes of temporal properties*, volume 3880, pages 106–134. Springer, 2006. `doi:10.1007/11687061_4`.

[KF07] Shriram Krishnamurthi and Kathi Fisler. Foundations of incremental aspect model-checking. *ACM Transactions on Software Engineering and Methodology*, 16(7), 2007. `doi:10.1145/1217295.1217296`.

[KHH$^+$01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. Getting started with aspectj. *Communications of the ACM*, 44:59–65, 2001. `doi:10.1145/383845.383858`.

[KK08] Emilia Katz and Shmuel Katz. Incremental analysis of interference among aspects. In *Proceedings of the 7th workshop on Foundations of aspect-oriented languages*, pages 29–38. ACM, 2008.

[KR06] Harmen Kastenberg and Arend Rensink. *Model Checking Dynamic States in GROOVE*, volume 3925, pages 299–305. Springer, 2006.

[KS03] Shmuel Katz and Marcelo Sihman. *Aspect validation using model checking*, volume 2772, pages 373–394. Springer, 2003.

[Mos08] Farida Mostefaoui. *Un cadre formel pour le développement orienté aspect :modélisation et vérification des interactions dues aux aspects.* PhD thesis, University de Montreal, August 2008.

[MP92] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems.* ACM, 1992.

[MV07] Farida Mostefaoui and Julie Vashon. Design level detection of interactions in aspect uml models using alloy. *Journal of Object Technology*, 6(7):137–165, August 2007.

[NKK97] Masahide Nakamura, Yoshiaki Kakuda, and Tohru Kikuno. Petri net based detection method for non deterministic feature interactions and its experimental evaluation. In *Feature Interactions in Telecommunications and Distributed Systems IV*, pages 138–152, July 1997.

[OMG09] OMG. *Unifed Modeling Language Superstructure (Version 2.2.)*, 2009. URL: `http://www.omg.org/spec/UML/2.2/Superstructure/PDF/`.

[SdM03] Damien Sereni and Oege de Moor. Static analysis of aspects. In *Proceedings of the 2nd international conference on Aspect oriented software development*, pages 30–39, 2003. `doi:10.1145/643603.643607`.

[SK03a] Marcelo Sihman and Shmuel Katz. Model checking applications of aspects and superimpositions. In *Proceedings of the Foundations of Aspect Oriented Languages*, pages 51–60, Germany, March 2003.

[SK03b] Maximilian Störzer and Jens Krinke. Interference analysis for aspectj. In *Foundations of Aspect-Oriented Languages*, pages 35–44, March 2003.

[SP05] Lionel Seinturier and Renaud Pawlak. *Foundations of AOP for J2EE Development.* Eyrolles, 2005.

[SZZ$^+$08] Haihao Shen, Sai Zhang, Jianjun Zhao, Jianhong Fang, and Shiyuan Yao. Xfindbugs: extended findbugs for aspectj. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 70–76. ACM, 2008. `doi:10.1145/1512475.1512490`.

[TCZY09] Kun Tian, Kendra Cooper, Kang Zhang, and Huiqun Yu. A classification of aspect composition problems. In *Proceedings of Third IEEE International Conference on Secure Software Integration and Reliability Improvement*, pages 101–109. Shanghai, IEEE, 2009. `doi:10.1109/SSIRI.2009.33`.

[XAXW09] Dianxiang Xu, Omar El Ariss, Weifeng Xu, and Linzhang Wang. Aspect-oriented modeling and verification with finite state machines. *Journal of computer science and technology*, 24:949–961, September 2009. `doi:10.1007/s11390-009-9269-5`.

[XR07] Guoqing Xu and Atanas Rountev. Regression test selection for aspectj software. In *Proceedings of the 29th International Conference on Software Engineering*, pages 65–74. IEEE, 2007.

[ZH12] Gefei Zhang and Matthias Hölzl. Weaving semantic aspects in hila. In *Proceedings of Transactions on Aspect-Oriented Software Development*, pages 263–274. ACM, 2012. `doi:10.1145/2162049.2162080`.

[Zha10] Gefei Zhang. *Aspect-Oriented State Machines.* PhD thesis, University Ludwig Maximilians, November 2010.

[ZZ07] Sai Zhang and Jianjun Zhao. On identifying bug patterns in aspect-oriented programs. In *Proceedings of the 31st Annual International Computer Software and Applications Conference*, pages 431–438, Beijing, July 2007. IEEE. `doi:10.1109/COMPSAC.2007.159`.

About the authors

Toufik Benouhiba is an associate professor at Annaba Badji Mokhtar university (Algeria). He obtaind the PhD degrees from the University of Technology of Troyes (France). He is a member of the LISCO Laboratory and works on formal and semi-formal verification approaches in addition to the using of metaheuristics for testing and verification. Contact him at `toufik.benouhiba@gmail.com`.

Amina Boudjedir is a PhD. student at the University of Badji Mokhtar-Annaba in Algeria. She has obtained her Master of Science degree in Computer Science from the University of Badji Mokhtar-Annaba, in 2009. She is a member of the LISCO Laboratory, pursuing a PhD. thesis on using rewriting systems for modeling and verification of aspects-oriented applications. This thesis is followed under the supervision of the Associate Professor Toufik Benouhiba and the Professor Djamel Meslati. Her main areas of interest include software engineering and more specifically Aspect-Oriented Paradigm, and Formal Methods. Contact her at `a.boudjedir@hotmail.fr`.

Acknowledgments We would like to thank the Professor Djamel Meslati; the head of the Laboratory of Complex Systems Engineering "Laboratoire d'Ingénierie des Systèmes COmplexes" (LISCO) for the comments and suggestions on the work.