JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2004

Vol. 6, No. 7, Special Issue: Aspect-Oriented Modeling, August 2007

Designing and Weaving Aspect-Oriented Executable UML models

Lidia Fuentes, Dpto. Lenguajes y Ciencias de la Computación, University of Málaga, Spain
Pablo Sánchez, Dpto. Lenguajes y Ciencias de la Computación, University of Málaga, Spain

Aspect-Oriented technologies, including Aspect-Oriented Modelling, provide a set of new constructs (e.g., advices or pointcuts), that help to improve the modularisation of crosscutting concerns. However, these new constructions can make it more difficult to understand how a system works as a whole, once all design modules are composed together, because: (1) designers may not be familiar with the new aspect-oriented constructions; and/or (2) aspect-orientation may cause new problems, such as the handling of aspect interactions. A straightforward and simple solution to check how a system works is to execute it. UML and its Action Semantics provide the foundations for modelling and executing object-oriented software systems. This paper presents a UML 2.0 Profile which extends the UML and its Action Semantics for the construction of aspect-oriented executable models and also a model weaver which makes the execution of such models possible. Our approach is illustrated using an Online Book Store system taken from the literature.

1 INTRODUCTION

Aspect-Oriented technologies improve the modularisation of software systems and artifacts by defining: (1) new constructions (e.g., aspects, advices) for the 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. Aspect-Oriented Modelling approaches have mainly concentrated their efforts on defining the set of constructions that make possible the appropriate encapsulation and composition of crosscutting concerns. As a result, several UML Profiles and design languages [27] have appeared.

Nevertheless, an important drawback of most of them, from the authors' point of view, is their lack of automatic tool support [14]. To the best of our knowledge, there are no tools, such as AJDT (AspectJ Development Tools) [6], that help aspect-oriented designers to reason about their aspect-oriented models by providing, for instance, crosscutting maps (as AJDT does). This implies that when software designers need to reason about how the modelled system would work as a whole, i.e., after composing aspects with the design modules they crosscut, software designers must weave them "manually" and/or "mentally", which is a very cumbersome and error prone task. Therefore, software designers must check by hand if pointcut

Cite this article as follows: Lidia Fuentes and Pablo Sánchez: *Designing and Weaving Aspect-Oriented Executable UML models*, in Journal of Object Technology, vol. 6, no. 7, Special Issue: Aspect-Oriented Modeling, August 2007, pages 109–136, http://www.jot.fm/issues/issues_2007_8/article5

models select more or less joinpoints than required, if the data flow through several aspects is correct, or how an aspect behaves in different situations. Further discussions about the behaviour of a system in the presence of aspects can be found in Clifton and Leavens [5] and Kiczales and Mezini [16].

A straightforward and simple mechanism for visualising how a system model works when all the design modules are composed together, is to execute it and observe its behaviour, i.e., to simulate the model. Model executability is a prerequisite for model simulation, a widely used technique in most engineering disciplines. Simulation allows us to obtain early prototypes of our systems which can be used to (1) analyse and test the behaviour of such systems against a set of requirements; (2) get feedback from stakeholders before system implementation. Inaccuracies inherent in an aspect-oriented design can then be detected during the model simulation, before moving on to implementation. Fixing such inaccuracies at design time is cheaper, faster and more desirable than carrying out necessary code modifications later on-the-fly. Further discussion on the benefits of simulation in software development can be found in Cottenier et al [10], Doldi [11] and in the Saturn experience [19].

In order to make a software system model executable, a prerequisite for model simulation, this model must contain a complete and precise behaviour description. UML and its Action Semantics provides the basis for complete and precise behaviour modelling of software systems. Several tools conforming to UML and its Action Semantics and able to execute/simulate UML models, have been released in recent years (e.g., Rhapsody, TAU G2, iUML, Rational Rose RT or IAR UML VisualSTATE). UML and its Action Semantics, and therefore these tools, are object-oriented, and consequently, they do not incorporate aspect-oriented support.

In order to overcome this shortcoming, this paper presents two complementary contributions: (1) an AO UML 2.0 Profile for complete and precise AO behaviour modelling, which extends the UML Action Semantics; (2) a weaving mechanism to automatically compose aspects with the design modules they crosscut. The complete system model can then be executed, which is the basis for simulating it. The model weaver proposed in this paper is implemented using well-known, widely-used open standards and it is independent of any specific UML tool. Using these two contributions software designers can execute aspect-oriented models, visualise their behaviour, reason more easily about them, analyse different alternative solutions and/or fix errors before moving on to implementation.

The solution presented in this paper also benefits the executable modelling community, which can now use aspect-orientation, with the well-known benefits regarding ease of development, maintenance and evolution, as well as reusability of individual design modules.

An Online Book Store System, taken from the existing literature [20], is used to illustrate the concepts presented in this paper. It has been adequately refactored with aspects.

In the following, the paper is structured as follows: Section 2 gives a general

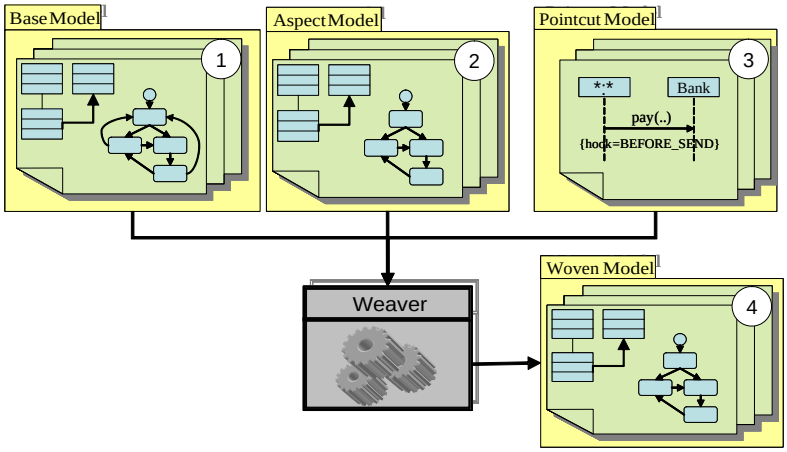


Figure 1: AO Executable Scenario

overview of the approach. Section 3 presents the Online Book Store System, used as an example throughout this paper. Section 4 explains the principles for executing UML models. Section 5 describes the UML 2.0 Profile for Aspect-Oriented executable modelling. 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. Section 9 provides some reflections on our approach, and finally, Section 11 outlines conclusions and future work.

2 OUR APPROACH

This section contains an overview of our approach. Our goal is to obtain aspect-oriented 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 modelling language and there is already a wide range of tools available to support it. UML and its Action Semantics provide the foundations for building object-oriented executable models, thus, the execution and simulation of UML models are already a reality.

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

1. First, a common UML executable model is constructed to model the non-crosscutting concerns. The *base model* is obtained.

2. Crosscutting concerns, including their precise and complete behaviour, are modelled as aspects using the AOEM Profile. This produces the *aspect model*.

3. How crosscutting concerns must be composed with the concerns they crosscut is specified by means of a *pointcut model*. The rules for modelling 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 aspect-oriented model, the woven model is imported into a UML tool with executing capabilities (e.g., Rhapsody). Thus, we can run/simulate our model.

To perform the weaving, the base, the aspect and the pointcut model need to be exported to a standard and interoperable format that can be manipulated. The woven model must be generated according to this standard format, in order to ensure it can be imported into a UML tool with execution capabilities. Such a format is provided by the XMI (XML Metadata Interchange) [22] standard. It allows us to serialise 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. Obviously, the model weaver can be constructed using APIs or model transformation languages that help us to deal with the XMI representation of a model.

The steps of this process are illustrated in the following sections using the Online Book Store example.

3 THE ONLINE BOOK STORE SYSTEM

An Online Book Store System, taken from the executable modelling literature [20], is used as an example to illustrate our approach. The Online Book Store has to provide a way for customers to place orders for books. From the set of use cases presented in Mellor and Balcer [20], we focus in this paper on the ordering of books, which is specified as follows:

1. A customer starts a new order by selecting a book and the required quantity.

2. The customer can continue adding more books to the order.

3. Once the customer is satisfied with his/her selections, the order goes to the check out stage. A message is sent to the credit card company to process the payment.

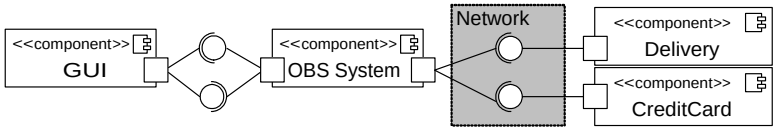


Figure 2: Excerpt of the OnLine Book Store System architecture

4. If the payment is approved, a shipping order is created. A message is sent to the delivery company to inform them that a new order is ready.

Additionally, some global requirements must be observed: (1) each time an order changes, it must be persisted; (2) all the purchases will be made in Euros; (3) because of business alliances, the system must interact with a specific credit card company, which carries out all the operations in US Dollars. Therefore, when exchanging messages with the credit card company, the system needs to perform currency conversions. Persistence and Currency Conversion are identified as crosscutting concerns of the system.

Figure 2 shows an excerpt of the Online Book Store System architecture, which is comprised of several components: the OBS System component, responsible for the OBS core functionality; The Graphical User Interface GUI; and the external Credit Card and Delivery services. Communication between the OBS System and the external services is performed through a public network. For privacy reasons, the CreditCard service imposes as an additional requirement that all the requests received must be encrypted. Encryption is also identified as a crosscutting concern of the system.

Our intention is to construct an aspect-oriented executable model of the Online Book Store System, where Persistence, Encryption and Currency Conversion are well-modularised as aspects, without hampering system development, maintenance, evolution or decreasing the reusability of the individual design modules.

4 EXECUTABLE UML MODELS

This section briefly describes the construction of a UML executable model for the Online Book Store, which is the first step in our approach. In order to construct executable models, two basic elements are required: (1) an *action 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.

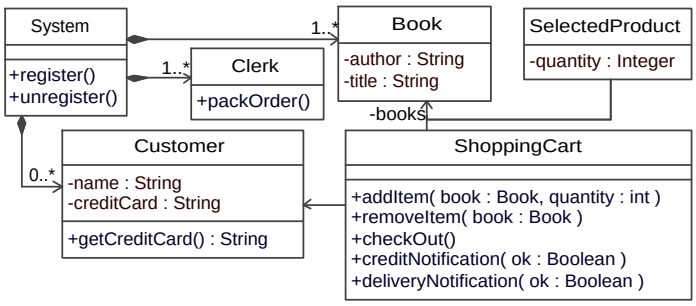


Figure 3: Class diagram for the OBS System internals

Operational semantics for UML models

The operational semantics of UML is still in the process of standardisation [24]. Nevertheless, several tools implementing non-standard operational semantics for UML models already exist (e.g., Nucleus Bridge Point, iUML, IAR UML VisualSTATE, Rational Rose RT, Rhapsody or Tau G2).

Fortunately, the ideas behind them are quite similar, and as the corresponding tool vendors are leading the creation of the new standard, it is reasonable to suppose that the final adopted standard will be similar to current versions. The process of constructing a UML executable model using these tools can be generalised and summarised 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 behaviour of each class is specified using a state machine, where each state represents a stage in the lifecycle of a typical instance of the class. A transition rule specifies the new state achieved when an object in a given state receives a particular event. Each event represents an incident during the object lifecycle, as the reception of a method call, a signal, or the expiration of a timer. Transitions and states may have associated procedures (sets of actions) that model the behaviour executed when a class instance enters, stays in or exits a state. Procedures are specified using an action language.

The Online Book Store system is firstly broken down into several components: OBS System, Credit Card and Delivery (see Figure 2). We will focus on the OBS System component. The class diagram of Figure 3 details the internal structure of this component. Basically, the OBS System contains a System class to register/deregister users and to start the application. The system must have at least one Book, some Customer data and Clerks to pack the orders. A ShoppingCart is used to store customer orders while they navigate the system. We will focus on the ShoppingCart class.

In the next step, the behaviour of each class is specified by means of a state machine. Figure 4 shows the state machine that models the lifecycle of the ShoppingCart class. Initially, the shopping cart is empty, until an event for adding a book arrives.

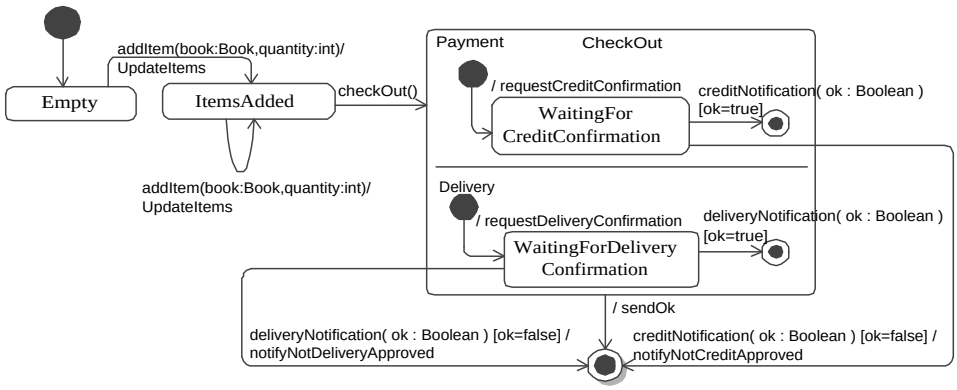


Figure 4: State Machine for the ShoppingCart class

In this case, it changes to the ItemsAdded state, where more events for adding a book can be received. Each time an addItem event is received, the UpdateItem procedure is executed. When the customer performs a check out, the ItemsAdded state is left, and the ShoppingCart enters into two states concurrently: Payment and Delivery, where messages for credit (requestCreditConfirmation procedure) and delivery (requestDeliveryConfirmation procedure) confirmation, respectively, are sent. If credit and delivery are successfully confirmed, each state finishes appropriately, the system leaves the concurrent state and a confirmation message is sent to the customer. If the credit or the delivery are not confirmed, the concurrent states are interrupted and the customer is informed about the error.

The Action Semantics

As commented before, procedures are specified by means of an action language. UML defines its own action language [25], which aims to provide modelers with the basis for a complete specification of UML models, including their full behaviour, which is specified using a set of platform-independent atomic actions. It allows for the execution of the UML models, and even to generate 100% of the code if desired [21].

The UML standard defines an *action* as "the fundamental unit of behaviour specification, which takes a set of inputs and converts them into a set of outputs". The UML action language defines operations that support the manipulation of objects and the logical constructs for the specification of algorithms. Examples of these actions are object creation, calls to methods or writing an attribute value, among others. Actions are contained in behaviours (procedures), which supply the context for them. The specific set of actions used in this paper are explained in Table 1.

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

ReadSelf	Returns a reference to the object where it is executed
CreateLinkObject	Creates an association class between two object ends
AddStructuralFeature	Add a value to 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

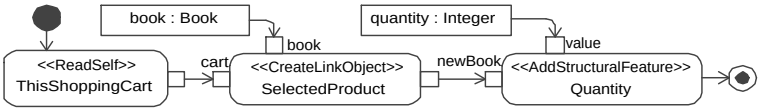


Figure 5: updateItems procedure for adding an item

that work 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 is based on the ideas presented by Bock [2].

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 (e.g., ≪ReadSelf≫). 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 5 shows the behaviour of the ShoppingCart object after receiving an addItem event, modelled according to the UML Action language and using the developed Profile. This procedure has two parameters, the selected book and the required quantity. The procedure creates a new association link object of the SelectedProduct association class (see Figure 3), between the ShoppingCart hosting the behaviour (returned by the ReadSelf action), and the selected book. The required quantity is finally written in the corresponding attribute (structural feature) of the created link object.

Figure 6 illustrates the behaviour requestCreditConfirmation, which is executed when the ShoppingCart object enters the Payment concurrent state (see Figure 4). In this case, the object calculates the total price of the order, recovers customer credit card data, and gets a reference to the CreditCard service. With these parameters, it requests a creditApproval from the CreditCard service. If the service confirms the transaction, the check out process continues; if the transaction is not approved, a message communicating the error is shown to the user.

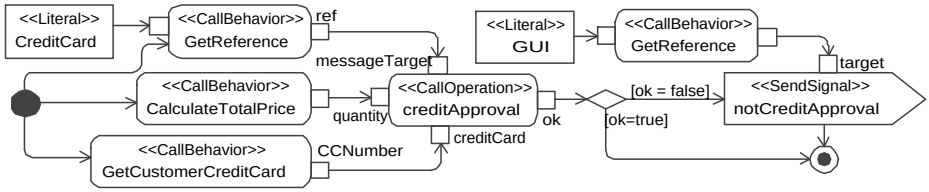


Figure 6: requestCreditConfirmation procedure for requesting credit approval

In this latter case, a behaviour defined by us (e.g., a subroutine), called GetReference, is used to get references to relevant components of the application, such as the GUI (Graphical User Interface), the CreditCard or Delivery external services.

The procedures modelled in Figures 5 and 6 are not complete as they do not observe Persistence, Encryption and Currency Conversion, which are added using aspects in the next section.

5 ASPECT-ORIENTED MODELLING

In order to obtain a complete model of the Online Book Store System, Persistence and Encryption must be added. This section illustrates how they are modelled in an aspect-oriented fashion. To support the construction of aspect-oriented executable models, this paper introduces the AOEM (Aspect-Oriented Executable Modelling) UML 2.0 Profile, which is integrated with the principles of Executable UML and its action language.

According to Fuentes and Sánchez [13], the AOEM Profile is specified in three steps: (1) definition of the joinpoint model; (2) definition of the modelling of aspects and their associated elements, such as advices; and (3) definition of the rules (e.g. pointcuts) that indicate how these aspects must be composed with the modules they crosscut.

Joinpoint model

The AOEM Profile uses a non-invasive joinpoint model (i.e., only the interception of execution points visible on the module interfaces is allowed), similar to JAsCo [31], Lasagne [32] or CAM/DAOP [26] models. This joinpoint model 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 behaviour of the design modules: (1) object creation and destruction; (2) the sending and receiving of a method; (3) the sending and receiving of a signal; (4) the throwing of an event; and (5) the raising of an exception. Aspect methods can be executed

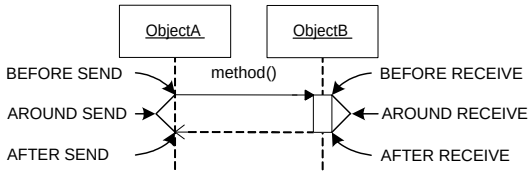


Figure 7: Message sending and receiving joinpoint model

GetMessName	Returns the name of the intercepted message
GetMessArg(n)	Returns the n-argument of the intercepted message
GetArgNamed	Returns the argument with the specified name
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 behaviour

Table 2: Aspect-oriented actions

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 the modelling level, can be considered special kinds of message (call or execution) joinpoints. Specific joinpoints concerning message sending and receiving are illustrated in Figure 7.

Aspect modelling

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

Thus, advices are modelled as activity diagrams (common procedures) without input objects. They can have zero, one or more output pins, in order to be able to modify values of the intercepted object flow. For instance, if an advice is executed before a message is sent, the value could modify the value of message arguments. The updated values would be placed as output values of the activity diagram representing the advice. In the particular case of advices executed *around* a joinpoint, the advice and the intercepted message should have the same number and kind of output objects.

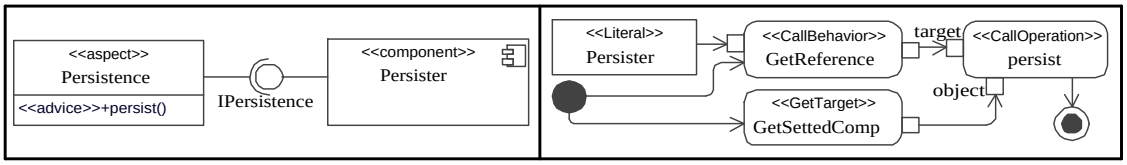


Figure 8: Persistence aspect: (left) structure definition (right) advice behaviour

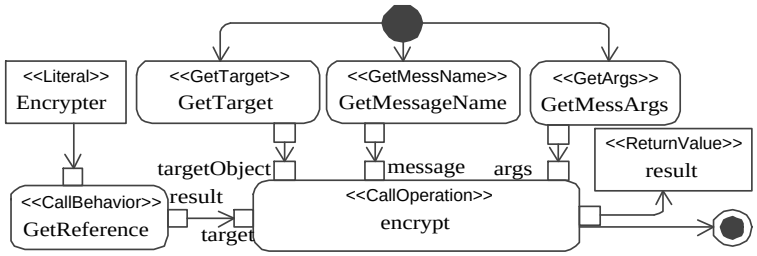


Figure 9: encrypt() advice

To introduce Persistence, Encryption and Currency Conversion into the Online Book Store system, three aspects are created. They are associated with the Persister, Encrypter and CurrencyConverter common components. (Figure 8 (left) illustrates the case for Persistence). These components have methods to persist objects, handle encrypted communications and perform conversions between different currencies, respectively. On the joinpoints, the task of the advices is to collect the required data to invoke the appropriate Persister, Encrypter and CurrencyConverter services.

Figure 8 (left) shows the design of persistence as an aspect. A class Persistence, stereotyped as ≪aspect≫, is created. An advice persist is added to this class. The aspect is associated, as commented above, with a classical Persister component. Figure 8 (right) shows the persist() advice, which persists objects after they have been modified as a result of a method execution. The advice recovers a reference to the object to be persisted (≪GetTarget≫ action) and calls the persist(object) method of the Persister component with this data.

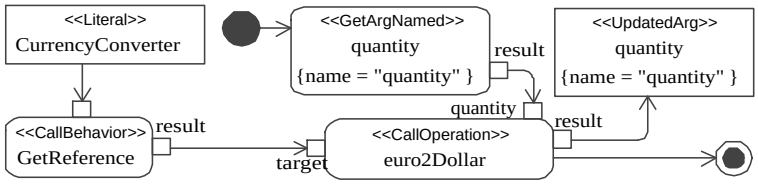


Figure 10: euro2Dollar() advice

Figure 9 depicts the advice for encryption. It relies on the Object[] encrypt(target: Object, method:String, args:Object[]) method of the Encrypter component. The encrypt method sends an encrypted request to the target object for executing method with args as arguments. It returns the collection of (decrypted) values resulting from executing the method. Thus, the task of the encryption advice (Figure 9) is to extract from the intercepted joinpoint (the sending of a message) the target object (≪GetTarget≫ action), the message name (≪GetMessName≫ action) and the arguments of the message (≪GetArgs≫ action), and, with these values, to invoke the encrypt method of the Encrypter component (≪CallOperation≫ action). The advice returns the values (decrypted) which result from executing the method.

Figure 10 illustrates the advice for currency conversion. In this case, the advice is executed when a message is sent to the CreditCard component, and it is required to convert an amount from Euros to US Dollars. The euro2Dollar advice gets the value of the argument of the message called quantity (≪CallOperation≫ action), and, with this argument, it invokes the euro2Dollar method of the CurrencyConverter component. The resulting quantity of the conversion is placed as an output of the advice, and it is indicated that the new value must be used as the quantity argument for the intercepted message.

Pointcut modelling

Finally, to complete our aspect-oriented model, we need to construct the pointcuts that specify how to compose the crosscutting concerns modelled as aspects, i.e., Persistence, Encryption and Currency Conversion, with the design they crosscut, i.e., the ShoppingCart class in our example.

A pointcut expression is a pattern that matches several join points and associates them with one or more aspect advices. In addition, a pointcut may express some constraints (e.g. the joinpoint has to be in a specific execution flow) that must be satisfied in order to execute the associated advices. At the modelling level, the common practice for specifying pointcuts is, basically, to use UML diagrams with wildcards (e.g., "*" to represent any sequence of characters or "?" to represent any sequence of arguments) [13, 29]. As our intention is to intercept interactions between objects (message sending/receiving), sequence diagrams are selected to model pointcuts because they are the main elements in UML that represent 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: an ordered collection of aspect advices, which will be executed in the specified order on the joinpoints selected by the pointcut.

The specific message of the sequence diagram that must be intercepted is stereotyped as ≪joinpoint≫. This stereotype has two tagged values: (1)point, which indicates whether the interception point is either the sending (SEND) or the reception

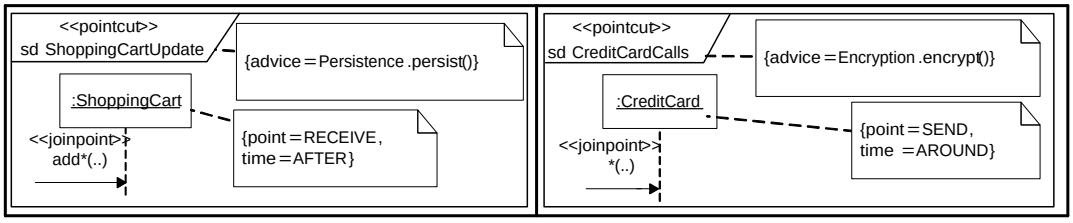


Figure 11: Pointcuts for: (left) Persistence (right) Encryption

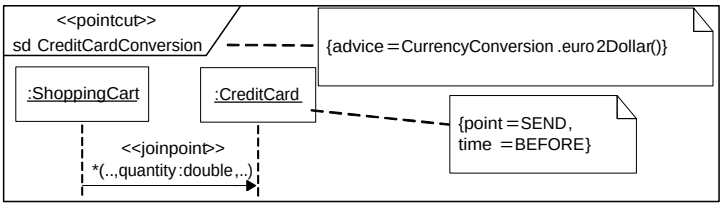


Figure 12: Pointcut for Currency Conversion

(RECEIVE) of the message; and (2) time, which 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 11 (left). 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 number of arguments. This message can come from any source, as this is not specified.

Figure 11 (right) shows the pointcut for adding Encryption, which specifies that around sending any message ("*(..)" wildcard combination) from any source (this is not specified) to the CreditCard service, the encrypt advice must be executed, in order to fulfill the security requirements imposed by the CreditCard service.

Figure 12 depicts the pointcut for Currency Conversion, which specifies that, each time any message, which contains an argument called quantity and the type of the argument is double, is sent from a ShoppingCart object to the CreditCard service, before sending such message, the euro2Dollar advice from the CurrencyConversion aspect must be executed.

More complex pointcuts than those shown in Figures 11 and 12 can be specified using the AOEM Profile. For instance, messages above the ≪joinpoint≫ message can be used to specify cflow-like constraints. These messages have a tagged-value cflowActive which is set to true if the joinpoint must be in the message flow, and it

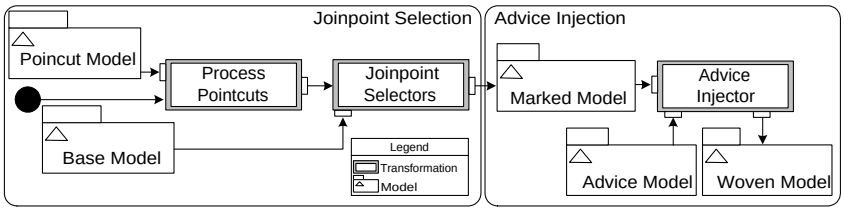


Figure 13: Model weaving process

is set to false if the joinpoint must not be in their flow. The ≪pointcut≫ stereotype has a tagged value named withinCode which serves to express withincode conditions. It is a collection of (method, active) tuples, where method indicates the scope of the withincode condition and active is a boolean value indicating whether the joinpoint must be within the specified scope[1].

6 A WEAVER FOR ASPECT-ORIENTED EXECUTABLE UML MODELS

In previous sections, the selected use case of the Online Book Store example has been modelled in an aspect-oriented fashion. However, to be able to execute it, aspect behaviours must be added to the modules they crosscut according to the pointcut specifications, i.e., the *weaving* process has to be executed. As the ultimate behaviour of common classes and aspects is expressed by means of activity diagrams, the problem of weaving executable models can be reduced to the problem of weaving activity diagrams. This section describes a static weaver for aspect-oriented models that conform to the AOEM Profile.

Before designing the weaver, a common problem related to aspect interaction needs to be solved: two pointcuts can share a subset of joinpoints which satisfies both them. In this case, the execution ordering of advices might be important to ensure the correctness of the application [12]. To solve this issue, our 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. In the Online Book Store example, this problem does not appear because, although euro2Dollar and encrypt can be executed over the sending of a same message (creditApproval), euro2Dollar is executed *before* the sending and encrypt *around* the sending, therefore euro2Dollar is always executed before encrypt.

The task of the model weaver is to inject the advice behaviours into the places

[1]Interested readers can find further information of the AOEM Profile in http://www.lcc.uma.es/∼pablo/AOExecutableUMLModels

indicated by the pointcut specifications. The weaving process is defined as a chain of model transformations. Figure 13 shows a simplified version of this chain of model transformations, where all the activities related to the checking of pointcut constraints, such as cflow constraints, have been removed in order to avoid overwhelming the reader with too much detail and for the sake of brevity.

We opted for a straightforward solution for implementing the model transformations, because, when we implemented the model weaver, there was a lack of stable and mature model transformation languages that were able to deal with UML Profiles. Thus, in order to avoid learning non-standard and not 100% stable model transformation languages, such as VMTS [18] or UMLX [34], which could disappear in the future because of the appearance of more standard and powerful languages, such as QVT [23], we opted for a simple solution to implement the model transformations and to demonstrate our ideas: we manipulate directly the XMI representation of the models using standards such XSLT [36] and XPath [35], as we had previous experience using these languages, and they provide a tool-independent, declarative, robust and standard solution for implementing simple model transformations.

This weaving process is comprised of two main phases, as illustrated in 13: joinpoint selection and aspect injection. Each one of these phases is explained in the following subsections.

Joinpoint Selection

First of all, the pointcut model is processed by the ProcessPointcuts model transformation, which generates a set of model transformations, called JoinpointSelectors, as output. A JoinpointSelector model transformation serves to search all the joinpoints that are selected by a pointcut. These joinpoints are stereotyped as ≪selected joinpoint≫, and the JoinpointSelector adds two tagged values to this sterotype: the advice that must be executed on that joinpoint and the advice execution time (i.e., BEFORE, AFTER, AROUND). This information will be required by the AdviceInjector model transformation in the next step. After applying the JoinpointSelectors to the BaseModel, the MarkedModel, how and where advices must be injected, as specified by the pointcuts, is obtained.

As the model weaver processes the XMI representation (an XML document) of the models, it allows the use of XPath expressions [35] to search the selected joinpoints. An XPath expression specifies a pattern that matches several XML tags within an XML document. Hence, the pattern specified by a pointcut is automatically transformed by the model weaver (using a model to text transformation) into a set of XPath expressions embedded in XSLT transformations, which select all the XML tags corresponding to joinpoints selected by the pointcut, mark these joinpoints and add the required information (advice name and execution time).

Figure 14 shows the transformation of the pointcut to add encryption (see Figure 11 (right)) into an XPath expression. Lines 01-03 shows an excerpt of the XMI

```
<!-- Credit Card class description -->
01 <packagedElement xmi:type="uml:Component" xmi:id="N1Oc" name="CreditCard">
02    ...
03 </packagedElement>

<!-- A Call Action to CreditCard service -->
05 <node xmi:type="uml:CallOperationAction" xmi:id="VRn1" operation="QD1i">
06   <target xmi:id="7S3g" name="target" type="N1Oc"/>
07 </node>

<!-- XPath expression for selecting any call to the CreditCard service -->
08 "//node[@xmi:type='uml:CallOperationAction' and target[@type='N1Oc']]"
```

Figure 14: Transformation into an XPath expression of the pointcut for encryption

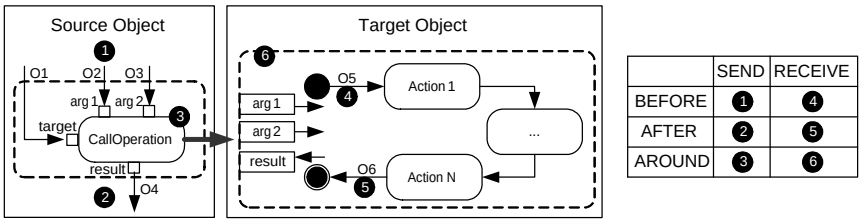


Figure 15: Advice injection

representation of the CreditCard component. Lines 04-07 contain an excerpt of the XMI representation of a call action to the CreditCard component ('QD1i' is the XMI identifier of the creditApproval operation). Line 06 states that the target of this call is of the CreditCard type. Line 09 illustrates the XPath expression for selecting all calls with the CreditCard class as target.

Using the XPath expression of Figure 14, the joinpoint selector: (1) looks for all the joinpoints selected by the pointcut for encryption; (2) marks these joinpoints as ≪selected joinpoint≫; and (3) adds advice=Encryption.encrypt() and time=AROUND as tagged values of the stereotype, using XSLT transformations.

Advice Injection

In the second step, the *advice injection*, the corresponding aspect advices must be injected into the selected joinpoints (call actions and activities representing procedures in our particular case, as shown in Figure 15). The AdviceInjector model transformation (see Figure 13) takes as inputs the MarkedModel and the AspectModel and produces as output the woven model.

Depending on the kind of joinpoint the advice crosscuts (i.e., the sending (SEND) or the reception (RECEIVE) of a message), and the execution time of the advice (i.e., BEFORE, AFTER or AROUND), an advice can be injected at six different places,

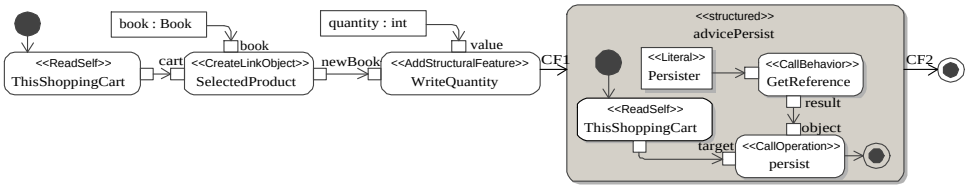


Figure 16: persist woven into ShoppingCart::updateItems

as illustrated in Figure 15. Each place corresponds to a specific value of the pair (joinpoint kind, execution time), as shown in Figure 15. For instance, if an advice has to be executed BEFORE SEND a method, it is added between the call action and the actions that precede it (Figure 15, label 1). If it has to be executed AROUND SEND or AROUND RECEIVE, the corresponding call action (Figure 15, label 3) or activity (Figure 15, label 6), respectively, are substituted by the advice.

Aspect advices are injected as structured activities (see Figures 16 and 17, gray background) inside the procedures they crosscut. These structured activities contain the same behaviour as the advices, but the aspect-oriented actions introduced by the AOEM Profile are appropriately transformed into common UML actions. The advice injection plus the aspect-oriented actions transformation require updating the object and control flows of the original procedures in order to ensure the correctness of the composition. These concepts are illustrated using the injection of the persist, encrypt and euro2Dollar advices (Figures 8, 9 and 10) into the base model of the Online Book Store System (Figures 5 and 6), as an example.

Figure 16 shows the injection of the persist advice into the updateItems procedure (Figure 5). It is injected just before the final node of the addItem procedure (an AFTER RECEIVE case (Figure 15, label 5)). The original control flow that went from the WriteQuantity action to the final node is removed; and the new control flows CF1 and CF2 are created. The GetTarget action of the original advice (Figure 8 (right)) is replaced by a ReadSelf action, because the advice is injected into the target object.

Figure 17 illustrates the result of injecting the euro2Dollar and encrypt advices before and around the creditApproval call to the CreditCard component, inside the requestCreditConfirmation procedure (Figure 6).

As the encrypt advice has to be injected according to an AROUND SEND case (Figure 15, label 3), the original call action is substituted by a structured activity (encryptAdvice) representing the encrypt advice, whereas the euro2Dollar advice, a BEFORE SEND case (Figure 15, label 1), is inserted as a structured activity (euro2DollarAdvice) between the original call (which, in this case, has been substituted by the encryptAdvice), and the actions that precede it.

In the advice2DollarAdvice structured activity, the aspect-oriented action GetArgNamed, used to retrieve the argument of the message named quantity, is transformed into an input parameter (also called quantity), which provides the argument for the

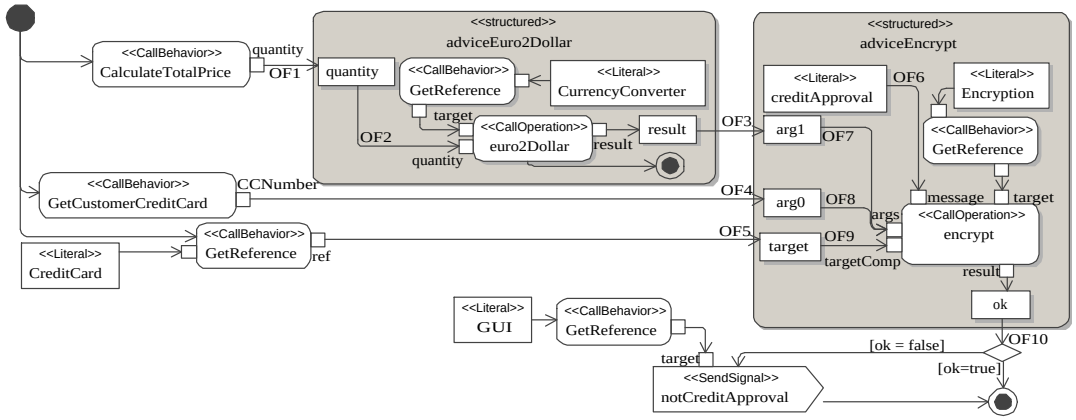


Figure 17: euro2Dollar and encrypt woven into ShoppingCart::requestCreditConfirmation

euro2Dollar call (OF2), of the structured activity. The object flow which provided this parameter in the original method call is removed and redirected to this new input parameter (i.e., the object flow OF1 is created). The output of the advice (the result object) is connected (object flow OF3) to the quantity argument of the original call (in this case substituted by the encryptAdvice), as indicated in the advice by the UpdateArg stereotype (Figure 10).

In the encryptAdvice structured activity, an input parameter is created by each argument of the substituted call action (arg0 and arg1). An additional input parameter (target) is also introduced to collect the target object of the call action. An output parameter (result) holds the return value of the substituted call action. In order to appropriately link the structured activity with the preceding and succeeding actions, the object flows OF3, OF4, OF5 and OF10 are generated. OF3, OF4 and OF5 supply the input parameters and the target object to the structured activity input objects. OF10 passes the advice return value to the following actions. The aspect-oriented actions of the original encrypt advice (Figure 9) are transformed as follows: (1) GetArgs originates the object flows OF8 and OF9; (2) GetMessName produces the literal creditApproval which feeds the message value pin (OF6); and (3) the GetTarget action gives rise to the object flow OF9.

We would like to point out that the transformation of aspect-oriented actions, the advice injection and the updating of object/control flows involve many special cases with many low-level details which are not mentioned here for the sake of brevity and simplicity[2].

[2]Interested readers can find further information of the AOEM model weaver in http://www.lcc.uma.es/∼pablo/AOExecutableUMLModels

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 select one that supports UML executable modelling and full capabilities for importing/exporting to a standard interchange format such as XMI. Unfortunately, at the present time there is no tool in existence which satisfies both requirements at the same time. The tools that have full XMI import/export facilities, such as MagicDraw[3], do not simulate UML models; and the UML tools that are able to simulate models, such as Rhapsody[4] or iUML[5], do not export the models to XMI, or any other standard format, fully. The part regarding actions is often lost, making it impossible to perform the weaving. Thus, to validate our results, we had to use a chain of tools, each one meeting our requirements partially.

UML modelling was done using the UML2 plugin for Eclipse[6], which is the most complete implementation of the UML 2.0 metamodel, including the whole UML Action language. It also offers full XMI export/import capabilities. Unfortunately, this plugin does not offer graphical support for constructing/visualizing UML diagrams, as it aims to be a "UML without pictures" [3] complete implementation of the UML 2.0 metamodel. MagicDraw was used to elaborate the graphical representation of the UML diagrams presented throughout this paper.

In a second phase, we need to perform the weaving. It can be implemented using any kind of model transformation language able to deal with UML Profiles and that can accept several models as input. As mentioned before, at the time of implementing the model weaver, there was a lack of mature model transformation languages with such features, and we opted for manipulating the XMI representation of the models directly. The static weaver takes the XMI representation 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, such as Java plus DOM and XSLT.

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, called Pópulo UML Virtual Machine, which is provided as an Eclipse plugin. Figure 18

[3] http://www.magicdraw.com/
[4] http://www.ilogix.com/homepage.aspx
[5] http://www.kc.com/products/iuml.php
[6] http://www.eclipse.org/uml2/

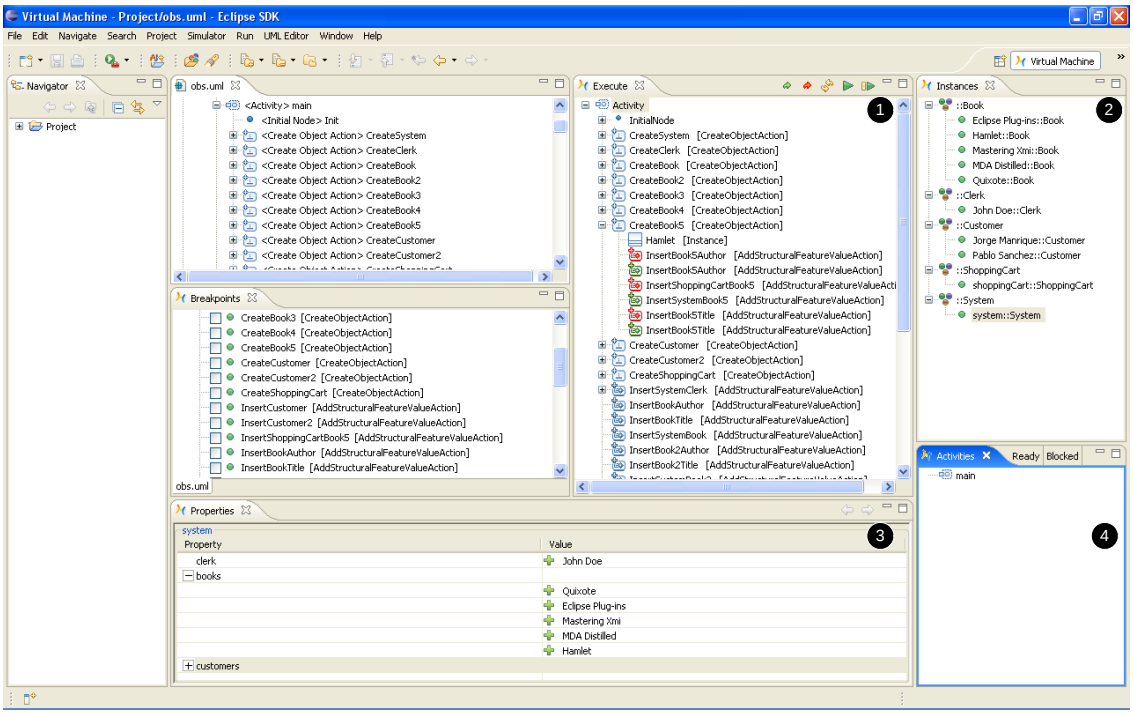


Figure 18: The Pópulo UML Virtual Machine

shows a screenshot of this tool. It allows designers to visualise the behaviour of executable UML models by interpreting the UML actions. It works as a model debugger and provides four views for observing: (1) the execution trace of the model (Figure 18, label 1); (2) the current status of the objects created by the application (Figure 18, label 2); (3) the values of the attributes of such objects (Figure 18, label 3); and (4) the current status of the stack of method calls, the status of the queue of actions and activities ready for execution and the status of the bag of actions and activities that are blocked waiting for receiving some object or control flows (Figure 18, label 4). The Pópulo UML Virtual Machine supports breakpoints and step by step execution. The Pópulo UML Virtual Machine is still under development. It will be released under an open-source license soon[7].

Meanwhile, the work presented in this paper can be reproduced following the steps and the tools described above. We also hope this lack of tool support will be solved when the tool vendors start to fully adopt the UML and XMI 2.0 standards.

[7]Interested readers can find information about the Pópulo tool in http://www.lcc.uma.es/∼pablo/Populo

8 RELATED WORK

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

Sunyé et al. [30] presented a framework for modelling 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 modelling time. The weaving process is implemented as a model transformation from design to implementation, which generates code from the UML actions.

C-SAW [14] is a framework for generating model weavers for aspect-oriented domain models. C-SAW is integrated with the GME [8] modelling environment. It focuses mainly on consistently adding constraints and properties to very large models, rather than encapsulating crosscutting behaviours in aspects. In C-SAW, pointcuts and advices are modelled using the ECL language, defined by the authors, which is a subset of OCL. This kind of declarative and OCL-based textual languages might be cumbersome for specifying pointcuts, as demonstrated by Stein et al [28]. Advices are also modelled using the ECL language, therefore, aspect and base models are expressed in different notations, which could generate some understanding problems. A more optimal solution, in our opinion, is to use the same notation for the aspect and the base model, so the learning curve could decrease for those designers that are already familiar with the notation of the base model. Additionally, the behaviour of the advices is not modelled in a strict sense, instead, it is specified in a syntax similar to C++.

Theme/UML [4] is an extension of UML for aspect-oriented modelling. It 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 [7, 8, 10] present an idea very similar to this paper, called Motorola WEAVR [10]. Currently, Motorola WEAVR can be considered the most mature model weaver, since it has been adopted in production by Motorola. This model weaver is integrated with the TAU G2 tool, and enables the use of powerful code generators provided by these tools. Additionally, it provides a interesting joinpoint model based on states, which allows the specification of semantic pointcuts in reactive systems [9]. However, Motorola WEAVR is based on the Telelogic TAU G2 implementation of the Executable UML principles. Cottenier et al defines an aspect-oriented Profile that extends the Telelogic SDL metamodel for the

[8] http://www.isis.vanderbilt.edu/projects/gme/

Action Semantics. This notation is not compatible with the current UML Action language and introduces some proprietary features that reduce its interoperability and tool-independence. The aspect-oriented model weaver is implemented as a Telelogic add-in [8], therefore it is not portable and tool-independent. Additionally, the weaving process is not clearly described in their work.

Groher and Völter [15] present a model weaver, called XWeave, which allows the weaving of models and metamodels based on the Eclipse Modelling Framework [9]. However, this model weaver is mainly focused on the structural definition of models and metamodels instead of their behaviour. Additionally, it does not contain any mechanism to model the precise behaviour of aspects. Ubayashi et al [33] propose MMAP (metamodel access protocol), which can be viewed as reflection mechanisms for manipulating a model as an instance of its metamodel. MMAP provides interesting benefits for the construction of model weavers for aspect oriented models. However, MMAP is limited to structural models (e.g. class diagrams), so the handling of crosscutting behaviours is not possible.

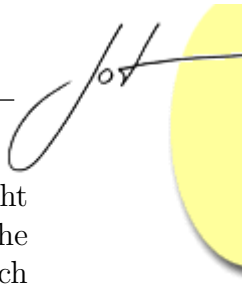
9 DISCUSSION

Using aspect-orientation at the modelling level, software designers can avoid the well-known problems derived from scattered and tangled representations of crosscutting concerns. Several aspect-oriented design notations have appeared in recent years [27], but aspect-oriented designers do not have any support for modelling the precise behaviour of their aspects and verifying their models. The executability of aspect-oriented models is a prerequisite for simulating them and running test cases that help to verify these models. This paper has described how to construct and execute aspect-oriented models. However, a systematic process for designing the test cases that allow designers to verify their models, such as those described by Xu and Xu [37], is beyond the scope of this paper and has been left for future work. This paper only describes the infrastructure required for executing test cases at the model level.

Currently, using our approach, an aspect-oriented model can be debugged by means of its execution. We provide two different scenarios using the Online Book Store System where our approach is useful for detecting potential errors due to the use of aspect-orientation.

Figures 11 (right) and 12 depict the pointcuts for adding encryption and currency conversion to the communications with the CreditCard service. The first pointcut does not specify any source, so it intercepts all the messages with the CreditCard service as target. The second pointcut specifies that the source of the message must be the ShoppingCart object. Therefore, the first pointcut might intercept communications coming from sources that do not require encryption, because they are performed over secure networks or we are using, for instance, a third-party component that

[9] http://www.eclipse.org/modeling/emf/



has encryption support built-in. On the other hand, the second pointcut might be missing some joinpoints, if there are other classes that communicate with the CreditCard service and that send quantities expressed in Euros. Hence, our approach can be used to analyse that a pointcut selects all the joinpoints an aspect should crosscut and these places only.

Figure 17 showed the result of injecting currency conversion and encryption into the requestCreditConfirmation procedure. In this case, the amount of the credit to be approved by the CreditCard service is converted to US Dollars and then encrypted with the other arguments of the request. Then, the encrypted message is sent to the CreditCard service. If we had defined a different ordering for the execution of these aspects, e.g., applying encryption before currency conversion, the composed behaviour would probably be incorrect. Therefore, our approach is also useful for analysing problems related to aspect interaction.

Additionally, using this approach, different alternative solutions can be analysed without the necessity of implementing them, simply by means of observing the behaviour of their models. We would like to point out that, in order to analyse some critical parts, or different alternatives, of an aspect-oriented model, designers do not need to specify the model completely. They only need to specify the behaviour of those parts that they want to analyse, and the irrelevant behaviours, such as CalculateTotalPrice in Figure 6, can be simply filled with a dummy specification, which will be refined at the implementation level. A deeper discussion on when it is faster, cheaper and more desirable to model instead of directly coding is not the goal of this paper, but the interested reader can refer to Cottenier et al [10, 8] and the Saturn experience [19], and Mellor and Balcer [20]. Counter positions can be found in Bell [1] and Kleppe et al [17]. A critical discussion of these issues is available in Hailpern and Tarr [20].

Another contribution of this approach is that, as the woven model is a common UML executable model, 100% of the code can be automatically generated if desired (this feature is currently supported by several tools, such as Rhapsody, TAU G2 or Rational Rose RT). 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.

We state in this paper that the AOEM model weaver is tool-independent because it works on the XMI representation of the models instead of using specific APIs from proprietary tools, as in the case of Motorola WEAVR. Nevertheless, this is not true at all, since different XMI standard interpretations have produced non-interoperable tools. Therefore, at the moment, our model weaver is constrained to the XMI produced by the UML2 plugin of Eclipse. However, this XMI flavour is becoming adopted as de-facto standard in the model-driven community, and commercial UML tools (e.g. Rational Rose, MagicDraw) are able to export models to this format. We hope these XMI interoperability problems disappear in the next few years.

Finally, at the current time, aspect-oriented models have to be executed from

their woven form. Therefore, designers must know how advices are transformed during the woven process, which might result in usability problems. The development of traceability mechanisms during the woven process has been left for future work . These mechanisms would enable the execution of aspect-oriented models in their unwoven form to be visualised.

10 CONCLUSIONS AND FUTURE WORK

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

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 Joinpoint Designation Diagrams (JPDDs) [29] in order to provide more expressive pointcuts. It was not done in this paper as it is not a trivial task. First, as JPDDs are not UML compliant currently, a UML Profile should be derived first. Secondly, to generate joinpoint selectors is not as simple as for our current pointcuts. Thus, we opted for using a simpler, but quite powerful, pointcut model following the JPDD philosophy for the first version of the AOEM Profile and the associated model weaver.

Experienced readers may miss inter-type declarations in the AOEM Profile. They were explicitly left out since the authors position is that they are not strictly required for aspect-orientation [13]. Nevertheless, they will be added to the AOEM Profile in future versions. More flexible ways for accessing the joinpoint context will also be investigated.

Additionally, the current implementation of the model weaver using XPath, XSLT, Java and DOM presents some scalability and maintenance problems. After experimenting with different model transformation languages, we have opted for implementing the model weaver using ATL[10]. We will also continue working on the implementation of the UML Virtual Machine and a user interface which provides support for a user-friendly model simulation.

11 ACKNOWLEDGEMENTS

This work has been supported by Spanish Ministerio de Ciencia y Tecnologa (MCYT) Project TIN2005-09405-C02-01 and European Commission Grant IST-2-004349-NOE AOSD-Europe and the European Commission STREP Project AMPLE IST-033710.

[10] http://www.eclipse.org/m2m/atl/

Authors would also like to thank Jorge Manrique his valuable contribution to the implementation of the Pópulo UML Virtual Machine as well as all the anonymous reviewers their useful comments and suggestions.

REFERENCES

[1] A. E. Bell. Death by uml fever. *ACM Queue*, 2(1):72–80, March 2004.

[2] C. Bock. UML 2 Activity and Action Models. *Journal of Object Technology*, 2(4):43–53, July-August 2003.

[3] C. Bock. UML without Pictures. *IEEE Software*, 20(5):33–35, September-October 2003.

[4] S. Clarke and E. Baniassad. *Aspect-Oriented Analysis and Design : The Theme Approach*. Addison-Wesley Professional, March 2005.

[5] C. Clifton and G. T. Leavens. A design discipline and language features for modular reasoning in aspect-oriented programs. Technical Report TR #05-23, Department of Computer Science, Iowa State University, December 2005.

[6] A. Colyer, A. Clement, G. Harley, and M. Webster. *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*. Addison-Wesley Professional, December 2004.

[7] T. Cottenier, A. V. de Berg, and T. Elrad. Modelling 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.

[8] 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 Modelling (AOM), 9th Int. Conference on Model Driven Engineering, Languages and Systems (MODELS)*, Genova (Italy), October 2006.

[9] T. Cottenier, A. van den Berg, and T. Elrad. Joinpoint inference from behavioral specification to implementation. In *Proc. of the 21st European Conference on Object-Oriented Programming (ECOOP)*, Berlin (Germany), July-August 2007.

[10] 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)*, Vancouver (British Columbia, Canada), March 2007.

[11] L. Doldi. *Validation of Telecom Systems with SDL: The Art of SDL Simulation and Reachability Analysis*. Wiley, April 2003.

[12] 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, *Proc. of 1st Int. Conference on Generative Pogamming and Component Engineering (GPCE)*, volume 2487 of *LNCS*, pages 173–188, Pittsburgh (Pennsylvania, USA), October 2002.

[13] L. Fuentes and P. Sánchez. Elaborating UML 2.0 Profiles for AO Design. In *Proc. of the 8th Workshop on Aspect-Oriented Modelling (AOM), 5th Int. Conf. on Aspect-Oriented Software Development (AOSD)*, Bonn (Germany), March 2006.

[14] J. Gray, T. Bapty, S. Neema, D. C. Schmidt, A. Gokhale, and B. Natarajan. An approach for supporting aspect-oriented domain modelling. In F. Pfenning and Y. Smaragdakis, editors, *Proc. of the 2nd Int. Conference on Generative Programming and Component Engineering (GPCE)*, volume 2830 of *Lecture Notes in Computer Science*, pages 151–168, Erfurt (Germany), September 2003.

[15] I. Groher and M. Völter. XWeave: models and aspects in concert. In *Proc. of 10th Int. Workshop on Aspect-Oriented Modelling (AOM), 6th Int. Conference on Aspect-Oriented Software Development (AOSD)*, pages 35–40, Vancouver (British Columbia, Canada), March 2007.

[16] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *Proc. of the 27th International Conference on Software Engineering (ICSE)*, pages 49–58, St. Louis (Missouri, USA), May 2005.

[17] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture–Practice and Promise*. Addison-Wesley Professional, April 2003.

[18] T. Levendovszky, L. Lengyel, and H. Charaf. Extending the DPO Approach for Topological Validation of Metamodel-Level Graph Rewriting Rules. *WSEAS Transactions on Information Science and Applications*, 2(2):226–231, February 2005.

[19] E. Long, A. Misra, and J. Sztipanovits. Increasing Productivity at Saturn. *Computer*, 31(8):35–43, August 1998.

[20] S. Mellor and M. Balcer. *Executable UML: A Foundation for Model Driven Architecture*. Addison-Wesley Professional, March 2002.

[21] Object Management Group (OMG). Action Semantics for the UML Request For Proposal (ad/98-11-01), November 1999.

[22] Object Management Group (OMG). MOF 2.0/XMI Mapping Specification, v2.1 (formal/05-09-01), September 2005.

[23] Object Management Group (OMG). MOF QVT Final Adopted Specification (ptc/05-11-01), November 2005.

[24] Object Management Group (OMG). Semantics of a Foundational Subset for Executable UML Models Request For Proposal (ad/2005-04-02), April 2005.

[25] Object Management Group (OMG). Unified Modelling Language: Superstructure v2.0 (formal/05-07-04). Chapter 5: Actions, July 2005.

[26] M. Pinto, L. Fuentes, and J. M. Troya. A Dynamic Component and Aspect-Oriented Platform. *The Computer Journal*, 48(4):401–420, March 2005.

[27] R. Chitchyan et al. Report synthesizing state-of-the-art in aspect-oriented requirements engineering, architectures and design. Technical Report AOSD-Europe Deliverable D11, AOSD-Europe-ULANC-9, Lancaster University, May 2005.

[28] D. Stein, S. Hanenberg, and R. Unland. A Graphical Notation to Specify Model Queries for MDA Transformations on UML Models. In U. Aßmann, M. Akşit, and A. Rensink, editors, *European Workshops on Model Driven Architecture: Foundations and Applications. Revised Selected Papers*, volume 3599 of *Lecture Notes in Computer Science*, pages 77–92, 2005.

[29] 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. Conference on Aspect-Oriented Software Development (AOSD)*, Bonn (Germany), March 2006.

[30] G. Sunyé, F. Pennaneac'h, W.-M. Ho, A. L. Guennec, and J.-M. Jézéquel. Using UML Action Semantics for Executable Modelling and Beyond. In K. R. Dittricha, A. Geppert, and M. C. Norrie, editors, *Proc. of the 13th Int. Conference on Advanced Information Systems Engineering (CAiSE)*, volume 2068 of *LNCS*, pages 433–447, Interlaken (Switzerland), June 2001.

[31] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development. In *Proc of 3rd Int. Conference on Aspect-Oriented Software Development (AOSD)*, pages 21–29, Boston (Massachusetts, USA), March 2003.

[32] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B. N. Jørgensen. Dynamic and selective combination of extensions in component-based applications. In *Proc. of the 23rd International Conference on Software Engineering (ICSE)*, pages 233–242, Toronto (Ontario, Canada), May 2001.

[33] N. Ubayashi, S. Sano, and G. Otsubo. A reflective aspect-oriented model editor based on metamodel extension. In *Proc. of the 1st International Workshop on Modelling in Software Engineering (MISE), 29th Int. Conference on Software Engineering (ICSE)*, page 12, Minneapolis (Minnesota, USA), May 2007.

[34] E. D. Willink. UMLX - A Graphical Transformation Language for MDA. In *2nd Workshop on Generative Techniques in the context of Model Driven*

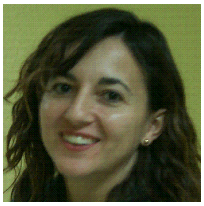
Architecture, 18th Int. Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), Anaheim (California, USA), October 2003.

[35] World Wide Web Consortium (W3C). XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath, November 1999.

[36] World Wide Web Consortium (W3C). XSL Transformations (XSLT) Version 1.0. http://www.w3.org/TR/xslt, November 1999.

[37] D. Xu and W. Xu. State-based incremental testing of aspect-oriented programs. In *Proc. of the 5th Int. Conference on Aspect-Oriented Software Development (AOSD)*, pages 180–189, Bonn (Germany), March 2006.

ABOUT THE AUTHORS



Lidia Fuentes received her PhD in 1998 from the University of Málaga, where has been an Associate Professor since 1993. Her research interests deal with Aspect-Oriented Software Development, Component-Based Software Development, MDD/MDA, SPLs and Software Agents. Her most significant publications can be found in IEEE Transactions of Software Engineering, IEEE Internet Computing or ACM Computing Surveys. She is actively participating in several European research projects on AOSD, MDD and SPLs, such as AOSD-Europe and AMPLE.

Pablo Sánchez is a PhD student at the University of Málaga since 2004. His main research areas are AOSD, MDD and SPL. He has published on aspect-oriented executable modelling and metamodels and model transformations for aspect-oriented models. Currently, he is participating actively in the European Commission funded projects AOSD-Europe and AMPLE; and he can be reached at pablo@lcc.uma.es. See also http://www.lcc.uma.es/∼pablo.