

Towards Executable Aspect-Oriented UML models*

Lidia Fuentes
Dpto. Lenguajes y Ciencias de la Computación
Universidad de Málaga, (Spain)
lff@lcc.uma.es

Pablo Sánchez
Dpto. Lenguajes y Ciencias de la Computación
Universidad de Málaga (Spain)
pablo@lcc.uma.es

ABSTRACT

Aspect-Oriented technologies, including Aspect-Oriented Modeling, introduces a set of new constructions, e.g., advices or pointcuts, that improve the modularization of crosscutting concerns. These new constructions can make it more difficult to understand or visualize how a system works after the different (design) modules are composed together. A straightforward and simple mechanism to observe how a system works is to execute it. UML and its Action Semantics provide the foundations for modeling and executing object-oriented software systems. This paper presents an aspect-oriented extension to the UML and its Action Semantics for the construction and execution of aspect-oriented models. Before executing such aspect-oriented models, they must be weaved. The corresponding model weaver is also presented in this paper.

Categories and Subject Descriptors

D.3.2 [Language Classification]: Design Languages—UML

General Terms

Design, Languages, Verification

Keywords

Aspect-Oriented Modeling, Aspect-Oriented Model Weaver, Aspect-Oriented Design Languages, Aspect-Oriented UML 2.0 Profiles

1. INTRODUCTION

Aspect-Oriented (AO) technologies improve the modularization of software systems and artifacts by means of the definition of: (1) new constructions (e.g., aspects, advices)

*This work has been supported by Spanish MCYT Project TIN2005-09405-C02-01 and European Commission Grant IST-2-004349-NOE AOSD-Europe and the European Commission STREP Project AMPE IST-033710.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Workshop AOM'07 March 12-13, 2007 Vancouver (British Columbia, Canada)

Copyright 2007 ACM 1-59593-658-5/07/03 ...\$5.00.

for suitable encapsulation of crosscutting concerns into single modules; and (2) mechanisms (e.g., pointcuts, weavers) to compose crosscutting concerns with the design modules they crosscut. AO Modeling approaches have mainly concentrated their efforts on defining the set of constructions for encapsulating crosscutting concerns into aspects. As a result, several UML Profiles and design languages [12] have appeared. Nevertheless, an important drawback of most of them is their lack of automatic weaving support. This implies that when software designers need to reason about how the modeled system would work after composing aspects with the design modules they crosscut, they must weave them “manually”, which is really cumbersome. For instance, software designers must check by hand if pointcut models select more or less joinpoints than required.

A straightforward and simple mechanism for visualizing how a system model works when all the design modules are composed together, is to execute it. Inaccuracies inherent in an aspect-oriented design can then be detected during the model execution, before moving on to implementation. Resolving such inaccuracies at design time is cheaper, faster and more desirable than carrying out necessary code modifications later on-the-fly.

In order to execute a software system model, it must contain a complete and precise behavior description. UML and its Action Semantics provides the basis for complete and precise behavior modeling of software systems. Several tools (e.g., Rhapsody, TAU G2 or iUML), conforming to UML and its Action Semantics, have been released in recent years. They are object-oriented, and obviously, they do not incorporate AO support.

This paper presents two complementary contributions: (1) An AO UML 2.0 Profile for complete and precise AO behavior modeling, which extends the UML Action Semantics; (2) a weaving mechanism to automatically add aspects to the design modules they crosscut. The complete system model can then be executed in order to observe how it works and how the modeled aspects behave. The model weaver proposed in this paper is implemented using well-known, widely-used open standards. It is independent of any UML tool. Using these two contributions software designers can execute aspect-oriented models, visualize their behavior and fix errors before moving on to implementation.

An Online Book Store System (OBS), taken from the existing literature, is used to illustrate the concepts presented in this paper. The OBS has been adequately refactored with aspects.

After this introduction, this paper is structured as fol-

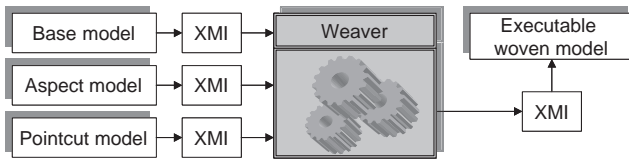


Figure 1: AO Executable Scenario.

lows: Section 2 gives a general overview of the approach. Section 3 presents the Online Book Store System. Section 4 explains the principles for executing UML models. Section 5 describes the UML 2.0 Profile for AO executable modeling. Section 6 contains the description of the model weaver. Section 7 focuses on the current tool support for our approach. Section 8 comments on related work, and finally, Section 9 outlines conclusions and future work.

2. OUR APPROACH

This section contains an overview of our approach. Our goal is to obtain AO models that can be executed. It is also our intention to use well-known and widely used standards whenever possible, in order to obtain vendor-independent solutions and avoid the need for learning new notations and languages.

UML is the most widely known and used software modeling language and there is a wide range of tools available that support it. The execution of UML models is already a reality. UML and its Action Semantics provide the foundations for building object-oriented executable models.

In this paper we define a process for the construction of AO UML executable models. This process relies on the existence of two elements: (1) a UML 2.0 Profile for the specification of AO executable models, which is called AOEM (Aspect-Oriented Executable Modeling); and (2) a model weaver for AO models that conforms to the AOEM profile. Using both elements, such a process is defined (See Figure 1) as follows:

1. First of all, a common UML executable model is constructed for modeling the non-crosscutting concerns. The *base model* is obtained.
2. Crosscutting concerns, including their precise and complete behavior, are modeled as aspects using the AOEM Profile. This produces the *aspect model*.
3. How crosscutting concerns must be composed with the other concerns is specified by means of a *pointcut model*. The rules for modeling pointcuts are also part of the AOEM Profile.
4. The base and aspect models are composed, which produces the *woven model*. This model is a common UML executable model.
5. Finally, to execute the complete AO model, the woven model is imported into a UML tool with executing capabilities (e.g., Rhapsody).

To perform the weaving and to import the woven model into a UML tool with execution capabilities, the UML models must be available in a standard and interoperable format.

This is possible nowadays using the XMI (XML Metadata Interchange) [8] standard. It allows us to serialize a UML model in an XML document, which can then be easily manipulated. The model weaver presented in this paper takes as input the XMI representations of the base, aspect and pointcut models and produces as output an XMI representation of a model of the woven system. The steps of this process are illustrated in the following sections using the OBS example.

3. THE ONLINE BOOK STORE SYSTEM

An Online Book Store (OBS) application, taken from [7], is used as a running example throughout this paper.

The Online Book Store has to provide a way for customers to place orders for books. A customer will start a new order by selecting a book and specifying the required quantity. The customer can continue adding more books to the order. Once the customer is satisfied with his/her selections, the order goes to the check out stage. Each time a new book is added to the order, the order must be persisted. Persistence is easily identified as an aspect of the system.

Our intention is to construct an aspect-oriented executable model of the OBS, which is described in the following sections.

4. EXECUTABLE UML MODELS

This section describes briefly the construction of a UML executable model for the OBS. In order to construct executable models, two basic elements are required: (1) an *actions language*, which contains those elements that abstract the atomic actions the models can carry out; and (2) an *operational semantics*, which specifies where and how the actions can be placed in a model and how a model must be interpreted. Both elements in the UML standard are described below.

4.1 Operational semantics for UML models

The operational semantics of UML is still in the process of standardization [9]. Nevertheless, several tools implementing non-standard operational semantics for UML models already exist. The corresponding tools vendors are leading the creation of the new standard, thus it is reasonable to suppose it will be similar to current versions.

Fortunately, the ideas behind them are quite similar and hopefully the final adopted standard will also be similar and the process of constructing a UML executable model using these tools can be generalized and summarized as follows: Firstly, the global system structure is established as a set of components. Then, the structure of each component is detailed by means of class diagrams. The behavior of each class is specified using a protocol state machine, whose transitions and states may have associated procedures (sets of actions), which model behaviors. Procedures are specified using an action language.

Therefore, the OBS system is firstly decomposed into several components: OBS System, Credit Card, Delivery, etc. We will focus on the OBS System component. The class diagram of Figure 2 details internals of this component. The OBS System contains, among others, at least one Book, some Customer data and a ShoppingCart to store customers orders while they navigate the system. We will focus on this class.

In the next step, the behavior of each class is specified

ReadSelf	Returns a reference to the object where it is executed
CreateLinkObject	Creates and association class between two object ends
WriteStructuralFeature	Write a value in an attribute of an object passed
CallBehavior	Invokes a procedure (another activity diagram with actions)
CallOperation	Invokes an object method
SendSignal	Sends a signal to a target object passed as parameter

Table 1: UML actions used to model the case study

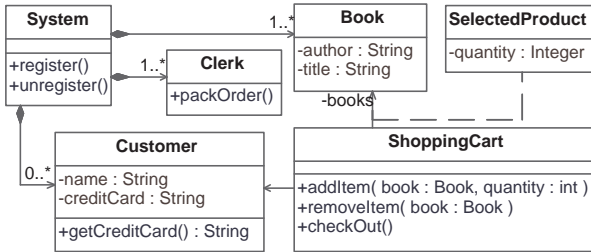


Figure 2: Class diagram for the OBS System internals

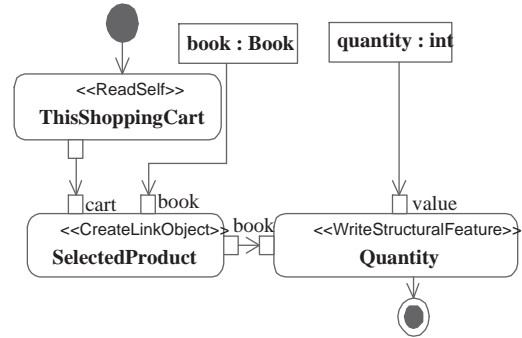


Figure 4: UpdateItems procedure for adding a item

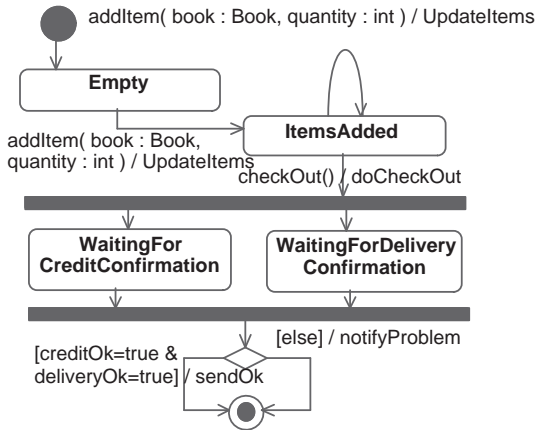


Figure 3: State Machine for the ShoppingCart class

by means of a state machine. Focusing on the ShoppingCart, the following states are identified (See Figure 3): Empty, ItemsAdded, WaitingForDeliveryConfirmation and WaitingForCreditConfirmation. The addItem event will make the system shift from Empty to ItemsAdded, if it is the first selection, or from ItemsAdded to ItemsAdded, if there were some books previously selected. In both cases, the updateItems procedure will be executed.

4.2 The Action Semantics

As commented before, procedures are specified by means of an action language. UML defines its own action language, named *Action Semantics* [10], where an *action* is “the fundamental unit of behavior specification, which takes a set of inputs and converts them into a set of outputs”. Examples of these actions are object creation, calls to methods, writing an attribute value, etc.

Intentionally, the UML action language does not enforce

any notation for drawing actions. Thus, each tool defines its own notation. To avoid the use of notations that works specifically for proprietary tools, we have developed a UML Profile, for specifying sets of actions (i.e. procedures) compatible with any UML tool supporting activity diagrams and abstract actions, which is a common case.

This Profile works as follows: Procedures are represented by means of UML activity diagrams. Actions are nodes of activity diagrams. For each action, we use the general action symbol (a round cornered rectangle). Inputs and outputs are depicted as *pins*. To distinguish each specific action (object creation, attribute reading/writing, etc.), this is stereotyped with its name. Additionally, it must have the same number of input/output pins as specified in the standard, which is ensured by means of OCL constraints.

Figure 4 shows the updateItems procedure modeled according to such Profile. It is executed in the context of a ShoppingCart object after receiving an addItem event. It has two parameters, the selected book and the required quantity. The procedure creates a new association link object of the association class SelectedProduct (see Figure 2), between the ShoppingCart that hosts the behavior (returned by the ReadSelf action), and the selected book. The required quantity is finally written in the corresponding attribute (a structural feature) of the created link object. Before finishing this procedure, the ShoppingCart object must be persisted. This will be specified using aspects in the next section, avoiding the tangling of Persistence with the updateItems concern.

5. ASPECT-ORIENTED MODELING

In order to obtain a complete model of the selected use case, persistence must be added. This section illustrates how it is modeled in an aspect-oriented fashion. To support the construction of aspect-oriented executable models, this paper introduces the AOEM (Aspect-Oriented Executable

GetMessName	Returns the name of the intercepted message
GetMessArg(n)	Returns the n -argument of the intercepted message
GetTarget	Returns a reference to the target of the intercepted message
GetSource	Returns a reference to the source of the intercepted message
Proceed	Executes the intercepted behavior

Table 2: Aspect-oriented actions

Modeling) UML 2.0 Profile, which is integrated with the principles of Executable UML and its action language.

In agreement with Fuentes and Sánchez [6], the AOEM Profile is specified in three steps: (1) Definition of the joinpoint model; (2) Definition of the modeling of aspects and their associated elements, like advices; and (2) the rules (e.g. pointcuts) that compose these aspects with the rest of the application.

5.1 Joinpoint model

The AOEM Profile uses a non-invasive joinpoint model, similar to JAsCo [15], Lasagne [16] or CAM/DAOP [11] models, which is suitable to be used with black-box software modules, such as third-party components or legacy systems.

The AOEM joinpoint model only allows designers to intercept observable behaviors of the design modules: (1) object creation and destruction; (2) the sending and receiving of a message/method; (3) the throwing of an event; and (4) the raising of an exception. Aspect methods can be executed *before*, *after* or *around* (in substitution of) these joinpoints. This paper will focus on the joinpoints related to the sending and receiving of a message, as the other cases, at modeling level, can be considered special kinds of the message (call or execution) joinpoints.

5.2 Aspect modeling

An *aspect* is modeled as a common class with special operations representing *advices*. Advices differ from common operations in that they are never invoked explicitly and they are executed by the AO weaver without the knowledge of the base class designer. For this reason, advices do not have parameters. Consequently, each aspect-orientated language has to provide mechanisms to allow advices to retrieve the information related to the joinpoint (e.g., the arguments of a message) they might need. A subset of the actions provided by the AOEM Profile to access the joinpoint context is shown in Table 2.

An advice is modeled as a common procedure (activity diagram) without input objects. It could have output objects that would modify message arguments or return values. In the case of *around* messages, the number and kind of output objects of the *around* message should be identical to the number of return values of the intercepted message.

Figure 5 shows the design of persistence as an aspect. A class `Persistence`, stereotyped as `<<aspect>>`, is created. An advice `persist` is added to this class (Figure 5.a). This aspect is associated with a `Persistent` class, which represents a traditional object-oriented class with methods for persisting objects. The `persist` advice (Figure 5.b) is able to intercept the execution of any method that changes the state of the object which contains it. Hence, the `persist` advice gets a reference to the object that host the method (`GetTarget` action), and invokes the `persist` operation of the traditional `Persistent` object, with that reference as argument.

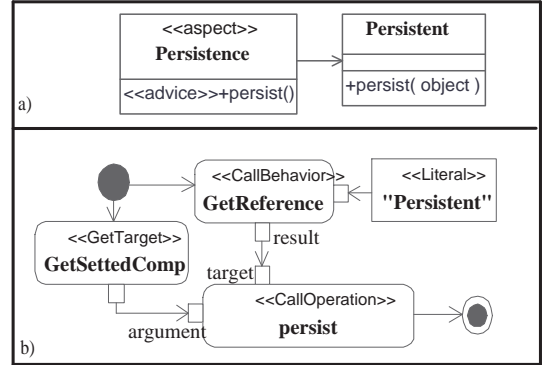


Figure 5: Persistence aspect: a) structure definition b) advice behavior

5.3 Pointcut modeling

Finally, we need to specify the pointcut that specifies how to compose `Persistence` and the `ShoppingCart`.

At the modeling level, the common practice for specifying pointcuts is, roughly, to use UML diagrams with wildcards (e.g., “*” to represent any sequence of characters or “?” to represent any sequence of arguments) [6, 13]. As our interest is to intercept interactions between objects (message sending/receiving), sequence diagrams are selected to model pointcuts because they are the main element in UML representing object interactions and they offer a user-friendly widely known notation.

A pointcut, according to the AOEM Profile, is expressed by means of a sequence diagram, stereotyped as `<<pointcut>>`. This stereotype has a tagged value called `advice`, which represents an ordered collection of aspect advices to be executed in the order as specified in the joinpoints that the pointcut selects.

The specific message of the sequence diagram that must be intercepted is stereotyped as `<<joinpoint>>`. This stereotype has two tagged values: (1) `point`, it indicates if the interception point is either the sending (SEND) of the reception (RECEIVE) of the message; and (2) `time`: that specifies when the advice is executed related to the joinpoint (BEFORE, AFTER, AROUND). Wildcards are available in class and method names: “*” represents any sequence of characters and “.” any sequence of arguments.

The pointcut for adding `Persistence` to the `ShoppingCart` class is shown in Figure 6. It specifies that the `persist` advice must be executed after the reception (i.e., after the method execution) of any message starting with “add” and with any

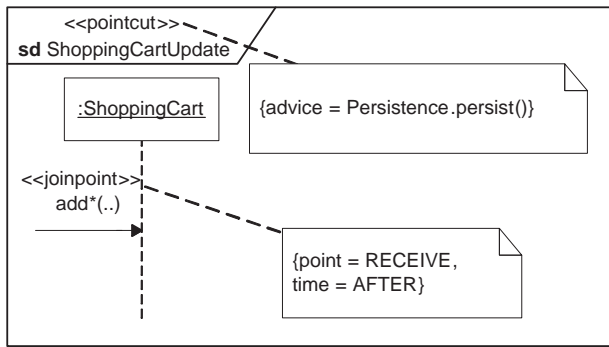


Figure 6: Pointcut for persistence

number of arguments. This message can come from any source, as it is not specified.

More powerful and complex pointcuts than those shown in Figure 6 can be specified using the AOEM Profile. For instance, cflow-like constraints can be specified placing messages above the `<<message>>`. The `<<pointcut>>` stereotype has a tagged value named `withincode` that serves to express `withincode` conditions. A complete description of these features is beyond the scope of this paper.

6. WEAVING EXECUTABLE MODELS

In previous sections, the selected use case of the OBS example has been fully modeled in an aspect-oriented fashion. However, before executing the OBS model, we need to weave it. As the final behavior of base classes and aspects is expressed by means of activity diagrams, the weaving problem can be reduced to weaving activity diagrams.

The task of the model weaver is to inject the advice behaviors into the places indicated by the pointcut specifications. It is achieved in two phases: joinpoint selection and aspect injection.

First of all, the *joinpoint selector* searches all the joinpoints that the pointcuts select. Additionally, the joinpoint selector adds to each selected joinpoint the name of the advice that needs to be injected and the time (BEFORE, AFTER, AROUND) for advice execution.

As the model weaver processes the XMI (an XML-based serialization) representation of the base model, it allows us to use XPath expressions in order to search the selected joinpoints. XPath [17] expressions specify patterns that match several XML tags inside an XML document. Thus, each pointcut is translated to XPath expressions, which selects all the XML tags representing selected joinpoints.

For instance, the pointcut of Figure 6 specifies that the `persist` advice has to be executed after the execution of any method of the `ShoppingCart` class whose name starts with `add` and with any number and kind of arguments. The joinpoint selector has to: (1) look for all the activities, hosted by the `ShoppingCart` class, that represents the body of a method with such signature; (2) mark them as `<<selected joinpoint>>`; and (3) add `advice=Persistence.persist` and `time=AFTER` as tagged values of the stereotype.

The second step is to inject the corresponding aspect advices into the selected joinpoints. Depending on the exact

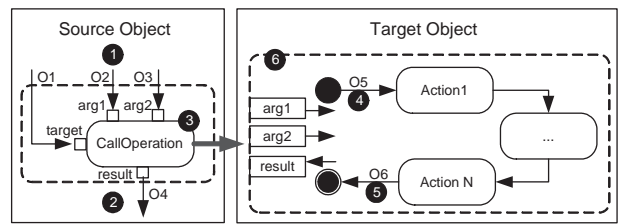


Figure 7: Advice injection points

	BEFORE	AFTER	AROUND
SEND	1	2	3
RECEIVE	4	5	6

Table 3: Correspondence of injection points

joinpoint where an advice has to be injected (i.e., BEFORE or AFTER a the sending of a message), and the specific time (i.e. BEFORE, AFTER or AROUND) where it has to be executed, it can be injected in six different points, as illustrated in Figure 7. Each point corresponds to a specific value of the pair (joinpoint kind, execution time), as shown in Table 3. For instance, if the aspect has to be executed BEFORE SEND, it is injected between the call action and the actions that precede it (Figure 7, label 1). If it has to be injected AFTER RECEIVE, as in our example, it is injected at the end of the intercepted activity (Figure 7, label 5).

Aspect advices are injected as structured activities (Figure 8 (gray background)) inside the procedures they crosscut. They contain the same behavior as the advices, but the aspect-oriented actions introduced by the AOEM Profile are appropriately translated to common UML actions. The injection and the translation of the aspect-oriented actions require updating the object and control flows of the original activity diagram in order to ensure the correctness of the composition.

For example, if an aspect is inserted BEFORE SEND a call action, the original objects flows that feed the inputs of the call will probably need to be redirected to feed the inputs of the structured activity representing the advice. As the advice might modify these object values, new object flows will have to be created between the output values of the advice and the inputs of the original call. More clearly, we need to redirect the arguments of the method call to the advice, and the outputs of the advice will serve as arguments for the method call because the advice will probably modify these values.

These concepts are specifically illustrated for the injection of the `persist` advice after the execution of `updateItems` procedure of the `ShoppingCart` class (Figure 8). The advice is added just before the final node of the activity specifying the `updateItems` procedure (an AFTER RECEIVE case, Figure 7, label 5). The original control flow from the `WriteQuantity` node to the final node (Figure 4) is removed, and the new control flows C1 and C2 are added. The `GetTarget` action of the advice is replaced by a `ReadSelfAction` as the advice is injected in the target object.

We would like to point out that the translation of aspect-

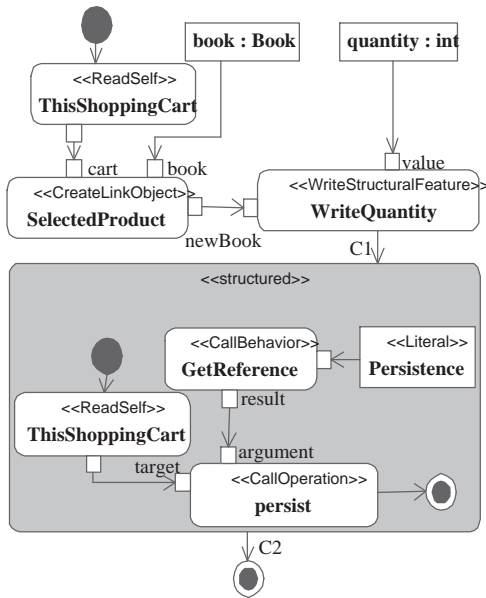


Figure 8: persist woven into ShoppingCart.addItem

oriented actions, the advice injection and the updating of object/control flows involve many special cases with many low level details which are not commented here for the sake of brevity and simplicity.

When more than one aspect need to be injected, several pointcuts can have a shared subset of selected joinpoints. In this case, the execution ordering of advices might be important to ensure the correctness of the application [5]. To solve this issue, this approach implements a simple mechanism: each advice has an integer assigned that is unique for all the advices in the model. This integer represents the execution priority of the advice. If two aspects are applied over the same joinpoint, they are executed from the higher (smallest integer) to the lower (biggest integer) priority.

7. CURRENT TOOL SUPPORT AND VALIDATION

This section describes the experiments carried out to validate our approach and the tools required to reproduce our experiments.

When selecting tools, the best choice would be to find one that supports UML executable modeling and full XMI import/export capabilities. Unfortunately, at the present time there is no tool in existence which satisfies both requirements at the same time. Thus, to validate our results, we had to use a chain of tools, each one meeting our requirements partially.

UML modeling was done using the UML2 plugin for Eclipse¹. It is the most complete implementation of the UML 2.0 metamodel, including the whole UML Action language, and it offers full XMI export/import capabilities. MagicDraw² was used to elaborate the graphical diagrams of this paper.

¹<http://www.eclipse.org/uml2/>

²<http://www.magicdraw.com/>

In general, any tool supporting UML Action Semantics, or activity diagrams plus the abstract actions, and with full XMI exporting capabilities would work.

In a second phase, we need to perform the weaving. The static weaver takes the XMI of the base, aspect and pointcut models as input and it produces the woven model. The implementation of the static weaver is tedious but simple, since we only have to manipulate XMI files (XML trees) following the rules of section 6. This can be implemented in any language with XML facilities, like Java plus DOM.

Finally, the woven model is executed. There is no simulation tool that supports full XMI importing capabilities. Therefore, the solution adopted was to import the model “manually” in a simulation tool, in our case Rhapsody, and to execute the model.

It is evident that there is a clear lack of effective and seamless tool support for our approach. Consequently we are now developing a UML execution engine, which will be able to execute the UML Action language from its XMI 2.0 representation. Meanwhile, the work presented in this paper can be reproduced following the steps and the chain of tools described above. We also hope this lack of tool support will be solved when the tool vendors start to adopt fully the UML and XMI 2.0 standards.

8. RELATED WORK

There is some preliminary work about aspect-orientation and executable models in the literature. It is described in this section and the drawbacks are identified.

Sunyé et al. [14] presented a framework for modeling aspect-oriented applications. It serves to construct aspect-oriented executable models, but the weaving is postponed until the implementation phase and thus, the execution of the complete model, including aspects, is not possible at modeling time.

Theme/UML [1] is an extension of UML for aspect-oriented modeling. Theme/UML supports all the UML 2.0 diagrams. Therefore, using Theme/UML we should be able to weave UML executable models, which specify procedures using the Action Semantics. However, although Theme/UML specifies the weaving semantics of the approach, until now the weaving must be done manually, since no tool support is available. We tried to implement a Theme/UML weaver, but without fruitful results since it is quite complex and it is not precisely defined beyond sequence and class diagrams.

Cottenier et al. [2, 4, 3] present an idea very similar to this paper, but based on the Telelogic TAU G2 implementation of the Executable UML principles. They defined an aspect-oriented Profile that extends the Telelogic SDL metamodel for the Action Semantics. This notation is not compatible at all with the current UML Action language and introduces some proprietary features that obviously can not be exported adequately to XMI. The aspect-oriented model weaver is implemented as a Telelogic add-in [4], therefore it is not portable and tool-independent. Additionally, the weaving process is not clearly described in their work.

9. CONCLUSIONS AND FUTURE WORK

This paper has described how to construct and execute aspect-oriented models. Using aspect-orientation at modeling level, software designers can avoid the well-known problems coming from scattered and tangled representations of

crosscutting concerns. Several approaches have appeared in recent years to provide modeling support for AOSD [12], but AO developers have not had any support until now to check the correctness of complete (woven) models. With our approach, an AO model can now be easily tested by means of its execution.

In order to support aspect-oriented executable modeling, this paper has presented a UML 2.0 Profile, called AOEM, for precise behavior modeling of aspects. Then, a model weaver for such Profile, with a feasible implementation and that uses well-known standards was described. The model weaver produces a common UML executable model as output that can run in any UML modeling tool with execution capabilities.

As the woven model is a common UML executable model, 100% of code can be automatically generated if desired (this feature is currently supported by several tools, like Telelogic Rhapsody and Telelogic TAU G2). As the generated code is non aspect-oriented, it allows development teams to use aspect-orientated models on any target language supported by their code generators.

As future work, it is our intention to incorporate more aspect-oriented features to the AOEM Profile. In this sense, we will substitute the current pointcut specification by Joint-point Designation Diagrams (JPDD's) [13] in order to provide more expressive pointcuts. It was not done in this paper as it is not trivial. First of all, as JPDD's are not UML compliant currently, a UML Profile should be derived first. Secondly, to generate XPath expressions is not so simple as for our current pointcuts. Thus, we opted for using a simpler, but powerful enough, pointcut metamodel following the JPDD philosophy for the first version of the AOEM Profile and the associated model weaver. Experienced readers would miss inter-type declarations in the AOEM Profile. They were explicitly left out as authors consider they are not strictly required for aspect-orientation [6]. Nevertheless, they will be added to the AOEM Profile in future versions. More flexible ways for accessing to the joinpoint context will also be investigated.

10. REFERENCES

- [1] S. Clarke and E. Baniassad. *Aspect-Oriented Analysis and Design : The Theme Approach*. Addison-Wesley Professional, March 2005.
- [2] T. Cottenier, A. V. de Berg, and T. Elrad. Modeling aspect oriented compositions. In J.-M. Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *LNCS*, pages 100–109, Montego Bay (Jamaica), October 2005.
- [3] T. Cottenier, A. van den Berg, and T. Elrad. Motorola weavr: Model weaving in a large industrial context. In *Proc. of the 6th Int. Conference on Aspect-Oriented Software Development, Industry Track (AOSD)*, address = Vancouver (British Columbia, Canada), month = March, year = 2007, note = Available at <http://www.iit.edu/~concur/weavr/papers/>.
- [4] T. Cottenier, A. van den Berg, and T. Elrad. Model weaving: Bridging the divide between elaborationists and translationists. In *Proc. of 9th Int. Workshop on Aspect-Oriented Modeling (AOM), 9th Int. Conf. on Model Driven Engineering, Languages and Systems (MODELS)*, Genova (Italy), October 2006.
- [5] R. Douence, P. Fradet, and M. Südholt. A Framework for the Detection and Resolution of Aspect Interactions. In D. S. Batory, C. Consel, and W. Taha, editors, *1st Int. Conf. on Generative Programming and Component Engineering (GPCE)*, volume 2487 of *LNCS*, pages 173–188, Pittsburgh, (Pennsylvania, USA), October 2002. Springer.
- [6] L. Fuentes and P. Sánchez. Elaborating UML 2.0 Profiles for AO Design. In *8th Workshop on Aspect-Oriented Modeling (AOM), 5th Int. Conf. on Aspect-Oriented Software Development (AOSD)*, Bonn (Germany), March 2006.
- [7] S. Mellor and M. Balcer. *Executable UML: A Foundation for Model Driven Architecture*. Addison-Wesley Professional, March 2002.
- [8] Object Management Group (OMG). MOF 2.0/XMI Mapping Specification, v2.1 (formal/05-09-01), 2005. <http://www.omg.org/cgi-bin/apps/doc?formal/05-09-01.pdf>.
- [9] Object Management Group (OMG). Semantics of a Foundational Subset for Executable UML Models Request For Proposal (ad/2005-04-02), April 2005. <http://www.omg.org/docs/ad/05-04-02.pdf>.
- [10] Object Management Group (OMG). Unified Modeling Language: Superstructure v2.0 (formal/05-07-04). Chapter 5: Actions, 2005. <http://www.omg.org/docs/formal/07-02-03.pdf>.
- [11] M. Pinto, L. Fuentes, and J. M. Troya. A Dynamic Component and Aspect-Oriented Platform. *The Computer Journal*, 48(4):401–420, March 2005.
- [12] R. Chitchyan et al. Report synthesizing state-of-the-art in aspect-oriented requirements engineering, architectures and design. AOSD-Europe Deliverable D11, AOSD-Europe-ULANC-9, May 2005. <http://www.aosd-europe.net/deliverables/d11.pdf>.
- [13] D. Stein, S. Hanenberg, and R. Unland. Expressing different conceptual models of join point selections in aspect-oriented design. In *Proc. of the 5th Int. Conf. on Aspect-Oriented Software Development (AOSD)*, Bonn (Germany), 2006. ACM Press.
- [14] G. Sunyé, F. Pennaneac'h, W.-M. Ho, A. L. Guennec, and J.-M. Jézéquel. Using uml action semantics for executable modeling and beyond. In K. R. Dittrich, A. Geppert, and M. C. Norrie, editors, *Proc. of the 13th Int. Conf. on Advanced Information Systems Engineering (CAiSE)*, volume 2068 of *LNCS*, pages 433–447, Interlaken (Switzerland), June 2001. Springer.
- [15] D. Suvéé, W. Vanderperren, and V. Jonckers. JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development. In *3rd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 21–29, Boston, (Massachusetts, USA), 2003. ACM Press.
- [16] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B. N. Jørgensen. Dynamic and selective combination of extensions in component-based applications. In *23rd International Conference on Software Engineering (ICSE)*, pages 233–242, Toronto (Canada), May 2001. IEEE Computer Society Press.
- [17] World Wide Web Consortium (W3C). XML Path Language (XPath) Version 1.0, 1999.