Formalization of an Aspect-Oriented Modeling Approach

Farida Mostefaoui and Julie Vachon

DIRO, University of Montreal, Quebec, Canada
{mostefaf, vachon}@iro.umontreal.ca

Abstract. Aspect-oriented programming offers special concepts, such as advices and join points, to implement crosscutting concerns which are activated at various points throughout a program, therefore modifying its base behavior. This article presents a high-level aspect-oriented modeling approach and shows how it can be formalized to allow the verification of aspect composition. Models written in Aspect-UML (our UML profile) are translated into COOPN/2 Petri nets, whose corresponding state graphs can be formally verified by model-checking.

1 Introduction

According to aspect-oriented programming (AOP) [1], crosscutting concerns can be added to a base program by introducing advices at specific points. Hence, one of the common criticisms of the aspect paradigm (as discussed in [2]) is the difficulty to reason about aspect interactions once they are woven into the compiled code. This article proposes a high-level modeling approach to carry out the analysis and design of aspect-oriented (AO) software, as well as to engage into formal verification of aspects composing such systems. A UML profile, called Aspect-UML, is proposed to address AO modeling issues. Moreover, Aspect-UML provides formal annotations, such as pre and post conditions, to accurately specify the behavior of sensitive elements such as join points, advices and pointcuts. The COOPN/2 formalism [3] serves as a semantic basis for Aspect-UML models. It thus provides object-oriented Petri nets to describe interactions between objects and aspects composing the system as well as the weaving of advices at join points. Thanks to the formal semantics of COOPN/2, these Petri nets lend themselves well to verification (e.g. deadlock analysis and CTL model-checking). Formalization of Aspect-UML models is the first step towards the verification of correct aspect composition and weaving. Assuming the base system and the aspects are both individually correct, the formal verification process focuses on aspect integration and interaction, rather than on individual local correctness of advices and pointcuts. It therefore aims to reveal important interference problems such as

- **Interference with base program.** Violation of the base program specification induced by woven aspects;
- **Interference with aspect.** Violation of an advice's local specification induced by the base program or some other woven aspect;

– **Interference with system invariants.** Violation of a system invariant following the introduction of new aspects.

Figure 1 shows our verification process into the development flow of our AO approach.

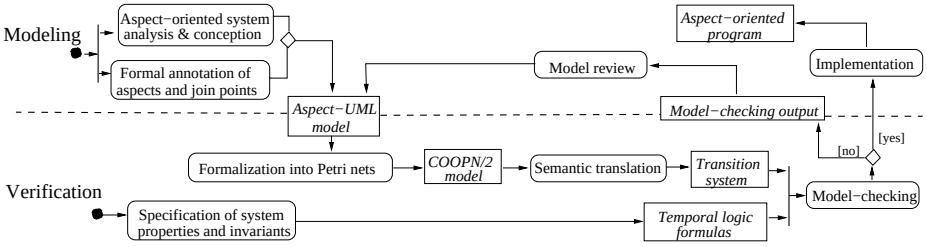

Fig. 1. Aspect-oriented methodology

2 Aspect-UML profile

Despite their strong discipline, current object-oriented development methodologies allow some concerns[1] to span over multiple modules, which results in systems that may be hard to understand, implement, maintain and evolve. Aspect-orientation is a discipline which abstracts and encapsulates crosscutting concerns into separate modules known as *aspects*. An aspect defines a set of crosscutting operations (i.e. *advices*) to be applied dynamically at specific places, called *join points*, in a base system. A set of join points can be gathered under a common interface called *pointcut*. An advice which implements a pointcut interface is declared to execute either `before`, `after` or `around` the join points referenced by this pointcut.

Aspect-UML [4] is the UML profile we have defined to provide modeling extensions taking into account the particular concepts of AOP. Figure 2 shows the Aspect-UML class diagram of a simple telephony application. Two aspects are shown, `Timing` and `Billing`, which represent new functionalities added to the base system to time connections and charge customers for them. Following our AO modeling approach (partially described in [5]), these aspects are introduced via pointcut interfaces `OpComplete`, `OpDrop`, and `OpStop`. Moreover, Aspect-UML provides annotations and constraints for the formal specification of model fragments (such as join points, advices and pointcuts) relevant to the verification of aspect composition.

– Join point [2] and advice specification: it is declarative and given in terms of pre and post conditions. (Note that join points are crosscutted operations on the class diagram while advices correspond to aspect operations.)

[1] often referred to as supplementary requirements.

[2] At the moment, our approach only considers types of join points associated with a method call.

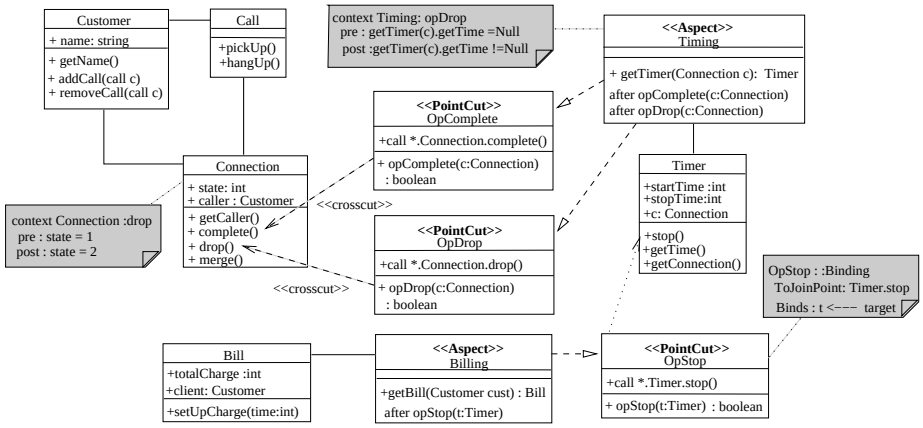

Fig. 2. Class diagram for the Telecom application

– Pointcut specification: it specifies how the evaluation context is passed from join points to related advices.

These Aspect-UML constraints are specified directly on the class diagram using UML notes (cf. rectangles with down-right corner bent over on Figure 2).

3 Formalization of Aspect-UML models in COOPN/2

To be formally verified, Aspect-UML models are translated into the object-oriented Petri net formalism COOPN/2. Figure 3 shows the translation of the Telecom application model (its Aspect-UML class diagram) into COOPN/2. Aspects and related classes are translated into Petri nets class modules (drawn as big ovals) equipped with (1) places (small circles) modeling their attributes, (2) external transitions (black rectangles) representing their public methods and (3) internal transitions (white rectangles) denoting spontaneous private actions. Dotted lines illustrate COOPN/2 transition synchronizations between objects. Thanks to these various features, aspect weaving can be modeled by adding places denoting the execution status (beginning, ending) of join points and by defining appropriate synchronizations with advices. In our example, the weaving of advice `Timing::opComplete()` after join point `Connection::complete` allows to start timing the connection as soon as it is established. It is realized by adding the following elements to the `Connection` object:

– a place named `endComplete`
– an internal transition named `afterComplete` that will spontaneously fire and synchronize with advice `Timing::opComplete()` as soon as place `endComplete` contains a token.

If resources are bounded, a finite state graph can be derived from the COOPN/2 model, thanks to its formal semantics[3]. Aspect composition errors can be detected by looking for unacceptable terminal states, since violation of pre/post

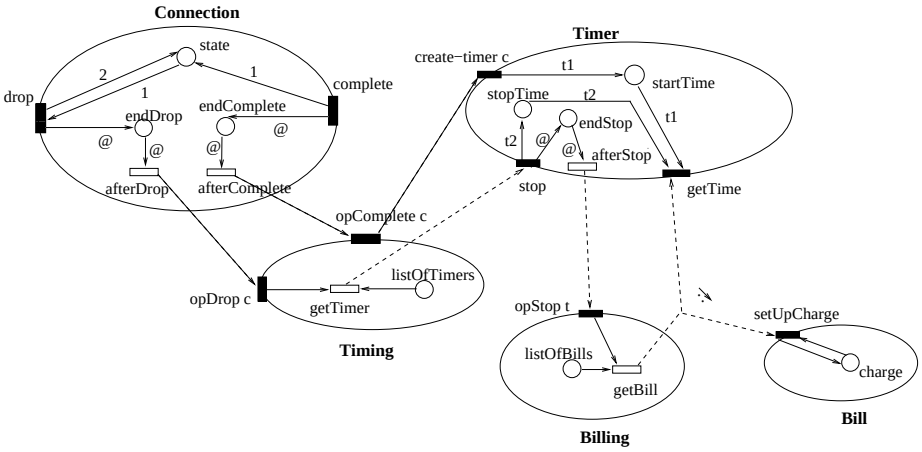

Fig. 3. COOPN/2 formal model of the Telecom application

conditions will induce blocking states in the state graph. Other properties, such as system invariants, can be formulated in temporal logic (e.g. CTL) and be verified by model-checking over the state graph. In the above Telecom example, the verification shall prove aspects `Timing` and `Billing` to be non interfering and correctly used. In other words, the verification aims to guarantee:

- Non-interference between aspects, neither with the base system. Specifications in terms of pre and post conditions (e.g. specification of `Timing:opDrop`) do not violate the specification of the base program at join points (and vice-versa).
- Preservation of system invariants. System invariants of the Telecom application remain true in the final woven system.

4 Conclusions

This paper briefly introduced our modeling and verification framework for aspect-oriented (AO) systems. High-level modeling of AO system can be achieved using our UML profile Aspect-UML. Formalization of Aspect-UML models into COOPN/2 opens the way to simulation and model-checking of significant non-interference properties of AO systems.

To this day, most proposals for AO modeling do not rigorously address the interference problem due to aspects. Among those who do [6, 7], most limit their analysis to the detection of *potential* conflicts obviously revealed by the syntax (e.g. aspects associated to the same join point are identified as potentially conflictual). Inevitably, this type of verification is limited regarding coverage and precision. To increase the accuracy and the significance of the verification process, our approach goes beyond syntax and takes into account the semantic of AO systems. Authors in [8] use model-checking to modularly verify aspect

advices. We also rely on incremental model-checking, but our approach, on the one hand, deals with the formalization of models rather than programs. On the other hand, our solution integrates the semantic transformation of our high-level Aspect-UML models into Petri nets, whereas [8] "*regard[s] this problem as orthogonal to [their] work*" and relies on external tools to generate state machines from Java programs. Contrarily to most methods which are dealing with source code, our approach tackles the problem from a top-down perspective by addressing modeling before system implementation. Moreover, our formalization provides an explicit semantics of AO features and is integrated to an autonomous framework for modeling, model-checking and even simulating AO systems.

Our modeling approach is being fine tuned to allow easy transition from *early* aspects to *design* aspects. Our framework therefore aims to provide use case-like documentation for aspects, model refinement and incremental verification. A model-checker and a deadlock analyzer for COOPN/2 are being developed. Of course, extensive exploration of states increases precision but it also incurs non negligible costs. We thus plan further work on optimizations and abstractions.

Other future directions for this project concern the implementation of an automatic translator from Aspect-UML models to COOPN/2 specifications, as well as the development of a detailed case study underlining the different kinds of interferences caused by aspects in AO systems.

References

1. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: ECOOP'97, LNCS. (1997) 220–242
2. Leavens, G., Clifton, C., eds.: Foundations of Aspect-Oriented Languages Workshop. In Leavens, G., Clifton, C., eds.: AOSD'05. Volume 4. (2005)
3. Biberstein, O.: COOPN/2: An object-oriented formalism for the specification of concurrent systems. PhD thesis, EPFL, Genova Universiy, Switzerland (1997)
4. Mostefaoui, F., Vachon, J.: Approche basée sur les réseaux de Petri pour la vérification de la composition dans les systèmes par aspects. RSTI - L'Objet (12/2006) 157–182
5. Vachon, J., Mostefaoui, F.: Achieving supplementary requirements using aspect-oriented development. In: ICEIS. (2004) 584–587
6. Douence, R., Fradet, P., Sudholt, M.: Composition, reuse and interaction analysis of stateful aspects. In: 3^{rd} Int. Conf. AOSD. (2004) 141–150
7. Weston, N., Taiani, F., Rashid, A.: Modular aspect verification for safer aspect-based evolution. In: RAM-SE Workshop, with ECOOP. (2005)
8. Krishnamurthi, S., Fisler, K., Greenberg, M.: Verifying aspect advice modularly. In: SIGSOFT'04/FSE-12. (2004)