Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 189 (2007) 87–103

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Aspect-Oriented Coordination

Lidia Fuentes[1] and Pablo Sánchez[2]

Dpto. Lenguajes y Ciencias de la Computación
Universidad de Málaga
Málaga, Spain

Abstract

The different software modules (e.g., components) that constitute a system are not isolated entities. In fact, they need to interact according to a certain coordination protocol in order to achieve a common goal. This coordination protocol crosscuts the different software modules, hindering their reusability, independence, maintenance and evolution, but these drawbacks can be solved by separating coordination from computations and encapsulating coordination outside the components that perform computations. Aspect-Oriented Programming has been demonstrated to be an interesting technology for handling certain crosscutting concerns, such as coordination. This paper explains how coordination protocols can be implemented outside computational modules using general purpose aspect-oriented programming languages.

Keywords: Aspect-Orientation, Coordination, Component, Protocol, State Machines.

1 Introduction

The software modules into which a system is decomposed are not isolated, they need to interact following a certain coordination protocol in order to achieve a common goal. In traditional software decomposition techniques, like Object-Oriented [27] or Component-Based [35] techniques, such a coordination protocol cannot be appropriately encapsulated into a single module (e.g., object or component). Thus, each software module that is part of a system has to perform two tasks related to two different issues: (1) *computation*, i.e., the execution of its functionality; and (2) *coordination*, i.e., the management of interactions with other entities in agreement with the coordination protocol. As a result, the coordination concern appears *tangled* together with the computational part of software modules, and *scattered* among them.

The crosscutting nature of coordination and the benefits derived from separating coordination from computation has been acknowledged by the coordination commu-

[1] Email: lff@lcc.uma.es

[2] Email: pablo@lcc.uma.es

1571-0661 © 2007 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2007.05.049

nity, and several *exogenous coordination models* [4,25,28,8] have been proposed in order to overcome these shortcomings. Separating computation from coordination leads to better modularised systems. As they are better modularised, their development, maintenance and evolution is easier [30]. The reusability of each individual software module is also increased because only the computational part of software modules is reused and no coordination protocol is additionally imposed [28,8]. Consequently, software modules can be more easily used as building blocks to compose applications. In addition, as coordination is better encapsulated, it can also be reused as a prebuilt software module in different applications [28,8].

Aspect-Oriented Programming (AOP) [20] has proven in recent years to be an appropriate technique to implement *crosscutting concerns*, such as coordination, outside the software module they crosscut. This paper explains how general purpose aspect-oriented programming languages can be used to implement coordination outside computational modules, according to an exogenous coordination model. We will focus on component-based systems, because we have previous experience in this paradigm and it can be easily understood. Furthermore, the ideas exposed throughout this paper can be easily generalised to other paradigms, such as for example Agent-Oriented [26] ones.

After this introduction, this paper is structured as follows: Section 2 describes the Auction System example. Section 3 justifies why coordination is a crosscutting concern. Section 4 provides some background on Aspect-Oriented Software Development. Section 5 explains how to handle coordination as an aspect. Section 6 shows a specific implementation of coordination as an aspect using JAsCo [34]. Section 7 provides some reflections on our approach. Section 8 comments on related work. Finally, Section 9 outlines conclusions and future work.

2 Motivating example

An on-line Auction System [3] is used as an example to illustrate the ideas presented throughout this paper. The Online Auction System allows subscribed users to negotiate over the buying and selling of goods according to a certain Auction protocol. [4] To participate in an auction, a user must first join it. Once enrolled, a user may make a bid. According to the auction protocol selected, customers will be able to make either several bids or only one; the bids could be either private or public; etc. Figure 1 shows an excerpt of the decomposition of the Auction System into components and interfaces. [5]

Components are considered, as in the Szypersky's definition [35], to be units of composition with contractually specified interfaces and explicit context dependencies, which could be available only in binary form. Components could be of a black-box nature and therefore we may not have access to their internal structure. An interface is a collection of operations/services signatures. An interface

[3] Taken from http://lgl.epfl.ch/research/omtt/auction.html

[4] A description of the different existing auction protocols can be found in http://en.wikipedia.org/wiki/Auction

[5] A complete description can be found in http://www.lcc.uma.es/∼pablo/CaseStudies/AuctionSystem

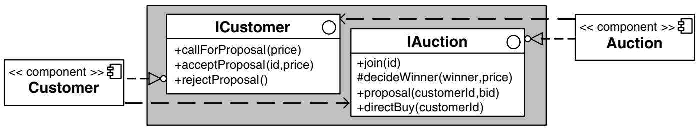

Fig. 1. Excerpt of the Auction System decomposition into components

can be provided or required by a component. Provided interfaces contain the operations/services a component offers to its environment and required interfaces contain the operations/services a component requires from its environment. Components are connected by wiring required/provided interfaces. In Figure 1 the IAuction interface is provided by the Auction component and required by the Customer component.

The Auction Systems is especially interesting for this paper, as different systems can be constructed simply by changing the auction (coordination) protocol that governs the interchange of messages between the Customer and Seller components, keeping the computational part of these components.

3 The necessity of separating computation and coordination

3.1 Coordination protocols

As already commented, software modules are not isolated entities. They communicate with each other following a pattern, called the *coordination protocol*, which governs their communications and interactions. The purpose of the coordination protocol is to provide a means of integrating a certain number of possibly heterogeneous modules together, by interfacing with each module in such a way that the collective set forms a single application [29].

A coordination protocol can be expressed using different formalisms. We have opted for State Machines [17], as they can be easily represented in UML and it is our intention to integrate the ideas that will be presented throughout this paper in Model-Driven Development [38] processes in future work.

The state machine that designs the coordination protocol is created at architecture design time by the software architects, driven by the application requirements. The state machine handles the interaction between two or more components and it is responsible for implementing the coordination protocol between these components, including the activities related to coordination tasks, such as sending notifications or synchronising components. Each state represents a different state of the coordination protocol between components. The events in the transitions represents the interception of messages interchanged between components through their provided/required interfaces. The original UML semantics of event consumption has been slightly modified. An event being consumed by a transition means in our case that the action associated to the event is executed and then the message is dispatched to the target. Events that do not fire any transition are not delivered to their target objects (in this case an exception could be raised). The actions associated to each event implement coordination tasks. This state machine can

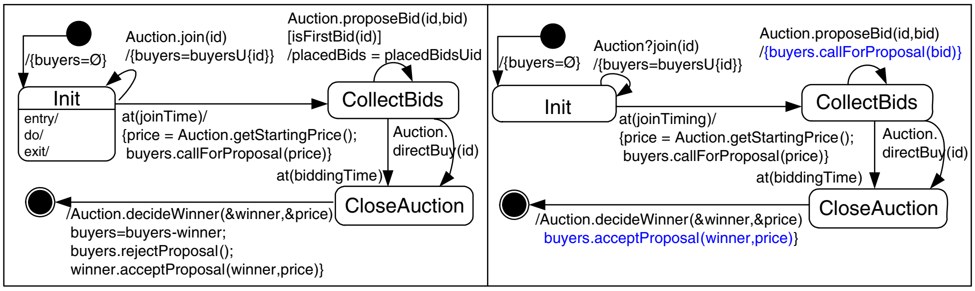
/{buyers=Ø}
Auction.join(id)
/{buyers=buyersU{id}}
Auction.proposeBid(id,bid)
[isFirstBid(id)]
/placedBids = placedBidsUid
Init
entry/
do/
exit/
CollectBids
at(joinTime)/
{price = Auction.getStartingPrice();
buyers.callForProposal(price)}
Auction.
directBuy(id)
at(biddingTime)
CloseAuction
/Auction.decideWinner(&winner,&price)
buyers=buyers-winner;
buyers.rejectProposal();
winner.acceptProposal(winner,price)}
/{buyers=Ø}
Auction?join(id)
/{buyers=buyersU{id}}
Auction.proposeBid(id,bid)
/{buyers.callForProposal(bid)}
Init
CollectBids
at(joinTiming)/
{price = Auction.getStartingPrice();
buyers.callForProposal(price)}
Auction.
directBuy(id)
at(biddingTime)
CloseAuction
/Auction.decideWinner(&winner,&price)
buyers.acceptProposal(winner,price)}

Fig. 2. State Machines for: Private One Bid (left) Public English Style (right) Auctions

be considered as the design of an exogenous coordinator, introduced transparently between the provided/required interfaces of the components (gray background in Figure 1).

In our model, components encapsulate only computations and they are not allowed to perform tasks or activities related to coordination. In order to carry out computations, components can explicitly invoke services of their required interfaces. They invoke these services on the components to which the required interface is wired. For instance, according to Figure 1, a Customer component can request a service (e.g., Auction.join) contained in the IAuction interface from the Auction component. Components invoke services being unaware of connectors, coordinators or any other intermediate entity placed between them.

Focusing on the Auction System example, several coordination protocols are available (e.g., private English style, private one bid, private Vickrey, etc.). Figure 2 (left) shows the coordination protocol, expressed as a UML State Machine, for a *private one bid* auction. In this case, when the auction begins, the list of buyers is initialised to the empty set and the system shifts to the Init state. In this state, only the sending of join messages to the Auction component is possible. Any other message sending is simply skipped (or, otherwise, an exception could be raised). When the specified time for joining the auction (joinTime) expires, a callForProposal message is broadcast to all registered buyers and the system moves to the CollectBids state. In this state, only the sending of proposeBid messages to the Auction component is permitted. Additionally, each customer can send only one message, as it is not possible in a a *private one bid* auction protocol to make multiple bids. When the specified time for bidding (biddingTime) expires, the auction is closed; the winner and the final price are computed and an acceptance message is sent to the winner. As the auction is private, a rejection message is sent to all the non-winner buyers, without any information about the winner or the final price.

Additionally, the Auction component offers a special service directBuy: the seller of each item specifies an amount that is an upper bound to the price of the auctioned item. Customers can invoke this service during the CollectBids phase if they desire to acquire the item for this amount, without making more bids. In this case, the Auction will finish, and the winner of the Auction will be the invoker of the service. Therefore, transitions between states can happen either as result of time consumption or actions executed by the components.

3.2 Pitfalls of merging computation and coordination into components

This section illustrates some of the problems, using the Auction System example, due to the tangling of computation and coordination inside the component that constitutes a system.

As already commented, multiple auction protocols exist, and therefore, it is feasible that the auction protocol of an Auction System could change in the light of new business rules. For instance, the current *private one bid* auction protocol of our Auction System could be replaced by a *public English style* protocol. In this case, two changes, which are commented on below and depicted in Figure 2 (right), would have to be made in the coordination protocol:

(i) Customers would now be able to make several bids on the same item, thus, the guard isFirstBid(id) is removed from the proposeBid transition. Additionally, as the auction is now public, each time a new bid is placed, all registered buyers must be notified about such a bid. Hence, a callForProposal(bid) message is broadcasted to all them, offering the possibility of making a higher bid.

(ii) When the auction finishes, as it is public, all the buyers must be notified of the winner's identity and the final auction price. Thus, instead of sending a rejectProposal() to the non-winning Customers as before, an acceptProposal(winnerId, finalPrice) message is sent to all them.

If coordination, instead of being encapsulated outside the Auction and Customer components, is hard-coded inside them, a change such as described above would imply that:

(i) The Auction and Customer components designed for an Auction System following a *one bid private* protocol could not be reused as they are in an Auction System using a *public English style* protocol.

(ii) As the coordination code could be scattered across the component methods, the change would affect multiple places inside components. In addition, to locate the code related to coordination would not be a straightforward task, as this code would be tangled with the computation code.

(iii) As the coordination protocol is not encapsulated outside components, it can not be reused as a building block to compose Auction System applications that use the same protocol but different interacting components.

In conclusion, it can be stated that coordination crosscuts components, and therefore if coordination is implemented tangled with computation inside components, system maintenance and evolution is hampered, as is the reusability and ease of composition of software modules at individual level. Similar arguments can be found in [4,25,28,8].

Aspect-Oriented Programming [20] aims to implement crosscutting concerns, such as coordination, outside the components they crosscut, in a transparent way. How Aspect-Oriented Programming can help transparently implement exogenous coordinators, such as the state machine of Figure 2, is investigated in this paper.

4 Aspect-Oriented Software Development (AOSD)

Aspect-Oriented Software Development [11] aims to solve the shortcomings of traditional software decomposition techniques, such as Object-Oriented (OO) [27] or Component-Based [35], regarding the encapsulation and composition of crosscutting concerns such as security, monitoring or persistence.

Aspect-orientation (AO) improves the separation of concerns providing the mechanisms for encapsulating appropriately each crosscutting concern into a single module, called *aspect*, and then specifying how this aspect must be composed with the software modules it crosscuts. Aspect-Orientation can be combined with currently existing software development paradigms like Object-oriented [19], Component-Based [32] or Agent-Oriented [2].

Aspect-Orientation principles are described below, together with the aspect-oriented terminology, shown in bold:

(i) Software ***base modules*** (e.g., objects or components) do not contain any reference or code related to crosscutting concerns (e.g., persistence).

(ii) Crosscutting concerns are encapsulated in special modules, named ***aspects***. Aspects contain special methods, called ***advices***, which expose their functionality.

(iii) Each software module permits the injection of crosscutting concerns at specific points, called ***joinpoints*** of their execution flows (e.g., after they have executed a method). When aspect-orientation is applied to component-based systems, joinpoints only refer to the behaviour exposed by the component public interface, such as component creation/destruction, message incoming/outcoming, event throwing, etc.

(iv) Special composition rules, named ***pointcuts***, specify those specific joinpoints of software modules where crosscutting concerns must be injected (e.g., after the execution of all the methods called `foo()`). In addition, pointcuts may specify constraints that must be satisfied at runtime in order to execute an aspect on a joinpoint (e.g., that the system is in a specific state).

(v) Finally, it is responsibility of the aspect-oriented compiler or platform to compose the whole system, ensuring that the aspect advices are executed on the joinpoints selected by the pointcuts. This composition process, known as ***weaving***, can be performed at compile time (called *static weaving*) or at load or even run-time (called *dynamic weaving*). In the latter case, the weaver may allow us to add and remove aspects at run-time. Additionally, the weaving process could require the modification of the source code or binary form of software modules in order to inject aspects into them (known as *invasive weaving*) or not (known as *non-invasive weaving*).

We would like to point out that, in any case, there is no necessity to view or modify the result of the woven application. If the application needs to be updated, the corresponding changes would be performed on the base modules, the pointcuts and/or the aspects. Then, the system will be recompiled (or rewoven). So, crosscutting concerns are never scattered or tangled at the implementation level.

In the case of invasive weaving, crosscutting concerns will be mixed with base modules after compiling, so the separation of concerns is lost at deployment and runtime. Therefore, an aspect cannot be deployed independently of its base modules and/or managed as a prebuilt component. In the case of non-invasive weaving, the aspect is compiled into a single module that can be deployed independently of base modules in different applications, and even placed in aspect repositories as a prebuilt component. In this case, the interception of selected joinpoints and the execution of aspect advices are performed at runtime by some kind of aspect-oriented platform. Thus, the separation of concerns is kept even at runtime. Additionally, the aspect-oriented platform could allow us to add and remove aspects dynamically, which makes this kind of weaving suitable for dynamic architectures, such as open systems.

When applied to component-based systems, the weaving is used to be non-invasive, in order to preserve the black-box property of software components and in order to be able to reuse the aspects as composition modules or prebuilt components.

5 Aspect-Oriented Coordination

The crosscutting nature of coordination protocols has been discussed in Section 3. As coordination is a crosscutting concern, the idea this paper proposes is to use aspect-orientation to adequately encapsulate the implementation of coordination into one or more aspects outside the computational components, in a transparent way.

At design time, a coordination protocol can be designed outside the components it coordinates using some kind of exogenous coordinator model, such as the state machine of Figure 1. Then, at implementation time, this coordinator can be implemented using aspect-orientation. The steps to implement a state machine that designs a coordination protocol using aspect-oriented programming are described below.

Each transition target.message[guard]/action, with origin State1 and target State2, contained in the state machines representing the design of a coordination protocol, is implemented as follows:

(i) All the coordination code, corresponding to the actions that must be executed when the transition is fired, is encapsulated in an aspect advice. This code includes:
  (a) The code associated to the exit of State1 (exit/ clause in UML 2.0).
  (b) The code implementing the action behaviour.
  (c) The code for dispatching the message to the target.
  (d) The code associated to the entrance in State2 (entry/ clause in UML 2.0).
  (e) The code for changing the system state from State1 to State2.
  (f) The code associated to the stay in State2 (do/ clause in UML 2.0).

(ii) Then, a pointcut is constructed to specify that this aspect must be executed if and only if:

(a) The system is in State1.
(b) The guard is satisfied.
(c) message is sent to target.

A pointcut is a pattern that selects several joinpoints of the application execution flow. This pattern is usually composed of: (1) an expression that represents a certain event in the application execution flow (e.g., the reception of a message); and (2) some constraints that must be satisfied in order to execute the advice associated to the pointcut (e.g., that some attributes have some specific values). The expressiveness of each aspect-oriented language regarding these constraints varies widely, so it is not possible to provide general rules about how to translate the steps (ii).c and (ii).d into a pointcut. Sometimes, they could be placed in the pointcut itself and other times these constraints will have to be checked at the beginning of the advice code.

The guards can be expressed in different languages, although the choice suggested by the UML standard is OCL [39]. Then, thinking about automatic model-driven transformations, a generator from OCL to our aspect-oriented programming language is needed in order to generate the implementation of these guards.[6]

After implementing all the transitions, a filter advice is added to the coordination aspect. It filters all the messages that have not fired any transition, according to state machine semantics. This implies, for instance, that a join message sent by a Customer to an Auction out of the Init state, will never reach the target. Optionally, the coordination aspect could raise an exception in order to notify the sender about this special situation.

We would like to point out that an important benefit of handling coordination as an aspect is that components communicate with each other directly, unaware of the coordination protocol, and without needing to keep references to external coordinator entities, such as, for instance, to a Mediator pattern [13], thereby achieving a high degree of transparency.

The next section shows an excerpt from an implementation of the Auction System example following these ideas and using the aspect-oriented language JAsCo (Java Aspect Components) [34].

6 Coordination as an aspect using JAsCo

JAsCo (Java Aspect Components) [34] is an aspect-oriented extension to Java that introduces mainly two new concepts: *aspect beans* and *connectors*[7]. Aspect beans encapsulate crosscutting concerns independently of specific component types. Aspect beans can be considered a special kind of components, which encapsulate crosscutting concerns. They can be compiled and deployed independently of base components. Connectors deploy one or more aspect beans within a particular application.

[6] Code generators from OCL to Java can be found in http://www.klasse.nl/octopus/index.html and http://dresden-ocl.sourceforge.net/introduction.html

[7] The meaning of a connector in JAsCo is not the same as in architectural description languages. A connector in JAsCo is used to bind aspect advices with actual joinpoints of the base components, and not to connect components

One important benefit of JAsCo is that it uses non-invasive weaving, which does not need to modify the internal structure of a component in order to apply aspects to it. Therefore, it can be applied over legacy classes (including Plain Old Java Objects (POJO's)) or third-party Java components without modifying them internally.

An aspect bean is a common Java class which also contains one or more hooks. A *hook* encapsulates a piece of crosscutting code (i.e., an advice in aspect-oriented terminology). A hook (Figure 3, Lines 05-13) is comprised of three main blocks:

Hook constructor (Figure 3, Lines 06-07) It declares an abstract pointcut, i.e. an abstract pattern of the joinpoints (e.g., method calls, method executions) the hook crosscuts. This pattern is instantiated with actual values by the connectors when the aspect is deployed inside a specific application [8].

isApplicable **clause** (Figure 3, Lines 08-09) It is a function that is evaluated at runtime and returns a boolean value, which has to be true in order to execute the hook body on joinpoints the hook crosscuts.

Hook body (Figure 3, Lines 10-12) It contains the crosscutting code. It also declares when this is executed in relation to the joinpoint, i.e., before after or around.

Using the generic scheme presented in the previous section, a coordination protocol, defined by a state machine, is translated into JAsCo as follows:

(i) An aspect bean is created to encapsulate all the code related to the coordination protocol.

(ii) For each transition, a hook is created.

(iii) This hook is executed when a message is sent. This means that all the constructors will have HookName(method(..args) as their signature and they will have call(method), which is the JAsCo abstract pointcut for intercepting the sending of a message, as their body.

(iv) The guard of the transition is transferred to the isApplicable clause, which checks that the guard is satisfied at runtime. Additionally, the isApplicable clause also has to check that the protocol is in the source state of the transition. Otherwise, the transition could not be fired.

(v) All the actions to be carried out when the transition is fired are placed in the hook body.

(vi) Finally, a Filter hook is added, in order to implement the filter aspect to ignore all the messages that do not fire any transition (another option would be to raise an exception).

Figure 3 shows an excerpt of the JAsCo implementation of the *private one bid* auction protocol. All the code related to the coordination concern is encapsulated in the PrivateOneBid aspect bean (Lines 00-29). For sake of brevity, in Figure 3 only the ProposeBid hook appears (Lines 15-29), which implements the Auc-

[8] This mechanism (abstract pattern plus late instantiation by connectors) increase aspect reusability [34]

```
00 class PrivateOneBid {              15 hook ProposeBid {
01                                    16   ProposeBid(method(..args)) {
02  Vector<Joinpoint> tranFired       17     call(method);} // Constructor
03  int state = INIT;                 18   isApplicable () {
04  Vector<CustomerId> placedBids;    19      return (this.state == COLLECT_BIDS) &&
05                                    20       (placeBids.contains(Customer args[0]));}
06  hook Filter {                     21
07    Filter(method(..args)) {        22     around () {
08      call(method);}                23       addTranFired();
09    isApplicable () {               24       placedBids.add(customerId);
10      return !tranFired();}         25       Object returnValue =  proceed();
11    around() {                      26       state = COLLECT_BIDS;
12       return null;                 27       removeTranFired();
13    }                               28       return returnValue;
14  } // around + Filter              29  }} // around + ProposeBid

30 static connector PrivateOneBidConnector {
31  PrivateOneBid.InitJoin   hk_1 = new PrivateOneBid.InitAuctionJoin(* Auction.join(*));
32  PrivateOneBid.ProposeBid hk_2 = new PrivateOneBid.ProposeBid(* Auction.proposeBid(*));
33  .....
34  PrivateOneBid.Filter     hk_3 = new PrivateOneBid.Filter(* *.*(*));}
```

Fig. 3. Excerpt of an private one bid auction protocol implemented in JAsCo

tion.proposeBid(id,amount)[isFirstBid(id)]/{placeBids+=id} transition; and the Filter hook (Lines 05-13).

The ProposeBid hook specifies, by means of its constructor, that it will be executed when (generic) messages are sent (Lines 16-17). "..args" represents the arguments of the intercepted message and it is accesible at runtime as an array of Object in the scope of the hook.

The isApplicable clause (Lines 18-20) returns true if the system is in the COLLECT_BIDS state, i.e., the source state of the transition, (Line 19), and the transition guard is satisfied (Line 20). If both conditions do not hold at runtime, the hook body will not be executed. If they hold, the hook body (Lines 22-29) will be executed around (Line 22), i.e. substituting, the sending of a message. This means we can perform actions before and after such message sending, and even disregard it.

The hook body starts adding the current joinpoint to the tranFired vector. This vector contains the current intercepted joinpoints that fired a transition, and it will be analysed by the Filter to decide if a message must be filtered or not. In order to avoid misunderstandings, we would like to clarify, the same message sent from the same sender to the same receiver and with the same arguments but in two different executions are considered as different joinpoints. Next, the code that implements the actions associated to the transition is executed (Line 24). Then, a call to the proceed() special instruction (Line 25) is performed. This instruction passes control to the intercepted joinpoint, which means the message will continue its path towards the target component. This sending may still be intercepted by other hooks not yet evaluated. After this, the proceed() instruction returns the return value of the intercepted message execution. Subsequently, the hook changes the protocol state to the state target of the transition (Line 26), removes the current joinpoint of the tranFired vector (Line 27) and finishes (Line 28). Problems associated with concurrency, i.e., access to the state variable by several hooks at the same time, are solved by the JAsCo compiler, so programmers do not need to deal with these issues.

Finally, to deploy the coordination aspect bean in a specific application, a JAsCo connector (Lines 30-34) must be constructed. It instantiates the hooks with actual values of the signatures of the methods each hook must intercept. Wildcards can be used to represent signature patterns. For instance, the ProposeBid hook is instantiated to intercept all the sendings of a proposeBid message (Line 32). The proposeBid message can have any number and type of arguments (* wildcard) and it must have the Auction component as target. The sequence in which hooks are executed is determined by the order in which each hook is instantiated inside the connector. The Filter hook must be instantiated at the end of the JAsCo connector in order to ensure it is the last being executed.

As JAsCo implements a non-invasive weaving, the result is semantically equivalent to connecting a new Coordinator component, which implements a specific auction protocol, to the interfaces of the Auction and Customer components. An interesting property of aspect-oriented programming is that the Customer and Auction components are not aware of this coordinator entity.

7 Discussion

In the introduction of this paper it is claimed that aspect-orientation allows developers to achieve a better modularisation of their systems thereby making system development, maintenance and evolution easier. An in-depth verification of this claim is beyond the scope of this paper, but the interested reader can found quantitative studies about these issues in Videira and Bajracharya [37] and Garcia et al [14], where the authors study how Aspect-Orientation can add value to our systems improving the modularisation. On the other hand however, it may add complexity and additional problems such as new dependencies or aspect interactions.

Regarding reusability, there are no existing quantitative studies that prove our claims, but we can provide qualitative scenarios, using the Auction System example, in order to evaluate them.

First of all, changes regarding the auction protocol, using aspect-orientation, and specifically JAsCo, would only affect the coordinator aspect bean, so the same Auction and Customer components could be used to construct Auction Systems with different auction protocols. Therefore, the same base components can be reused to construct systems with different coordination patterns. In the case of JAsCo, which implements a non-invasive dynamic weaving, this can be done even at runtime. If we would like to change the auction protocol of a running system, the coordinator aspect bean that implements it can be unplugged at runtime and the new coordinator aspect bean implementing the new auction protocol plugged in. Therefore, this kind of dynamic aspect-oriented language is really suitable for dynamic evolution, such as open systems.

Secondly, let us suppose we need to construct an Auction System with different computational requirements, but which follows the same auction protocol. For instance, let us suppose the bids can be proposed in different currencies, so the Customer and Auction components must be multicurrency. This change would only affect the computational part of these components, but not the coordination aspect bean implementing the auction protocol. As the coordinator aspect bean is implemented independently of specific implementations of the Auction and Customer components, it can be reused as is.

However, the reusability of aspect bean coordinations has some limitations. Although components are not aware of how they are being coordinated, they have to provide through their interfaces the methods that allow them to be coordinated. For instance, in the Auction System, the Customer component must provide a callForProposal method, in order to allow the reception of the initial auction price. Changes in the coordination protocol could require new methods in component interfaces. Consequently, the reusability of a component and/or an aspect bean coordinator will also depend on the capability of its interface to support the change. In the case of the Auction System, the complete set of existing auction protocols can be known in advance. Thus, the change can be anticipated. In other cases, such a complete set may not be known, thus, the change can not be anticipated, and consequently, some changes may not be adequately supported.

In the ideal case, the interface of the components should not impose any protocol. If a component imposes a protocol which does not match the coordination protocol, it would need to be adapted [40]. How to implement adaptors using aspect-orientation has been presented in [6,12]. Fuentes and Sánchez [12] outline how adaptors can be implemented using aspect-oriented programming to solve signature, protocol and quality of service mismatches. Therefore, an extra benefit of aspect-orientation is that the same linguistic support can be used to implement coordinators and adaptors.

Another limitation of the approach is that the coordinator aspects can only react to the observable behaviour of the components. Therefore, if the state of the coordination protocol needs to advance according to the result of non-observable actions carried out according toa component, this approach would not work. The solution would be to force base components to send events each time this state changes as result of externally non-observable actions. These events would be intercepted by the coordinator aspect, which would carry out the required actions to coordinate the components. Another solution would be to use gray-box components, where components could expose their private methods or even their state through their interfaces. These methods could not be invoked by component clients, and, of course, the component state could only be read, but coordinator aspects could observe them and, for example, execute aspects before, around or after the execution of such methods. This solution is similar to coordinated roles [28].

As the complexity of the coordination pattern grows, the size of the coordination aspect bean would also increase, running the risk of becoming an unmanageable piece of code. Most recent aspect-oriented languages, such as JAsCo or EAOP [9] , are able to apply aspects to aspects. This feature could be used to define coordinators of coordinators, permitting the creation of complex coordinators by composing simple coordination patterns, such as in Reo [4] or exogenous connectors [25].

[9] http://www.emn.fr/x-info/eaop/

8 Related work

This section describes previous work where coordination has already been separated from component computations and encapsulated in a special module. Traditional and aspect-oriented approaches are included.

8.1 Traditional techniques

The idea of separating coordination from computation is not new in the literature. In recent years, design patterns, endogenous and exogenous coordination models have appeared which attempt to achieve this goal. We outline each of them.

8.1.1 Design patterns

Several design patterns [13] focus on solving coordination problems. For instance, the *Mediator* pattern tries to decouple components reducing the number of interconnections. The *Wrapper* and *Role* patterns are used to extend or limit the functionality of a software module. However, design patterns have some drawbacks: (1) They are abstract solutions that are often difficult to instantiate; (2) The resolution of a coordination problem might require combining several design patterns (e.g., a *Mediator* plus a *Wrapper*), with the corresponding increase in complexity; (3) Some of them need to modify the source code of software modules where they are applied, i.e., they are an invasive solution; (4) The implementation of a design pattern using Object-Oriented techniques could lead to crosscutting code, thus they would be better implemented using aspect-orientation [16,14]; and (5) Design patterns often need components to be aware of the existence of external coordinator entities, like a Mediator, and to keep references to them. These shortcomings do not exist in aspect-oriented coordination. An excellent and extensive discussion of the advantages of aspect-orientation over traditional object-oriented techniques, including design patterns, can be found in Truyen's PhD thesis [36].

8.1.2 Endogenous coordination models

Endogenous models and languages, like Linda [15], provide coordination primitives that must be incorporated within a computation. These coordination models have focused on decoupling the senders and the receiver of a message, but entities are still responsible for using communication primitives, localising communication channels, etc. They also have to implement the coordination protocol. If this protocol changed, components would become obsolete. The coordination primitives of endogenous languages crosscut application components, with the discussed drawbacks regarding ease of maintenance, evolution and reusability. Such drawbacks do not exist in aspect-oriented coordination.

8.1.3 Exogenous coordination models

The approach presented in this paper is quite similar to exogenous coordination models, like Manifold [5], Reo [4], coordinated roles [28] or exogenous connectors [25] , where a coordinator entity reacts to the external behaviour of components and initiates actions by itself. Indeed, we have presented Aspect-Oriented Programming as a suitable technology for implementing exogenous coordinators, in agreement with Capizzi et al [7]. However, a disadvantage of exogenous coordination models with respect to aspect-orientation is that base components often need to to interact inderectly with the coordinator entity, for instance, sending events, or to satisfy some constraints, for instance, exogenous connectors [25] forbid components to invoke other components. Using Aspect-Orientation, the coordinator entity is introduced transparently to base components. Additionally, exogenous models often need some special kind of linguistic support, which is exclusive for managing coordination [5,4,28].

Using Aspect-Orientation, exogenous coordinators are introduced transparently to base components. Additionally, the linguistic support for aspect-orientations can also be used to implement adaptors [6,12] and/or to separate other crosscutting concerns, such as Persistence [33] or Scheduling [23], among others [24]. However, at the current moment, Aspect-Orientation should not be considered the Holy Grail to separate easily any kind of crosscutting concern, as there are still some challenges to solve [22,21].

8.2 Aspect-Oriented platforms with coordination support

The first references to coordination as an aspect appears in [18,9,8]. Hernández et al [18] introduce the notion of coordination as an aspect and raises some problems related to its implementation, most of them solved in this paper (with the exception of coordinating state changes that can not be observed externally to the components). Cuesta et al [9] show how to specify coordination as an aspect at the architectural level by means of an aspect-oriented ADL. However, they do not address how to implement coordination as an aspect. Corchuelo et al. [8] present an aspect-oriented language, CAL, for implementing multiparty interaction protocols. Nevertheless, this language is specific for coordination and it can not be used to separate other crosscutting concerns.

A wide study on coordination as an aspect was presented in [10] and [1]. However, this work was based on two specific Aspect-Oriented component platforms: CAM/DAOP [32,31] and MALACA [3]. Both use specific languages to specify crosscutting concerns.

This paper illustrates how coordination can be separated from computation using basic aspect-oriented features, available in all aspect-oriented languages or platforms, without any special linguistic support for coordination. Consequently, special features for coordination, as in CAM/DAOP, MALACA or CAL, are not required. Currently, there is a wide range of aspect-oriented languages available, which target different development needs and domains (real time systems, web services, component systems, application servers). Thus, using the same linguistic support, aspect-orientation, exogenous coordinators can be implemented as an aspect in different application domains with different needs and constraints.

Finally, Capizzi et al. [7] propose an idea similar to this work. They convert endogenous coordination models into exogenous ones, by placing crosscutting code of endogenous models inside aspects. Furthermore, this paper also illustrates how to bridge the gap between design and implementation, providing clear rules for implementing coordination protocols specified by means of state machines.

9 Conclusions

This paper has presented an approach for separating coordination from computation at the implementation level using aspect-oriented programming. It also provides a mapping between a state machines specification of a coordination protocol and aspect-oriented artifacts.

Encapsulation of the coordination concern into an aspect makes system maintenance and evolution easier. Components are also more reusable and are composed more easily. In addition, Aspect-Orientation allows components to be unaware of the existence of coordinator entities, avoiding the necessity of keeping references to external coordinators. Aspect-Orientation composition mechanisms can be non-invasive, so the coordination concern may be added to components without the necessity of modifying their internal structure.

The set of rules for mapping state machines to aspects, provided in this paper, has to be applied manually, which is a tedious, time-consuming and error-prone process. As future work, we will try to automate this process using Model-Driven Development [38] techniques. We will also investigate how aspect-orientation can help to implement other kinds of exogenous coordination models, such as [25,28,4]

Acknowledgement

This work has been supported by Spanish MCYT Project TIN2005-09405-C02-01 and EC Grant IST-2-004349-NOE AOSD-Europe and EC STREP IST-33710 AMPLE. The authors would like to thank Javier Cámara, Javi Cubo and Juan Manuel Murillo for providing useful comments on exogenous coordination.

References

[1] Amor, M., L. Fuentes and M. Pinto, *Coordination as an aspect in middleware infrastructures*, in: *Proc. of the 5th Int. Workshop on Aspects, Components and Patterns for Infrastructute Software (ACP4IS), 5th Int. Conference on Aspect-Oriented Software Development (AOSD)*, Bonn (Germany), 2006.

[2] Amor, M., L. Fuentes and J. M. Troya, *A component and aspect-based architecture for rapid sofware agent development*, in: C. Lemâtre, C. A. Reyes and J. A. González, editors, *Proc. of the 9th Ibero-American Conference on Artificial Intelligence (IBERAMIA)*, LNCS **3315**, Puebla (México), 2004, pp. 32–42.

[3] Amor, M., L. Fuentes and J. M. Troya, *Training compositional agents in negotiation protocols*, Integrated Computer-Aided Engineering **11(2)** (2004), pp. 179–194.

[4] Arbab, F., *Reo: a channel-based coordination model for component composition*, Mathematical Structures in Computer Science **14(3)** (2004), pp. 329–366.

[5] Bonsangue, M., F. Arbab, J. de Bakker, J. Rutten, A. Scutella and G. Zavattaro, *A transition system semantics for the control-driven coordination language manifold*, Theoretical Computer Science **240(1)** (2000), pp. 3–47.

[6] Cámara, J., C. Canal, J. Cubo and J. M. Murillo, *An Aspect-Oriented Adaptation Framework for Dynamic Component Evolution*, Electronic Notes in Theorical Computer Science (ENTCS) (2007), to appear in this number.

[7] Capizzi, S., R. Solmi and G. Zavattaro, *From Endogenous to Exogenous Coordination Using Aspect-Oriented Programming*, in: R. D. Nicola, G. L. Ferrari and G. Meredith, editors, *Proc. of the 6th Int. Conference on Coordination Models and Languages (COORDINATION)*, LNCS **2949** (2004), pp. 105–118.

[8] Corchuelo, R., J. A. Pérez and A. Ruiz-Cortés, *Aspect-oriented interaction in multi-organisational web-based systems*, Computer Networks **41(4)** (2003), pp. 385–406.

[9] Cuesta, C. E., M. P. Romay, P. de la Fuente and M. Barrio-Solórzano, *Coordination as an architectural aspect*, Electronic Notes in Theorical Computer Science (ENTCS) **154(1)** (2006), pp. 25–41.

[10] F. Sanen et al, *A domain analysis of key concerns: known and new candidates*, Technical Report AOSD-Europe-KUL-6, AOSD-Europe Network of Excellence (2006), http://www.aosd-europe.net/deliverables/d43.pdf.

[11] Filman, R., T. Elrad, S. Clarke and M. Akşit, "Aspect-Oriented Software Development," Addison-Wesley, 2004.

[12] Fuentes, L. and P. Sánchez, *AO approaches to component adaptation*, in: *Proc. of the 2nd Int. Workshop on Coordination and Adaptation Techniques (WCAT), 19th European Conference on Object Oriented Progamming (ECOOP)*, Glasgow, (United Kingdom), 2005.

[13] Gamma, E., R. Helm, R. Johnson and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," Addison-Wesley Professional, 1995.

[14] Garcia, A., C. Sant´Anna, E. Figueiredo, U. Kulesza, C. Lucena and A. von Staa, *Modularizing design patterns with aspects: A quantitative study*, in: A. Rashid and M. Akşit, editors, *Transactions on Aspect-Oriented Software Development I*, LNCS **3880**, 2006, pp. 36–74.

[15] Gelernter, D., *Generative communication in linda*, ACM Transactions on Programming Languages and Systems **7(1)** (1985), pp. 80–112.

[16] Hannemann, J. and G. Kiczales, *Design pattern implementation in Java and AspectJ*, in: *Proc. of the 17th Int. Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, Seattle (Washington, USA), 2002, pp. 161–173.

[17] Harel, D., *Statecharts: A Visual Formalism for Complex Systems*, Science of Computer Programming **8(3)** (1987), pp. 231–274.

[18] Hernández, J., M. Papathomas, J. M. Murillo and F. Sánchez, *Coordinating concurrent objects: How to deal with the coordination aspect?*, in: *Proc. of the Int. Workshop on Aspect-Oriented Programming, 11th European Conference on Object-Oriented Programming (ECOOP)*, Jyväskylä (Finland), 1997.

[19] Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. G. Griswold, *An overview of AspectJ*, in: J. L. Knudsen, editor, *Proc. of the 15th European Conference on Object-Oriented Programming (ECOOP)*, LNCS **2072**, Budapest (Hungary), 2001, pp. 327–353.

[20] Kiczales, G., J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier and J. Irwin, *Aspect-oriented programming*, in: M. Akşit and S. Matsuoka, editors, *Proc. of the 11th European Conference on Object-Oriented Programming (ECOOP)*, LNCS **1241**, Jyväskylä (Finland), 1997, pp. 220–242.

[21] Kienzle, J. and S. Gélineau, *AO challenge - implementing the ACID properties for transactional objects*, in: *Proc. of the 5th Int. Conference on Aspect-Oriented Software Development (AOSD)*, Bonn (Germany), 2006, pp. 202–213.

[22] Kienzle, J. and R. Guerraoui, *AOP: Does It Make Sense? The Case of Concurrency and Failures*, in: B. Magnusson, editor, *Proc. of the 16th European Conference on Object-Oriented Programming (ECOOP)*, LNCS **2374**, Málaga (Spain), 2002, pp. 37–61.

[23] Kourai, K., H. Hibino and S. Chiba, *Aspect-oriented application-level scheduling for J2EE servers*, in: *Proc. of the 6th Int. Conference on Aspect-Oriented Software Development (AOSD)*, Vancouver (British Columbia, Canada), 2007, pp. 1–13.

[24] Laddad, R., "AspectJ in Action: Practical Aspect-Oriented Programming," Manning Publications, 2003.

[25] Lau, K.-K., L. Ling and Z. Wang, *Composing components in design phase using exogenous connectors*, in: *Proc. of the 32nd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Cavtat/Dubrovnik (Croatia), 2006, pp. 12–19.

[26] Luck, M., R. Ashri and M. d'Inverno, "Agent-Based Software Development," Artech-House Publishers, 2004.

[27] Meyer, B., "Object-Oriented Software Construction," Prentice Hall, 2001, 2 edition.

[28] Murillo, J. M., J. Hernández, F. Sánchez and L. A. Álvarez, *Coordinated roles: Promoting re-usability of coordinated active objects using event notification protocols*, in: P. Ciancarini and A. L. Wolf, editors, *Proceedings of the 3rd Int. Conference on Coordination Languages and Models (COORDINATION)*, LNCS **1594**, Amsterdam (The Netherlands), 1999, pp. 53–68.

[29] Papadopoulos, G. A. and F. Arbab, *Coordination models and languages*, Advances in Computers - The Engineering of Large Systems **46** (1998), pp. 330–401.

[30] Parnas, D. L., *On the criteria to be used in decomposing systems into modules*, Communications of the ACM **15(12)** (1972), pp. 1053–1058.

[31] Pinto, M., L. Fuentes, M. E. Fayad and J. M. Troya, *Separation of coordination in a dynamic aspect oriented framework*, in: *Proc. of the 1st Int. Conference on Aspect-Oriented Software Development (AOSD)*, Enschede (The Netherlands), 2002, pp. 134–140.

[32] Pinto, M., L. Fuentes and J. M. Troya, *A dynamic component and aspect-oriented platform*, The Computer Journal **48(4)** (2005), pp. 401–420.

[33] Rashid, A. and R. Chitchyan, *Persistence as an aspect*, in: *Proc. of the 2nd Int. Conference on Aspect-Oriented Software Development (AOSD)*, Boston (Massachusetts, USA), 2003, pp. 120–129.

[34] Suvée, D., W. Vanderperren and V. Jonckers, *JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development*, in: *Proc. of the 2nd Int. Conference on Aspect-Oriented Software Development (AOSD)*, Boston (Massachusetts, USA), 2003.

[35] Szyperski, C., "Component Software: Beyond Object-Oriented Programming," Addison-Wesley, 2002, 2 edition.

[36] Truyen, E., *A critical analysis of traditional object-based composition*, in: *Dynamic and Context-Sensitive Composition in Distributed Systems, PhD Thesis*, Department of Computer Science, Katholieke Universiteit Leuven (Belgium), 2004 .

[37] Videira, C. and S. K. Bajracharya, *Assessing aspect modularizations using design structure matrix and net option value*, in: A. Rashid and M. Akşit, editors, *Transactions on Aspect-Oriented Software Development I*, LNCS **3880**, 2006, pp. 1–35.

[38] Völter, M., T. Stahl, J. Bettin, A. Haase and S. Helsen, "Model-Driven Software Development: Technology, Engineering, Management," Wiley, 2006.

[39] Warmer, J. and A. Kleppe, "The Object Constraint Language: Getting Your Models Ready for MDA," Addison-Wesley, 2003, 2 edition.

[40] Yellin, D. and R. Strom, *Protocol specfication and component adaptators*, ACM Transactions on Progamming Languages and Systems **19(2)** (1997), pp. 292–393.