Specifing and Verifing Aspect-Oriented Systems in Rewriting Logic

Amina BOUDJEDIR
LISCO Laboratory, Department of computer science
Badji Mokthar University. Annaba, Algeria
a.boudjedir@hotmail.fr

Toufik BENOUHIBA[1], Djamel MESLATI[2]
LISCO Laboratory, Department of computer science
Badji Mokthar University. Annaba, Algeria
[1]toufik.benouhiba@gmail.com
[2]meslati_djamel@yahoo.com

Abstract – Aspect-oriented (AO) systems have to deal with an important problem which is the management of aspect interaction. In this paper, we introduce a first tool, known as AO-Maude, which is based on Maude language for the specification and the verification of the AO systems. The proposed tool relies on the reflection feature of rewriting logic that allows us to represent in the Meta-Level the structure of the base system, aspects and the weaver mechanism. The contributions of this paper are twofold. First, we provide a support for the specification of the AO systems in Maude language and thus discharge the user from the task of the definition of the weaver mechanism each time. Second, our extension offers a support to the AO systems in Maude while managing the aspect interaction problem in general and the scheduling problem in particular. The proposed tool is illustrated with a concrete case study.

Keywords – Aspect-oriented system; aspect interaction; Aspect-UML; rewriting logic; Maude; Meta-Level; verification

1. INTRODUCTION

Aspect-oriented (AO) systems have been proposed to capture transversal preoccupations. They are considered as a necessary complementarity to the object oriented systems [1]. Generally speaking, an AO application is composed of two parts: *Base system* to implement the system functions and *Aspects* to implement the Cross-cutting concerns. An Aspect also consists of two parts: *pointcut* and *advice*. A pointcut is a set of many join points where an advice should be executed. 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 AO weaver ensures the integration of the base system and aspects functionality.

However, AO weaver can drastically change the semantic of the base system (e.g. some properties can be affected by the introduction of some aspects [2]) and thus unexpected results can emerge. In the AO, this issue is commonly known as the aspect interaction problem [1, 3]. In fact, there are many kinds of aspect interaction problem: dependence, scheduling, redundancy, etc [1]. For example, the scheduling problem, which is the subject of this paper, occurs when many independent aspects are concerned by the same joint point. In this case, the execution of these aspects may have some undesirable effect on the base system if they are executed in any order. Some of these orders can interact badly with the properties of the base system. In such circumstances, the aspects interfere with each other in a potentially undesired manner and they can be used in a harmful way that invalidates desired properties and thus change the semantic of the base system. Note however that the presence of this

conflict does not lead necessarily to a violation of base system properties.

Many works have tried to tackle this problem in different ways. By using the model-checking technique, several approaches have been proposed for the verification of aspect-oriented systems. The authors of [4] define incremental aspect model-checking which consisted to modularize the verification of aspects (i.e. verify properties against aspect without having access to the program source). The authors of [5] present an approach for modeling and verifying aspect-oriented systems with finite state machines. They define class and aspect models with state machines. These models are then composed and weaved into a final model via weaving mechanism. Once the model that represents the entire system is generated, they proceed to verify the system against the desired system properties by using the LTSA (Labelled Transition System Analyser) model checker [6]. However, the weaving process was not rigorously defined and the authors did not consider the scheduling problem since they suppose a predefined execution order. Another attempt for the formal verification of the aspect-oriented systems exploits the techniques of model checking. We can cite the proposed approach of [7]. This approach builds the aspect model and verifies the deadlock problem with Spin model checker [8]. In a series of papers [9, 10, 11], Katz and his group have addressed various issues of model checking aspect-oriented code. For instance in [11], the authors suggest an assume-guarantee structure to achieve modular and generic verification of AO systems. They verify that for any base state machine satisfying the assumptions of a given aspect, the woven state machine is guaranteed to satisfy the desired properties.

Depending on the source-code level, several works have been proposed in the area of the static aspect analysis [12, 13] where aspect conflict can be detected depending on pointcut definitions. We can also cite the work of [14] in which the authors present a language named compAr in order to model aspects with *around* advices. However, the complexity of the source-code can be an important drawback of these approaches. In addition, the aspect interaction problem has to be detected and fixed in early development stages in order to minimize maintenance costs.

Depending on the design level, different approaches [15, 16, 17,] tried to integrate aspects within abstract models to ensure early detection of interaction problem. For instance, we propose in [17] a rewriting system [18] in order to verify and detect bad aspect interaction. We used the Aspect-UML [15] which is a UML profile that extends the classic UML use case and class diagrams with different concepts of AO. We translated the base system of the Aspect-UML models into Maude [19] specifications. Then the aspects and their subsequent concepts are translated into Maude specification. Finally, a weaving step is defined in order to integrate the aspects into the base system. Afterwards, all these specifications are formally verified by the Maude tool in order to detect possible conceptual errors concerning aspect interactions. Although, the result of the proposed approach helps us to detect bad aspect interaction, the implementation of the approach contains some *messy* code. In fact, in the early proposed approach the user specifies not only the aspect models, but also the aspect composition and the weaver process. This later makes the task very tedious to do it each time.

In this work, we aim to provide a support that hides all the details of the weaver. The user thus cares only about the specification of the base system and aspect. This support is an extension of the rewriting systems in general and of Maude language in particular for the specification and verification of Aspect-UML models. This extension is realized as a first AO-Maude support. This one relies on the reflection feature of Maude system which allows us to represent in the Meta-Level: the general form of base system and aspects of the Aspect-UML models, the processing of the aspect composition and the weaver mechanism. The user represents only the Aspect-UML models by *base* and *aspect* modules. Afterward, he gives the task of the composition and integration of the aspects within the base system to the defined weaver. The result of this composition and integration is examined later in order to detect and verify aspect interaction problem.

This paper is organized as follows. In section 2, we give an overview on the rewriting logic and the reflective capabilities of the Maude system. In section 3, we outline the main phases adopted in the realization of our support. We illustrate, in section 4, the proposed support in a case study. Finally, section 5 concludes the paper.

2. REWRITING LOGIC AND THE META-LEVEL OF MAUDE

Rewriting logic is introduced by José Méseguar [19]. This logic is a reflective framework for

expressing a very wide range of concurrent systems and languages. Thus, many languages based on this logic (ASF+SDF [20], CafeOBJ [21], Maude have been proposed.

In this paper, we use Maude language which is a specification and programming language. It allows us to define data types by giving signatures and equations. The behavior is specified by the use of rewrite rules. Maude also supports the modeling of object oriented systems and integrates an LTL model-checker that can be used to verify the required properties. This modeling and verification is supported in different ways. Currently, Maude offers two ways (the *Core* Maude and *Full* Maude) to support that. The two ways are similar but are based on different levels of the language. In addition, Maude language supports some Meta-functionalities [19] that help us to build new environments and languages by implementing an extension of Maude. Full Maude is the real example of the extension of Maude. It endows the language with an even more powerful and extensible *module algebra* [22]. Full Maude itself can be used as a basis for further extensions by adding new functionality. It is possible both to change the syntax or the behavior of existing features and to add new features. In this way, many concurrent systems have inspired extensions to different kinds of systems via *Full* Maude specifications such as: real-time system [23], probabilistic system [24], etc. Thus, since *Full* Maude offers a way to define new environments and tools; we agreed to use its functionalities in order to provide an attractive support for the modeling and the verification of aspect-oriented systems by rewriting systems. These features allow us to define not only the weaver mechanism at the Meta-Level, but also to avoid the re-implementation of the code and thus take advantage of the infrastructure provided.

3. OVERVIEW ON THE AO-MAUDE

The aim of our work is to define a support in which all details of the aspect composition and the weaver mechanism are hidden. This is done by defining the syntax of each *base* model, *aspect* model, aspect composition and the weaver mechanism at the Meta-Level of the AO-Maude. The idea behind this definition is to rid the user from the task of the definition of aspect composition and weaver mechanism each time. The user has only to represent the Aspect-UML models by *base* and *aspect* modules. Afterward, he gives the task of the composition and integration of the aspects within the base system to the defined weaver. As it is shown in Figure 1, the AO-Maude is divided into two parts: what we have defined in the Meta-level and what the user should write. The specification of AO-Maude follows the following steps:

3.1. Definition of a useful module

The aim of this step is to define a module that specifies the different concepts of the aspect model (i.e. aspect type/sort, attributes sort, methods sort and general form of advices). All these elements are an extension of other concepts in Maude. The idea behind this definition is to provide a generic module that can be imported at any time by the user as well as some other Maude module (likes the *Nat* module for natural number, etc).

3.2. Definition of base/aspect modules' syntax

Since the AO-Maude is a first proposed tool, we agreed to specify all the declarations and statements of *base* and *aspect* modules in the same manner of Maude modules style (i.e. we keep all the different concepts of these modules such as: sorts, operators, equations, rules, etc). This idea allows as not only to avoid the re-implementation of the code (i.e. defining new parser and compiler) and thus taking advantage of the infrastructure provided, but also to rid the user of the step of the learning of new syntax. However, in order to make the deference between the Maude modules and AO-Maude (*base* and *aspect*) modules, we have enclosed the *base* module body between the keywords *bmod* and *endbm* and the *aspect* module body between the keywords *amod* and *endam.*

3.3. Transformation of the base/aspect modules into ordinary modules

The aim of our work is to provide a support that allows the user to specify the aspect models and detect aspect interaction. The detection of aspect interaction is based essentially of the analysis of the preservation or the violation of the pre/postconditions of method/ advice. This principle has been used in our previewed work [17], where the user specifies the behavior of each *method/advice* by two rewriting rules in order to detection at the end aspect interaction. The first rewriting rule is used in the case where the method/advice preconditions are preserved whereas the second rule is used when these preconditions are violated. However, we think that it would be better to unload the user from

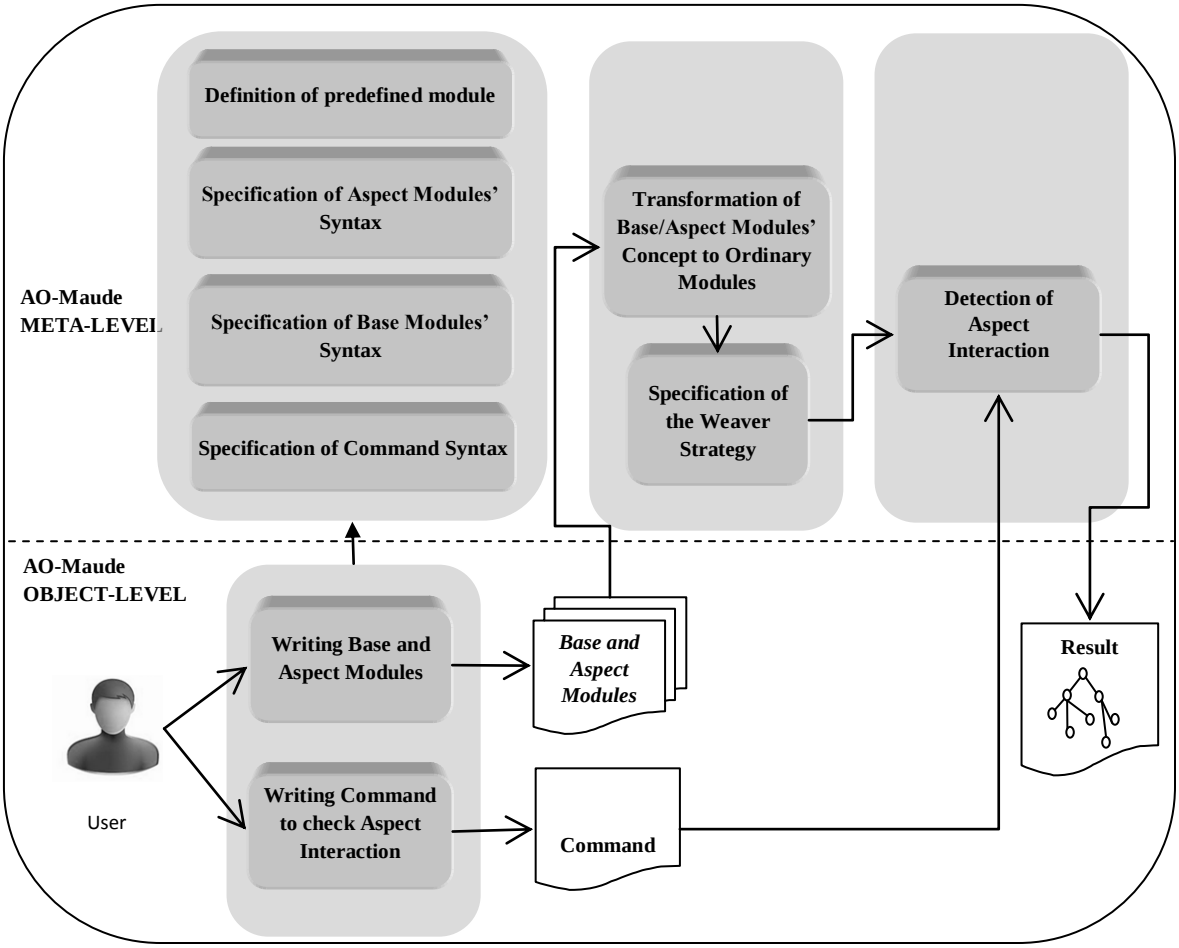

Figure 1: The overall view of AO-Maude

this task because it becomes heavy and tedious to do it especially when the number of aspect (advices) and methods is important.

Consequently, to ensure the detection of the preservation or the violation of the pre/postconditions of method/advice, we agreed to define at the Meta-Level some functions that handle the rules of each method/advice. These functions transform each rule that represents the behavior of each method/advice into two rewriting rules. The first rewriting rule is used in the case where the method/advice preconditions are preserved. In this case the method/advice can be executed with success whereas the second rule corresponds to the case where the method/advice preconditions are violated. As a consequence, the execution of the method/advice leads to an erroneous state.

3.4. Definition of the strategies of the weaver

The aim of this step is to discharge the user from the task of the definition of the weaver mechanism each time. Thus, all the *messy* code of the weaver of [15, 16, 17] will be hidden in the Meta-Level.

When the user specifies the *base* system and *aspect* modules in the first stage, he proceeds, in the second stage, to the step of the composition and the integration of these aspects via the base system. This step is guaranteed via the internal strategies of the defined weaver. In a general way, the different steps of the defined weaver are the following:

- **Detecting the invoked joint point during the execution of the base system**. The aim of this step is to detect among the different base system methods, the method that represents the join point which is

indicated by the different introduced aspects.

- **Collecting the before and after advices that share the detected join point.** To ensure the execution of the *before* and *after* advices (note that only *before* and *after* advices are considered in this paper), we have used a set of functions that collect the *before* and *after* advices in two different lists.
- **Permuting the collected before and after advices.** We have used a set of equations that helps us to get two lists of all possible permutation of the *before* and *after* advices. The idea behind that is to ensure, on one hand, the non-deterministic composition of all order of the conflicting advices and, on the other hand, the detection of the preservation or the violation of the pre/postconditions of each advice in each permutation.
- **Composing and integrating the different advices.** We have used a set of functions to compose and execute each advices permutation. Thanks to the built-in functions, each advice of each permutation is executed in the Meta-Level. We have also used a set of functions to switch between the base system and the composed advices in order to guarantee at the end the integration of the aspect in the base system.

3.5. Definition of the command that ensure the verification of aspect interaction problem

The verification of the composition and the integration of these advices in the base system (which is known as the weaver mechanism) is guaranteed via our defined command that follows this principle:

Principle: Let $adv_1,\ldots,adv_n$ be the advices to be executed on a join point, $pre(adv_i)$ be the precondition of adv_i and $post(adv_i)$ be the postcondition of adv_i . Our proposed command tries to ensure the following points:

***Advice-Advice interaction*.** Our defined command tries to ensure the interaction between advices. The verification of the advices ordering consists of verifying that the postconditions of the advice adv_i should implies the preconditions of the adv_{i+1} as : $post(adv_i) \Rightarrow pre(adv_{i+1})$ for every *i*.

***Base-Advice interaction*.** We aim that the defined command ensures also the interaction between the base system methods and the advices. Thus, the command tries to verify that:

- The postconditions of the base system method (it can be a join point) should implies the preconditions of the first advice adv_1 as : $post(base) \Rightarrow pre(adv_1)$ or ;
- The postconditions of the last advice adv_n should imply the preconditions of the base system method (it can be a join point) as: $post(adv_n) \Rightarrow pre(base)$.

4. CASE STUDY

To illustrate our work, we present an example used in [16] which is a telephony application. Figure 2 shows the Aspect-UML class diagram of this example. The base system, modelled by a set of classes, provides core functionalities to simulate devices and connections. To these basic functionalities, aspects can be added, such as the interrupting callee and the call forwarding features. These two aspects are used to handling busy lines. They crosscut the base system through the pointcut *OpComplete* which concerns the join point *Complete*. Note that this is a typical situation of aspects conflict because two operations will be added before the join point and executed in a given order. Figure 1 shows how both the aspects are added to enrich the base class diagram.

4.1. Representation of the base system in AO-Maude

In the proposed work, the user writes only the *base* and *aspect* modules in the same manner as an ordinary Maude module. We present below a part of *connection* class as:

```
bmod CONNECTION is
   pr CONFIGURATION .
   op Connection : -> Cid .          ---1 ClassName
   op C-Status: C-State ->Attribute. --2 Attributes
   op Origin: Oid -> Attribute .
   op Destination : Oid -> Attribute .
   op Complete : Oid -> Msg .              3 Methods
   ...
   crl :  Complete(C1)                          ---4
      <D1 : Device | D-Status: Waiting >
      <C1 : Connection | C-Status : State >
      <D2 : Device | D-Status : Idle >
      =>
      <D1 : Device | D-Status : Busy >         ---A
      <C1:Connection | C-Status :Connected >
      <D2 : Device | D-Status : Busy >
   if  State == Disconnected.                  ---B
endbm
```

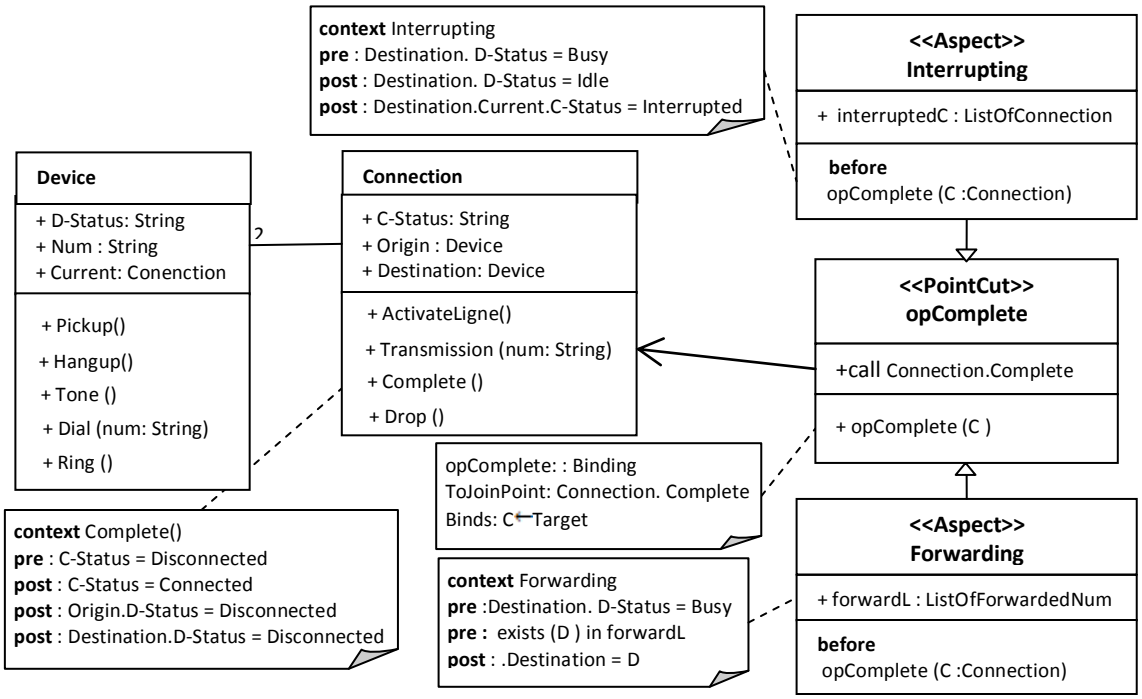

Figure 2: A part of the class diagram of the telephony application

The *connection* class is represented with a base module. This module should import the Configuration Maude module in order to represent the main concepts of object-oriented systems. The name of this class is represented by an operator in mark 1. The attributes of this class are represented with operators of sort Attribute (mark 2). The methods (we take only one method) are also represented by operators as it is shown in mark 3. The behavior of each method is represented by conditional rewriting rule (mark 4). The left hand side of this rule represents the object C1 of class *connection* with the actual C-Status State. The Term *Complete()* means that a message is sent to the object C1 asking for the execution of Complete() method. Whereas, the right hand side of this rule shows the state of the behavior of the object after executing the *Complete()* method. The pre and postconditions of the method are represented respectively by the marks B and A.

4.2. Representation of aspects in AO-Maude

We illustrate in the following a part of the interrupting aspect of the figure 2. This aspect is represented by an aspect module where it should import the Asp&Adv-Configuration AO-Maude module. It defines an operator to represent the name of aspect (as it is shown in mark 2). The attribute of this aspect is represented in the mark 3. The name, the type of this advice and the invoked join point are represented with the term *DefAdv* in mark 4. The specification of the behavior of the advice InterruptAdvice is represented with a rewriting rule. The pre and postconditions of this advice are respectively represented with the condition of the rule and the left hand side of this rule.

```
amod INTERRUPTING is
   pr CONNECTION .
   pr Asp&Adv-Configuration .                    ---1
   op Interrupting : -> Aid .          ---2 AspectName
   Op InterruptedC: List{Oid} -> AspAttribute.---3
   Attribute
   op InterruptAdvice: ->  AdvName .  ---AdviceName
   ...
   crl:
   DefAdv
     <InterruptAdvice,Before,Complete(C1)>    ---4
   <D1 : Device | D-Status : Waiting >
   <C1:Connection | C-Status : Disconnected >
   <D2 : Device | D-Status : State >
   <C2 : Connection | C-Status : Connected >
   <InterruptAdvice: Interrupting |I-Status : Idle>
   =>
   <D1 : Device | D-Status : Waiting >
   <C1:Connection | C-Status : Disconnected >
   <C2 : Connection | C-Status : Interrupted >
   <D2 : Device | D-Status : Idle , >
   <InterruptAdvice : Interrupting|I-Status
                                  :Interrupting >
   if  State == Busy.     ...
endam
```

4.3. Transformation of the base/aspects modules into ordinary Maude modules

Once the user has defined the *base* and *aspect* modules, the AO-Maude transforms each module into an ordinary Maude module (as it is shown in section 3 step(3)). Since each introduced *base* or *aspect* module is similar to the Maude module (i.e. all the different concepts of these modules are kept), the AO-Maude transforms only the behavior of each method/advice (which is represented by one rewriting rule) into two rewriting rules. Note that this transformation is done in the Meta level and with a transparent way to the user. We just present how the *connection* class becomes (in the same manner the different aspects are transformed):

```
mod CONNECTION is ...
crl :  Complete(C1)
   <D1 : Device | D-Status: Waiting >
   <C1 : Connection | C-Status : State >
   <D2 : Device | D-Status : Idle >
   =>
   <D1 : Device | D-Status : Busy >
   <C1:Connection | C-Status :Connected >
   <D2 : Device | D-Status : Busy >
   ResultExecution(Complete(C1),Success)
if  State == Disconnected.

crl :  Complete(C1)
   <D1 : Device | D-Status: Waiting >
   <C1 : Connection | C-Status : State >
   <D2 : Device | D-Status : Idle >
   =>
   <D1 : Device | D-Status : Busy >
   <C1:Connection | C-Status :Connected >
   <D2 : Device | D-Status : Busy >
   ResultExecution(Complete(C1),Error)
if  State =/= Disconnected.      ...
endm
```

Since all the concepts (class, attributes and method name) are presented in the same way as they were defined in the base module, this module presents only the transformation of the method Complete into two rewriting rules. The first rule will be executed when the preconditions of this method are preserved (the preconditions should ensure that State is equal to Disconnected), in this case the method is executed with success. Otherwise, the second rule will be execute and indicates an error execution of method. We have used a term *ResultExecution*(Complete(),Success/Error) to show the successful /failure execution of the method Complete.

4.4. Detection of aspect interaction with the defined command

In this steps, the user proceeds to verify the composition and the integration of the aspects in the base system (the written classes). Remember that our purpose consists of ensuring the composition and the integration of all possible advices order and checking if all pre and postconditions of the advices and methods are preserved. By using our defined command *CheckExecution*, we can verify the composition and the integration of aspects in the base system as:

```
CheckExecution(
        <C1 : Connection | C-Status :  Idle >
        <D1 : Device | D-Status: Idle >
        <D2 : Device | D-Status: Idle >  Pickup()

ASPECTs(
< InterruptAdvice : Interrupting | I-Status : Idle
>
< ForwardAdvice  : Forwarding  | F-Status  : Idle >
))
```

The AO-Maude starts the verification from the initial terms of the *CheckExecution* command. It tests whether all possible orders of advices can be executed with success. AO-Maude finds out two possible solutions (since we have only two advices). In the first solution, we have obtained a failure execution of the *InterruptAdvice*. This situation is due to the following: before executing the join point *Complete()*, the AO-Maude starts the composition of the advices by executing the *InterruptAdvice* as the first advice. This advice interrupts the current connection and changes the status of destination to *Idle*. Once the InterruptAdvice ends, the control flow is passed to the *ForwardAdvice*. At that time a warning message is printed by the fact that the preconditions of this advice are not verified (pre: Destination. D-Status = Busy, see figure 2). Thus, the execution of the *InterruptAdvice* before the *ForwardAdvice* leads to the violation of the preconditions of ForwardAdvice. 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. Thus, the execution of the *ForwardAdvice* should hence be considered first as it was found in the second solution.

5. CONCLUSION

In this paper, we have investigated the aspect interaction problem in general and aspect scheduling in particular. We have presented a new tool for the modeling and the verification of aspect-oriented systems in rewriting logic. In this tool, reflection feature played a decisive role. This tool, which name is AO-Maude, allowed us to define in the Meta-Level the structure of *base*

and *aspect* modules, weaver mechanism and new command that ensures the verification of the composition and the integration of aspects in the base system.

By using the new command in a case study, it is possible to check whether the interaction of aspects affects either their properties or the base system properties.

The current tool can be improved in different ways. The first idea is to extend this tool by integrating the *around* advices and considering more general kind of pointcuts by defining them on aspects. We can also define other commands that helps us to display the search graph generated by the last search.

6. REFERENCES

[1] R. Pawlak, J.P. Retaillé and L. Seinturier, "Programmation orientée aspect pour Java/J2EE", First Edition, Paris, 2004.

[2] S. Dioko Dioko, R. Douence and P. Fradet, "Aspects Preserving Properties", ACM Press, Vol. 3, pp.393-422, 2012.

[3] K. Tian, k. Cooper, K. Zhang and H. Yu, "A Classification of Aspect Composition Problems", IEEE International Conference SSIRI. Shanghai, pp. 101-109, 2009.

[4] S. Krishnamurthi, K. Fisler and M. Greenberg, "Verifying aspect advice modularly", ACM SIGSOFT Symposium on Foundations of Software Engineering, USA, pp.137-146, 2004.

[5] D.X. Xu, O. El-Ariss, W.F. Xu, and L.Z. Wang, "Aspect-Oriented Modeling and Verification with Finite State Machines", Journal of computer science and technology, pp.949-961, 2009.

[6] LTSA Web Site: http://www.doc.ic.ac.uk/ltsa/.

[7] G. Denaro and M. Monga, "An Experience on Verification of Aspect Properties". The International Workshop on Principles of Software Evolution, Austria, 2001.

[8] G.J Holzmann, "The model-checker SPIN", IEEE Transcripts on Software Engineering, pp. 01-17, 1997.

[9] S. Katz and M. Sihman, "Aspect-validation using model-checking", the International Symposium on Verification, Springer, pp.389-411, 2003.

[10] M. Sihman and S. Katz, "Model checking applications of aspects and superimpositions", Foundations of Aspect Oriented Languages, Bonn, Germany, pp.51-60, 2003.

[11] M. Goldman, and S. Katz, "Modular generic verification of LTL properties for aspects". Foundations of Aspect Languages Workshop, Germany, 2006.

[12] R. Douence, P. Fradet, and M. Sudholt, "Composition, reuse and interact analysis of stateful aspects", International Conference AOSD, pp.141–150, 2004.

[13] M. Störzer and J. Krinke, "Interference analysis for AspectJ". Workshops on Foundations of Aspect-Oriented Languages, 2003.

[14] R. Pawlak, L. Duchien and L. Seinturier, "CompAr: Ensuring Safe Around Advice Composition", International Conference on Formal Methods for Open Object-Based Distributed Systems, France, 2005.

[15] F. Mostefaoui , "Un cadre formel pour le développement orienté aspect : modélisation et vérification des interactions dues aux aspects" , PhD thesis, Canada, 2008.

[16] F. Mostefaoui and J. Vashon, "Design-level Detection of Interactions in Aspect UML models using Alloy", Journal of Object Technology, vol.6, pp 137-165, 2007.

[17] A. Boudjedir, T. Benouhiba and D. Meslati, "Verification of aspect composition and integration using rewriting systems", the International Symposium on Modelling and Implementation of Complex Systems, pp.138-148. Algeria , 2010.

[18] N. Dershowitz and J.P. Jouannaud, "Rewrite Systems", Formal Models and Semantics, North-Holland, pp.243-320, 1990.

[19] M.Clavel, F.Durán, S.Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer and C. Talcott. "Maude Manual (Version 2.6)", SRI, 2011.

[20] A.V. Deursen, J. Heering and P. Klint, "Language Prototyping: An Algebraic Specification Approach", W. Scientific, 1996.

[21] K. Futatsugi and R. Diaconescu, "CafeOBJ Report", W.Scientific, 1998.

[22] F. Duran, "A Reflective Module Algebra with Applications to the Maude Language", PhD thesis, University of Malaga, Spain, 1999.

[23] P.C. Olveczky, "Specification and Analysis of Real-Time and Hybrid Systems in Rewriting Logic", Thesis, 2000.

[24] G. Agha, J. Meseguer and K. Sen, "PMaude: Rewrite-based specification language for probabilistic object systems". Workshop on Quantitative Aspects of Programming Languages, 2005.