# Aspect-oriented interaction in multi-organisational web-based systems

Rafael Corchuelo , José A. Pérez, Antonio Ruiz-Cortés

*Departamento de Lenguajes y Sistemas Informáticos, Universidad de Sevilla, Avda. de la Reina Mercedes, s/n, Sevilla 41012, Spain*

**Abstract**

Separation of concerns has been presented as a promising tool to tackle the design of complex systems in which cross-cutting properties that do not fit into the scope of a class must be satisfied. Unfortunately, current proposals assume that objects interact by means of object-oriented method calls, which implies that they embed interactions with others into their functional code. This makes them dependent on this interaction model, and makes it difficult to reuse them in a context in which another interaction model is more suited, e.g., tuple spaces, multiparty meetings, ports, and so forth. In this paper, we show that functionality can be described separately from the interaction model used, which helps enhance reusability of functional code and coordination patterns. Our proposal is innovative in that it is the first that achieves a clear separation between functionality and interaction in an aspect-oriented manner. In order to show that it is feasible, we adapted the multiparty interaction model to the context of multiorganisational web-based systems and developed a class framework to build business objects whose performance rates comparably to handmade implementations; the development time, however, decreases significantly.

*Keywords:* Aspect orientation; Interaction models; Multiparty interactions

## 1. Introduction

Distributed object computing is nowadays considered the way forward in devising multi-organisational solutions. Many leading companies have realised that the Internet is more than a sell-and-buy arena and offers many opportunities to thrive in the web world. As the Internet settles, companies are finding ways to take advantage of its capabilities, and there is an ever-increasing demand for frameworks [36–39] to build multiorganisational web-based systems (MOWS) [26–28,49] at sensible costs. Such frameworks typically describe a MOWS as a collection of components that are exposed to programmers as services that are usually implemented as a set of interrelated business objects that encapsulate a semantic protocol for a logical unit of work, e.g., transferring money, processing an order, updating a customer

☆ Supported by the Spanish Interministerial Commission on Science and Technology under grant TIC2000-1106-C02-01.

* Corresponding author.

*E-mail addresses:* corchu@lsi.us.es (R. Corchuelo), jperez@lsi.us.es (J.A. Pérez), aruiz@lsi.us.es (A. Ruiz-Cortés).

record or moving a task to the next person in a workflow queue [36,38]. From an abstract point of view, business objects model real-world actors and data, and this is the reason why they greatly enhance communication amongst developers and customers and help reduce production costs [40,41].

In theory, programmers should only care about the functionality they provide. Unfortunately, this vision is far too idealistic because business objects need to care about concerns such as synchronisation, persistence, or replication, and they are seldom isolated in a system, i.e., they constitute the functional pieces of a semantic protocol and have to work coordinately with others in order to achieve a common goal. Separation of concerns was presented as a promising tool to enhance modularity, understandability and reusability of functional code during the development of a business object. In fact, this is the cornerstone of the new aspect-oriented programming (AOP) paradigm [3,33,34,57], which got inspiration from Dijkstra's and Parnas's early expressions of the seminar principle of software decomposition [31,34,69].

Since the earliest works on separation of concerns, synchronization [61], distribution [61,80], security [83], coordination [5,74], persistence [59], replication [13] and other domain-specific features such as the one described in [50] have been considered as clear concerns that can be dealt with by using aspect languages or frameworks. Unfortunately, prominent aspect-oriented approaches do not succeed in isolating computation from interaction with other objects in a system, i.e., the mechanism by means of which two or more objects get in touch and can carry out an elementary piece of work. As we show in Section 5, current proposals rely on variations of the classical message-passing interaction model and embed object-oriented method calls into the functional code, which prevents developers from adapting their code easily if the interaction model needs to be changed or an object needs to work in contexts in which different interaction models are used. Furthermore, semantic protocols are often hard-wired into the functional core, which makes them difficult to reuse, too [5,46].

Besides point-to-point communication, many other interaction models have been proposed in the literature [68], some of which also provide synchronisation or coordination mechanisms. Each one has its own strengths and weaknesses, so that there is not a universally accepted interaction model, but a wide spectrum of choices. Thus, it seems desirable for aspect-oriented languages dealing with interaction to exist. Such languages would enhance reusability of functional code and coordination patterns, which could be applied to similar situations as long as adequate mapping languages existed. Such languages would also allow programmers to decide which interaction model best suits their needs, which has a direct effect on understandability, maintainability and evolvability. Furthermore, this technology for object interaction should be applied automatically so that business objects remain as much abstract, handy and reusable as possible [5,46]. Therefore, the concerns of abstractness and reusability argue for an aspect-oriented solution to the problem of separation amongst coordination, interaction models and functionality. This way, business objects would not embed interactions into their functionality, and they might be coordinated using different interaction models depending on the context in which they are going to be reused.

This motivation is supported by previous research results by other authors. For instance, in [67], the authors mention that much of the software evolution, reuse and integration are of an unexpected nature. However, this is not necessarily due to a poor design, but rather due to the fact that the world is changing so fast that it is impossible to predict the paths through which software will evolve. Selecting an adequate interaction model is an important decision, and the same functionality encapsulated in a class might be integrated more easily into a scenario using message-passing, but using tuple spaces in another scenario. Thus, having a way to change the interaction model whilst preserving functionality seems to be quite a reasonable motivation.

Furthermore, in [29], the authors presented a case study comparing aspect-oriented technologies. They concluded that they make it easier to write and change some concerns and also that the

underlying design of aspectual applications is more important than the concrete mechanisms provided by those technologies. The latter is an important result, because it is worth noting that separation of concerns has been addressed from points of view other than AOP. For instance, in [10], composition filters are used to compose cross-cutting concerns on multiple objects; in [30], the authors present a proposal based on decomposing software into two levels called aspectual and functional; in [60,81], the authors show two approaches based on reflection capabilities. Generally speaking, separation of concerns can be thought of as an approach that can be beneficial not only at the programming phase, but also at different stages of the software life cycle and at various levels of abstraction [48], including the abstraction level that the concrete interaction mechanism used provides.

These results support our motivation to separate interaction from functionality, which, to the best of our knowledge, is an innovative, original concern that has not been addressed so far in the context of aspect orientation. [1] We illustrate that our idea is feasible by presenting a new interaction model for MOWS called open multiparty interaction model (OMIM). It builds on the basis of a well-known interaction model and enhances it so as to deal with both passive and active objects in an open context. We also provide a class framework to implement business objects using this model, and prove that it does not affect performance but reduces development time significantly. The framework allows to use different algorithms to implement coordination and communication, thus making it possible to customise the underlying implementation.

[1] Notice that we use the term 'interaction' to refer to the underlying mechanism used to stimulate objects, independently from the synchronisation or coordination mechanism used to implement semantic protocols in business objects. This may be confusing if the reader takes a look at old papers on this topic. In [5] or [46], for instance, the authors use the same term to refer to coordination. However, we decided to use the term 'interaction' because it captures the idea behind an object getting or sending stimuli from/to others; coordination concerns are higher level, and it seems to be a well-established research area in which this term is considered rather obsolete.

The rest of the paper is organised as follows: in Section 2, we prove that our idea can go beyond classical object-oriented method calls and present a novel, higher-level interaction model that is adequate in the context of MOWS; in Section 3, we present a class framework we designed to implement our proposal; in Section 4, we show how the development of a MOWS may benefit from using our framework and evaluate it regarding performance and development time; in Section 5, we analyse other authors' work and conclude that none of them attempts to separate interaction from other aspects; finally, we present our main conclusions in Section 6.

## 2. A high-level interaction model for MOWS

Roughly speaking, a MOWS can be viewed as a system composed of several coarse-grained business objects that reside on different organisations and need to cooperate frequently by means of the Internet. They seem to be the cornerstone of next generation distributed software developments.

Client/server primitives such as remote procedure call or message passing are the de facto industrial standard for object interaction in this context, and new materialisations such as SOAP [12,35] are sprouting out at an increasing pace. However, solutions that are based on (sequences of one-way) binary interactions are difficult to apply in typical MOWS in which several objects need to cooperate simultaneously in order to achieve a common goal [44]. Such problems include: transferring money from a bank to another by means of a point of sales terminal (three objects), paying taxes on-line (three objects in Spain: a taxpayer, the Exchequer, and Spain's Certification Authority), filtering in e-commerce [36] (a customer, a filter system, and several service providers), or reaching a virtual agreement in an auction sale (multiple objects). Current technology provides transactional servers for grouping individual binary interactions, but there is not a clear separation between the problem to be solved and the underlying technology to be used or the protocol needed to coordinate the objects involved in a transaction.

Most object-oriented analysis and design methods recognise the need for coordinating several objects and provide designers with tools to model such multiobject collaborations. Different terms are used to refer to them: object diagrams [11], process models [16], message connections [18], data-flow diagrams [78], collaboration graphs [84], scenario diagrams [76] or collaborations [32, 79]. These proposals are accompanied by a rich set of examples that show the adequacy of such modelling tools in fields including finance, telecommunication, insurance, manufacturing, embedded systems, process control, flight simulation, travel and transportation, or systems management. Recently, this need for multiobject interaction has also been recognised in the field of multiagent societies [6,8,9,14,66], and it has also deserved the attention of industrial infrastructure providers such as Microsoft, which provides a tool to orchestrate any group of web services on the Internet as a part of their new .NET platform [75]. The underlying interaction model, however, is object-oriented method calls in all cases, which implies designers still need to decompose complex collaborations into a carefully designed sequence of calls. Several authors have also reported on design patterns and frameworks that allow to model and implement those collaborations as business objects that allow for reuse of common coordination patterns [36–39].

Unfortunately, current programming languages do not seem to have adequate support so that several objects can interact simultaneously, although a great deal of theoretical research has been carried out in this field [68]. JavaSpaces [45] is one of the most prominent recent proposals in this context. It is a Linda-based [15] model for coordinating Java objects on the web that provides a mechanism for storing a group of related objects and retrieving them based on a value-matching lookup for specified fields. It allows the programmer to easily build distributed semantic protocols that can be designed as flows of objects through one or more servers. If your application can be modeled this way, JavaSpaces technology will provide many benefits, but workflow is not enough often, chiefly in the field of MOWS and e-commerce applications. The reason is that many problems in this context require several objects to cooperate simultaneously in order to achieve a global goal, and transforming them into workflow tasks or sequences of method calls may harm expressiveness and thus easy adaptation to new environments.

For instance, assume we have to design a debit-card system [77], which is one of the basic behaviour patterns in a distributed e-commerce application. Such a system is composed of a set of point of sales terminals and a number of computers that hold customer accounts and merchant accounts; the goal is to design a business object responsible for transferring money from a customer account to a merchant account every time a customer pays with his or her debit card. With JavaSpaces, for instance, it is relatively easy to devise a simple solution to solve this problem, but it would have two main drawbacks: first, the whole transaction can be viewed as an atomic event coordinating three objects, but this interaction model leads to a solution in which this event needs to be decomposed into a number of method calls that need to be coordinated by means of a specific protocol based on tuple spaces; furthermore, this protocol may be reusable in other systems, but current technology merges protocols and functionality so much that sorting out the difference is almost impossible [57].

### *2.1. Our proposal*

The concerns presented above argue for an interaction model able to describe coordination patterns amongst an arbitrary number of objects independently from the functionality they implement and the current underlying technology.

In order to devise such an interaction model, we got inspiration from the well-known multiparty interaction model [44], which is well suited to capture the essence of problems in which several objects residing on different machines need to cooperate coordinately and simultaneously. It provides a higher level of abstraction because it allows them to exchange data and perform some joint actions coordinately without taking implementation details into account. Multiparty interactions are higher level in the sense that they abstract the

concrete mechanism used to synchronise, coordinate or communicate several objects simultaneously. They thus make it feasible to customise the implementation of multiobject interaction since the low-level coordination protocols are no longer hard-wired into business objects, but specified by means of high-level language constructs. Multiparty interactions hide several underlying point-to-point communication operations, as well as the order in which they must occur or the algorithm used to achieve multiparty synchronisation. The programmer does not need to care about these low-level details, which can be generated in a completely automatic way, thus easing maintainability.

Although this interaction model has attracted the attention of many researchers who have focused on implementation issues [1,22,24,25,43,44,51,53,85], it suffers from several drawbacks that need to be addressed before using it in the context of a MOWS. In particular, the model does not allow for passive objects, objects participating in an interaction need to be known at compile time, and they need to have access to the local state of the objects with which they communicate. That is the reason why we have developed an enhanced version that addresses these problems and solves them adequately. We refer to it as OMIM.

In order to support OMIM, we have designed a language called CAL [23] and a set of algorithms that allow to implement it quite efficiently [71–73]. In the following subsections, we summarise our results so that the reader may have an overall idea of the work we have already done. In Section 3, we report on the framework we have built, which is one of the contributions of this paper. It integrates these algorithms and proposals by other authors, thus proving that interaction may be separated from functionality; the result benefits from enhancing reusability and allows to customise the implementation by quickly changing the algorithms used to implement a system. The impact on the development time is quite significant, as shown in Section 4.3.

### 2.2. Linguistic support

In this section, we glance at CAL and describe its main features by means of the debit-card system presented previously. This problem can be easily described by means of multiparty interactions because a three-party interaction needs to be carried out when a clerk inserts a debit card into a terminal in order to transfer funds from a customer's account to a merchant's account. Fig. 1 shows a description of the debit-card system in CAL, and it is analysed in the following subsections.

#### 2.2.1. Describing interactions

Interactions are defined by means of the following syntax:

```
interaction ⟨name⟩[⟨participants⟩](⟨slots⟩)
where ⟨read/write permissions⟩
```

Each interaction is given a different name, a number of participating objects, a number of slots, and some read/write permissions. In the example in Fig. 1, an interaction called transfer has been defined, and it is a three-party interaction that allows to interact a terminal that plays role term and two bank accounts that play roles source and dest. This interaction is intended to be the channel by means of which they can coordinate so that funds can be transferred from the source account to the destination account.

Interactions are equipped with a local state that is composed of several slots. In our example, interaction transfer has two slots called sum and approval. sum is used to store the amount of money to be transferred, and approval is a flag that indicates whether the source account can transfer such a sum to the destination account. These slots make up a local state that simulates the temporary global combined state in the basic multiparty interaction model [44], being the most important difference that an object does not need to have access to the local state of other objects in order to get the information it needs.

The read/write permissions state which participant in an interaction can read and/or write each slot. In our example, the terminal is responsible for storing the sum to be transferred in slot sum, whereas it only reads slot approval in order to display a message on its screen; the account playing role source can read slot sum to decide whether it can transfer such a sum, and it can write slot

```
interaction transfer[Terminal term; Account source, dest]
   (float sum; boolean approval) where
      term writes sum, reads approval;
      source writes approval, reads sum;
      dest reads approval, sum;

behaviour Terminal requires
 interface ITerminal;
{
 *[
  Wait_For_Sale();
  -- Try to engage interaction 'transfer' together with the customer's account
  -- as source and the merchant's account as destination
  transfer[term!self, source!Get_Customer_Account(), dest!Get_Merchant_Account()]
  {
      sum = Get_Price();
      Report_Result(approval);
  }
 ]
}

behaviour Account requires
 interface IAccount;
{
 *[
  -- Behave as a source account: Try to engage 'transfer'
  -- together with any terminal or destination account
  transfer[source!self]
  {
      approval = Authorise_Payment(sum);
      [approval → Charge(sum);];
  }
  []
  -- Behave as a destination account: Try to engage 'transfer'
  -- together with any terminal or source account
  transfer[dest!self]
  {
    [approval → Pay_In(sum);];
  }
 ]
}
```

Fig. 1. A description of the debit-card system in CAL.

approval to store its decision; finally, the destination account can read both slots, but it is not allowed to write any of them.

Every object can offer participation in one or more interactions simultaneously. In every offer, a participant states which role it plays in the interaction, and may establish constraints on what objects should play the other roles. An interaction may be executed as long as a set of objects satisfying the following constraints is found: (i) there is an object per role willing to participate in that interaction and to play that role; (ii) those objects agree in interacting with each other, i.e., the constraints they establish are satisfied. A set of objects which can execute an interaction is what we call a enablement.

Since exclusion must be guaranteed, an object cannot commit to more than one interaction at a time. However, since an object can offer participation simultaneously in more than one interaction, it can be in more than one enablement. So, when two or more enablements share objects, they cannot be executed simultaneously. The set of enablements that cannot be executed are said to be refused.

*2.2.2. Describing behaviours*

Each interaction requires a number of objects, and they must behave the right way. We use the following syntax to describe behaviour patterns:

```
behaviour ⟨name⟩
   requires ⟨interfaces⟩
{
   ⟨behaviour statement⟩
}
```

Each one is given a different name, and requires a number of operations (i.e., an interface) to be implemented by the objects onto which it can be mapped. In the example in Fig. 1, two behaviour patterns are described: Terminal, which describes the behaviour of a terminal, and Account, which describes the behaviour of a bank account, both as debtor and creditor.

Pattern Terminal requires that the operations in interface ITerminal (Fig. 2) be implemented by the objects that can behave the way it describes: Wait_For_Sale, which encapsulates the details concerning waiting for a new sale and interactions with the clerk initiating it, Get_Price, which can be invoked after a new sale has been initiated and reports its price, Get_Customer_Account and Get_Merchant_Account, which provide references to the involved accounts, and Report_Result, which can be invoked to report whether a transfer has been done or not. Similarly, pattern Account requires that the three operations in interface IAccount be implemented: Charge, to withdraw money from an account, Pay_In, to pay money into it, and Authorise_Payment, which decides whether an account can afford a payment or not.

The operations required by a behaviour pattern are the operations whose execution is coordinated by means of multiparty interactions. In order to model how a terminal or an account cooperate, we use interaction statements of the form I[expression_list]{comm_stat}, where I is the name of an interaction, the expressions within brackets identify the objects with which the object executing such a statement is interested in cooperating, and comm_stat is a communication statement involving the slots of interaction I. The expressions between brackets are of the form role!id, which means that role role must be played by the object identified by id. The expression role!self is used to denote the role played by the object that issues the offer. If no expression is given for a role, it means that it can be played by any object.

Therefore, the behaviour of a terminal can be summarised as follows: it is an infinite loop where it first waits for a new sale operation to begin and then tries to engage interaction transfer together with the objects that model the customer's account and the merchant's account. If this interaction is fired, the terminal participating in it executes then its communication code, which consists of storing the sum to be transferred in slot sum, and displaying a message on its screen. The behaviour of an Account also consists of an infinite loop where engaging interaction transfer is offered either as a source account or a destination account. If the interaction where an account plays the first role is fired, it checks whether it can afford the charge, stores the result in slot approval and updates its balance accordingly. If the interaction in which it

```
interface ITerminal {
  void Wait_For_Sale();
  float Get_Price();
  OID Get_Customer_Account();
  OID Get_Merchant_Account();
  void Report_Result(boolean done);
}
```

```
interface IAccount {
  void Charge(float sum);
  void Pay_In(float sum);
  boolean Authorise_Payment(float sum);
}
```

Fig. 2. Interfaces required the by behaviours in Fig. 1.

plays the other role is fired, it simply reads slot approval and then updates its balance accordingly.

Obviously, a multiparty interaction delays an object that tries to read a slot that has not been initialized yet, e.g., a terminal that executes the statement Report_Result(approval) is delayed until the source account participating in the same interaction has written slot approval. Thus, the communication statements are executed in a critical region where no race conditions can occur.

*2.2.3. Mapping behaviours onto object classes*

Behaviour patterns are abstract because they describe how an object that implements a set of operations cooperates with others. These operations are also abstract, and they usually need to be adapted when we need to map a behaviour onto an object class.

CAL provides a simple mechanism for adapting operations, and it is shown in Fig. 3. In this example, behaviour Account has been mapped onto a Java class called bankAccount, but it does not provide the operations this pattern requires. For instance, there is not an operation for deciding whether charging a sum is affordable or not. Fortunately, the expression getBalance( ) $\geqslant$ sum implements it easily.

This mapping allows us to write fully abstract, reusable behaviour specifications. This is important, because in other aspect languages such as COOL [61], RIDL [61], AspectJ [56] or AML [50], aspect specifications reference the classes onto which they are applied by name, which makes it difficult to reuse them effectively.

```
class bankAccount  {
  private float balance;

  public void updateBalance(float sum) { balance += sum; }
  public boolean getBalance() { return balance; }
}

map behaviour Account onto class bankAccount where
  Charge(sum) = updateBalance(-sum);
  Pay_In(sum) = updateBalance(+sum);
  Authorise_Payment(sum) = (getBalance() ≥ sum);
```

Fig. 3. Mapping behaviour Account onto class bankAccount.

## 3. The framework

We have carefully designed a framework that offers a number of high-level services for implementing CAL. [2] We have arrived at its design largely through experimentation, and our main design goals were the following:

(1) The framework should be extensible, so that new middlewares or coordination algorithms may be easily incorporated. This enhances the choices the designer has to produce the final version of his or her business objects independently from the functionality he or she has programmed.
(2) It should allow for passive objects. The traditional multiparty interaction model assumes that every object participating in a multiparty interaction is active, i.e., they can autonomously offer to participate in an interaction. However, objects are passive in many real-world problems, i.e., they offer interaction on demand; thus, a good framework should be able to deal with them without compromising effectiveness.
(3) It must allow to describe other orthogonal aspects using well-known aspect-oriented languages.

Fig. 4 shows a snapshot of a running system that sketches the architecture of our solution. It is composed of the following elements:

*The gatekeeper:* It is one of the most important components of our architecture because it is responsible for tasks such as security policies, billing, generating and managing UUIDs, locating interaction coordinators or interacting with the system administrator.

*Interaction coordinators:* They are responsible for detecting enabled interactions and arbitrating amongst conflicting ones, i.e., interactions that cannot be executed simultaneously because they share a common object.

[2] The framework is available on request. Please, send mail to jperez@lsi.us.es to ask for the package.

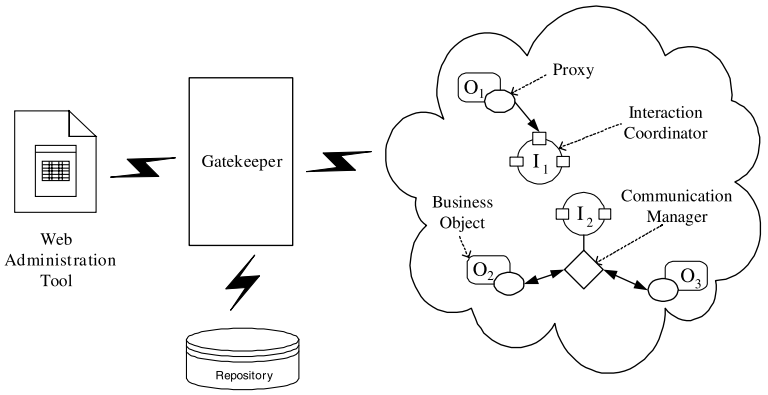


Fig. 4. The architecture of our solution.

*Proxies:* In our framework, user objects are considered to be external entities that use proxies to interact. This makes a clean separation between functionality and coordination details and simplifies the framework implementation because it does only need to care about proxies, independently from the objects they represent. This implies that those objects may have pure functionality or may have been woven with other aspects previously. For instance, an object having synchronisation or replication constraints may be implemented by weaving functional code with a COOL [61] specification and the proposal described in [13].

*Communication managers:* They are responsible for managing communication amongst a number of objects that have committed to an interaction. This way, many different occurrences of the same interaction may be running simultaneously and independently from each other. Communication managers are also responsible for coping with faults during multiparty communication [85].

At a first glance, it might seem that the gatekeeper is a bottleneck component, but it is not. The reason is that the functionality it offers is used only when new objects or interactions are added to the system, or when an object needs to fetch references to the coordinators responsible for the interactions in which it may be interested. It is also worth noting that nothing prevents us from creating several instances of the gatekeeper, thus reducing the impact of a crash. However we usually refer to this component as 'the gatekeeper' because all of its instances are functionally equivalent.

It is also worth mentioning that having proxies does not amount to inefficiency because they reside in the same memory space as the objects they represent. Furthermore, separating coordination concerns from business objects at runtime is worthwhile because this draws a clear line between the functionality they encapsulate and the way they interact with others. This facilitates the construction of a weaver because proxies may be implemented using a combination of design patterns that are collectively called the role object pattern [38]. This helps keep the different contexts in which a business object may participate separated.

Fig. 5 shows the class framework we have designed. Notice that our design is general enough to accommodate several middlewares as well as several coordination or communication algorithms proposed in the literature. The designer may make a choice depending on the application domain and the problem to be solved. If all of the objects know the objects with which they have to interact, the designer might use a classical algorithm for coordinating them; however, if some objects are selected at runtime, the designer may use the algorithms we have implemented to deal with open multiparty interactions [71,73]. It is worth mentioning that proxies implement two interfaces because they have to communicate with both objects and coordinators. We have split their interface into IPartProxy, which groups the operations an object needs, and ICoordProxy, which groups the operations coordinators need.

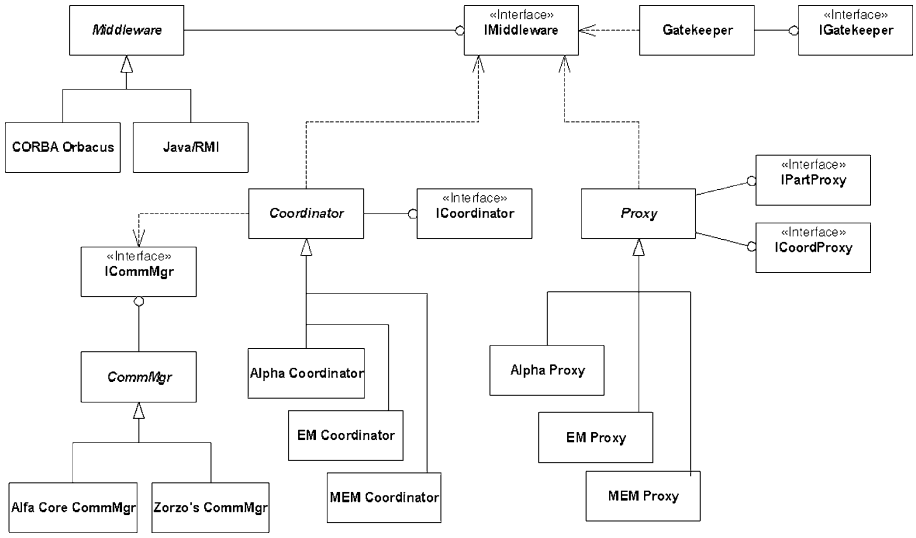


Fig. 5. Class framework.

### 3.1. Coordination algorithms

Several solutions to implement multiparty interactions have been proposed in the literature. We have found a variety of centralised and distributed techniques for implementing this interaction model, but, unfortunately, most of them have been devised to deal with a fixed set of objects [7, 17,24,25,52,54]. Although they may work well in some MOWS, it is problematical insofar the services business objects offer may be used by a dynamic set of objects.

An important feature of a good business framework is that it must allow for evolution [38]. Thus it is needed a better solution allowing for open interactions in which the set of participating objects is not known until runtime. We have designed an algorithm for dealing with multiparty coordination in this context, and we refer to it as $\alpha$. It is responsible for two main tasks: (i) detection of interacting groups, i.e., groups of objects that are interested in participating in the same interaction, and (ii) arbitrating amongst conflicting interactions that share a common object. Therefore, $\alpha$ was split into two parts referred to as $\alpha$-Solver [72] and $\alpha$-Core [71,73].

We sketch the main ideas behind $\alpha$-Solver by means of a simple example depicted in Fig. 6. Assume that our system has an interaction called I, and that it coordinates three active objects that may play roles P, Q and R. Let us also assume that objects $p_1$ and $p_2$ are interested in playing role P, objects $q_1$ and $q_2$ offer to play role Q, and object $r_1$ offers to play role R. The offers are represented by means of arrows from proxies to coordinators, and each one is labeled by the list of expressions that identifies the participants that may engage the interaction. For instance, the expression $[P!self,\ Q!q_2]$ offered by $p_2$ means that this object is willing to play role P in interaction I, that it requires object $q_2$ to play role Q, but does not care about the object playing role R.

$\alpha$-Solver processes the offers as they arrive and builds the consolidation graph in Fig. 7 incrementally. This graph is used to detect groups of objects that are willing to participate in the same interaction. For instance, assume that the offer made by $p_1$ arrives first so that $\alpha$-Solver constructs a consolidation graph with only one node $[p_1, (q_1), (r_1)]$ that is interpreted the following way: there is an offer in which object $p_1$ is willing to play role P, object $q_1$ is required to play role Q and object $r_1$ is required to play role R. If the second offer is made by object $p_2$, a new node of the form $[p_2, (q_2), (\ )]$ is added to the graph, but no connecting node is constructed because the

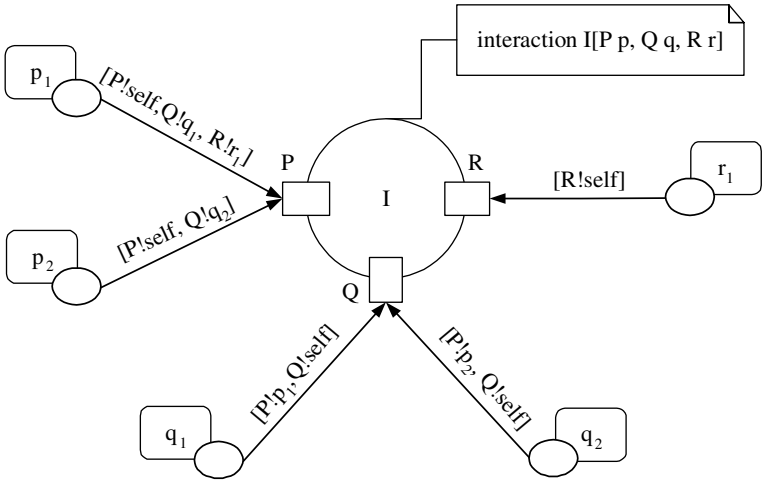


Fig. 6. A simple system.

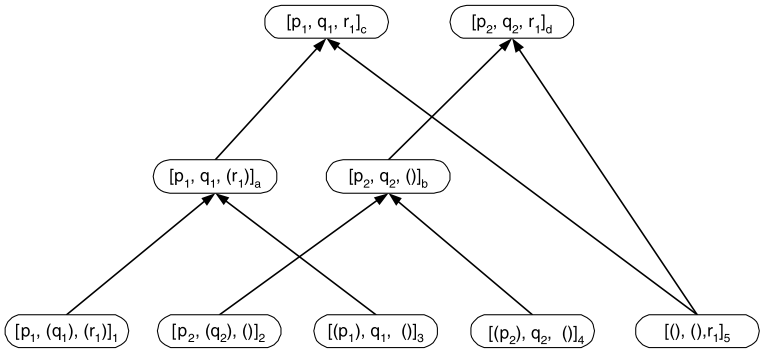


Fig. 7. Consolidation graph for the system in Fig. 6.

tuples so far processed cannot be consolidated, i.e., objects $p_1$ and $p_2$ cannot interact together.

If the offer made by $q_1$ is then received, a node of the form $[(p_1), q_1, ( )]$ is added. Since it consolidates with $[p_1, (q_1), (r_1)]$, a connecting node of the form $[p_1, q_1, (r_1)]$ is added. It indicates that both $p_1$ and $q_1$ are willing to participate in interaction I and agree in committing to this interaction together with object $r_1$. Notice that no interaction group is found until object $r_1$ makes its offer. When this happens, two interaction groups are found simultaneously, but, unfortunately, they are conflicting because they share a common participant. Thus, α-Core is called to arbitrate amongst these conflicting groups and decide which one commits to interaction I first.

The idea behind α-Core is quite simple because shared participants are considered to be shared resources amongst the coordinators responsible for the interactions in which they are interested. In order for an interacting group to execute an interaction, α-Core must ensure exclusive access to all of the objects participating in that interaction. The algorithm we use to lock objects is based on a simple idea that was presented years ago in the field of operating systems [19]: α-Core locks objects in order of increasing UUID. Although this idea did not work well in the field of operating systems because processes are difficult to programme so that they request resources in increasing order, it has been proven to be quite effective in this context (cf. Section 4.3).

### 3.2. Coordinating passive objects

In previous section, we presented a system whose objects were assumed to be active, i.e., their proxies send messages to coordinators responsible for the interactions in which they are interested. However, objects in MOWS tend to be passive in the sense that they offer services on request, and each interaction has a relatively small set of active objects. For instance, it is likely that both accounts in the debit-card system are passive and do not try to engage any interaction actively.

Our framework can deal with such objects using a simple protocol: each time an active object initiates an interaction in which a passive object may participate, the runtime system sends a notification to the proxy associated with that object; if it can participate in that interaction, the proxy then makes an offer, thus behaving as if its object was active; otherwise, it stores the notification and reprocesses it each time the passive object it represents changes its state. If one of the active objects willing to participate in that interaction gives up, the coordinator also notifies the involved passive objects and their proxies stop monitoring them.

For instance, assume that objects that play roles P and Q in the system in Fig. 6 are active and objects that play role R are passive. On reception of the offer made by $q_1$, the coordinator of interaction I adds the connecting node $[p_1, q_1, (r_1)]$ to the consolidation graph and knows that $p_1$ and $q_1$ are willing to interact as soon as $r_1$ is ready to play role R. Thus, the coordinator searches for such an object and notifies its proxy that it is requested to participate in interaction I. The proxy then stores this notification and waits for its object to reach a state in which it can participate in this interaction. If its current state allows it to participate, the notification is sent immediately.

Sometimes, many passive objects reside on the same machine. In those cases, the designer may decide to assign a set of objects to the same proxy in order to improve scalability. Passive objects usually reside on a database, so that having an individual proxy responsible for each one may degrade performance significantly. In those cases, it is better to use a small set of proxies managing the whole set of objects in a class. The ratio between proxies and objects depends completely on the context, and it is a trade-off between reliability, performance and scalability.

## 4. Realising the benefits

There are three usual approaches to separate concerns, and we refer to them as linguistic, object-oriented, and architectural. In [29], the authors report on the results of a case study in which they compared these approaches and concluded that each one has its own strengths and weaknesses so that no-one clearly outstands globally. The linguistic approach is quite efficient in general but cannot deal with aspect evolution easily because aspects tend to be somewhat linked to language constructs; contrarily, object-oriented approaches that are based on the reflective capabilities of the underlying language are usually quite flexible but inefficient.

Our proposal consists of both a language to describe business objects separately from the functionality of base objects and a framework that can be used to implement this language or business objects independently. It thus benefits from the strengths of both proposals simultaneously, as we show next.

### 4.1. Changing the interaction model

One important feature of our proposal is that it allows to reuse objects that implement functionality easily, even if the underlying interaction model is changed. We illustrate this by means of an example in which we reuse class bankAccount in an environment in which JavaSpaces [45] is the interaction model used.

Fig. 8 presents the Java code needed to reuse class bankAccount so that some accounts are charged commission. commissionCharger is a business object that monitors tuple space commissionSpace. Every time a tuple of the form comission(oid, p) appears in this space, it fetches the object of class bankAccount identified by oid and charges it percentage p.

This example shows that the basic functionality in class bankAccount can be reused effectively in

```
public class commission implements net.jini.core.entry.Entry {
   public String OID;
   public Float percentage;

   public commission() {}
}

public class commissionCharger {
  protected bankAccount locateAccount (String OID)
  { ...Locate a bank account through its OID... }

  public void run()
  {
    JavaSpace space = spaceAccessor.getSpace("commissionSpace");
    commission orderTemplate = new commission();
    commission newOrder = null;
    bankAccount account = null;

    try
    {
      while (true)
      {
        newOrder = (commission)space.take(orderTemplate, null, Long.MaxValue);
        account = locateAccount(newOrder.OID);
        if (account.getBalance() > 0)
          account.updateBalance(-account.getBalance() *
                                newOrder.percentage.getFloatValue()));
      }
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }
}
```

Fig. 8. A business object to charge commissions using JavaSpaces.

contexts other than transfers as long as it is kept independent from the underlying interaction model used.

### *4.2. Integration with other orthogonal aspects*

The language we have designed can be easily integrated with other aspects as shown in Fig. 9. The idea is to weave aspects such as intra-object synchronisation or replication using specific languages so that the class onto which we apply a behaviour designed using CAL has already been woven with the aspects under consideration. For instance, Fig. 10 shows a COOL specification that states that no two concurrent threads may be executing method updateBalance simultaneously or concurrently with method getBalance. We can use the COOL weaver to merge this specification into class bankAccount so that the resulting class has intra-object synchronisation. Similarly, we can use the AspectJ advices reported in [13] to add replication capabilities to our class smoothly.

Very little work has been reported in the literature about dealing with aspects that are not orthogonal, i.e., aspects that depend on each other or conflict each other. Refs. [30,70,82] are amongst the very few papers on this topic, but they do not attempt to provide a solid characterisation of

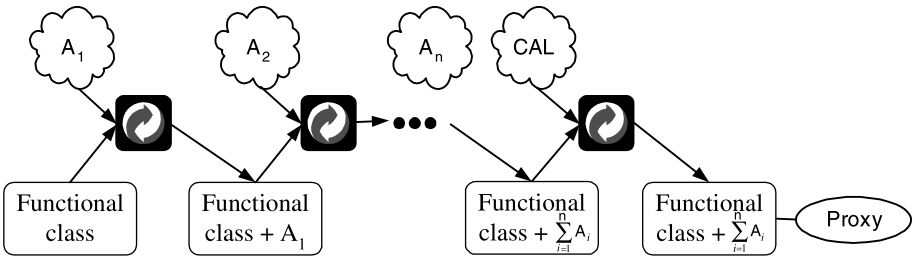


Fig. 9. Weaving orthogonal aspects with CAL.

```
coordinator bankAccount {
  selfex updateBalance;
  mutex {updateBalance, getBalance};
}
```

Fig. 10. An orthogonal aspect that can be woven with a CAL behaviour.

non-orthogonal aspects and how they can be dealt with. They only report on how to deal with some conflicting aspects using a given technology, and the authors conclude that integrating such aspects remains an open research field. In such cases, the classes that compose the framework can be extended to deal with such concerns, but this obviously depends completely on the concern to be added.

### *4.3. Empirical results*

According to [62,63], AOP and other techniques can be evaluated by using synthetic experiments. Next we present our results and main conclusions from an experimental study we conducted in order to evaluate our proposal. They prove that it is feasible, its efficiency compares to handmade implementations, and reduces the development time significantly.

Our experiments were implemented by several independent teams composed of three students each. The students were selected from a four-level course according to the Personal Software Process Tests [42,47,55], which implies they were quite homogeneous. All the students had good programming skills using Java technologies, so they only had to be trained in using our framework and AspectJ. The programmes were run on a 10 Mbps Ethernet network in which every object run independently from the others on its own computer. The machines we used were equipped with Pentium 200 MHz processors, 64 MB of RAM memory, and run Windows NT 4.0, Sun's JDK 1.2.1, and Orbacus 3.3.2.

Regarding our synthetic experiments, we focused on several well-known coordination patterns presented in [44], and the debit-card system presented in Section 2.2. Table 1 summarises the patterns we used and the keys by means of which we refer to them in the following figures.

After training our students, we selected some of them and divided them into 12 groups of three students each. They implemented the above-mentioned coordination patterns, the difference being that Teams 1–4 used our framework (Group A), Teams 5–8 used AspectJ and CORBA (Group B), and the others used standard Java and CORBA (Group C).

Fig. 11 reports on the efficiency of the implementation each group produced. We measured the average number of interactions per second each implementation achieved during a run that was long enough to complete 10,000 interactions; the figure shows the average efficiency in each group. This metric is meaningful because the standard deviation did not exceed 5.4%. The average number of interactions per second our framework achieves using the algorithms we outlined in Section 3.1 is 163.12, which compares very well to handmade implementations using AspectJ or standard Java. The penalty our framework introduces is about 0.62% with respect to AspectJ, and 5.34% with respect to standard Java. This penalty seems reasonable and compensates for the reduc-

Table 1
Coordination patterns used to test our proposal

| Key | Description |
|---|---|
| DCS | The debit-card system presented in Section 2.2 |
| LE(n) | The Leader Election problem, which is very usual in round-based auctions on the Internet. The number of auctioneers is given in parenthesis |
| DF(n) | The Dining Philosophers problem, which is the paradigm of those situations in which an object needs to get exclusive access to several objects simultaneously. The number of philosophers is given in parenthesis |
| MM | The Matrix Multiplication problem, which is a typical example of a task that needs to be divided into several subtasks on which several workers work loosely coupled until the results need to be aggregated. This is a typical pattern in systems that need to search for goods or services using several providers [36] |
| TH | The Towers of Hanoy. Although this pattern is not likely to arise in practice, it is a good example of a problem in which a relatively small number of objects need to interact very frequently using complex coordination patterns. We used it to test our implementation in an extreme situation |

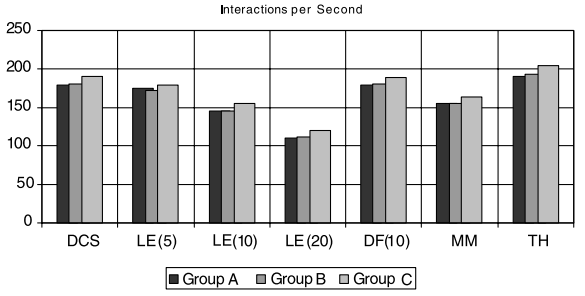


Fig. 11. Performance of our framework.

tion in development time. We evaluated the time our students took to complete their assignments and show the results in Fig. 12. As the figures prove, using the framework amounts to 121.43% less development time than using AspectJ, and 114.29% less development time than using standard Java.

From these results, we conclude that using AspectJ to solve specific problems in the field of coordination does not entail a significant reduction of the time needed to develop a project. We think that the reason is that AspectJ does not provide programmers with specific constructs to deal with this aspect, so they have to spend almost the same time at implementing it than using standard Java. It is also worth mentioning that the implementation could not be reused, and that the students soon began arguing for a specific language able to deal with coordination.

We followed the guidelines in [63], and also conducted some 'debugging and change' experiments. The first one consisted of seeding each implementation with an error that the students had to locate an solve. Each team worked on an implementation they had not developed to avoid interferences and simulate that they have to solve a problem encountered on a programme they have not developed. As Fig. 13 shows, teams using our framework could correct the error 44.44% faster than the teams working with AspectJ and 66.67%

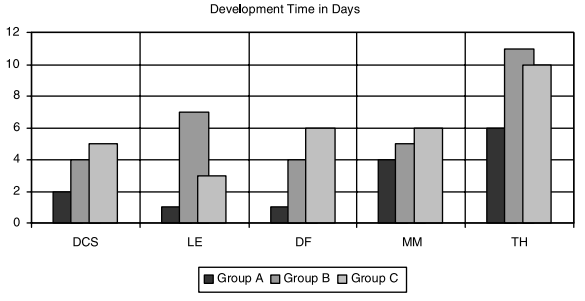


Fig. 12. Average development time.

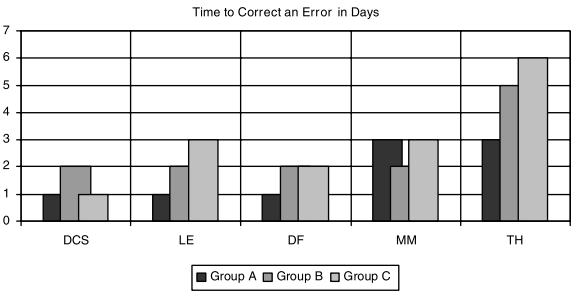


Fig. 13. Time to correct an error.

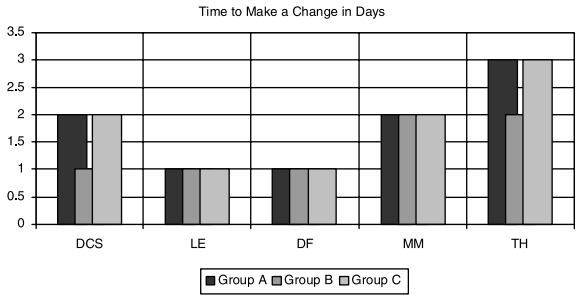


Fig. 14. Time to make a change.

faster than the teams working with standard Java. The experiments were repeated introducing errors in the semantic protocol, the base class, and the coordination pattern, and they all concluded in similar results. Furthermore, we noticed that the teams that used AspectJ or our framework had less discussions about the semantics of the base code, which was expected according to the results reported in [63].

We also measured the time our teams took to change their code at different levels. In this case, the teams using AspectJ or our framework did not took significantly less time than the teams using standard Java, as shown in Fig. 14. We think that the reason may be that programmers that use aspect-oriented technologies tend to think that changes can be largely dealt with by using the aspect language. That is the reason why they did not take enough time to analyse the code before they began making changes. Notice that some teams in Group B outperformed the other teams in some cases. However, we do not think this difference is important because the time they took was almost the same in average. There is not a concluding reason to explain why those teams excelled at changing their code.

## 5. Related work

Coordination is a concern that has attracted the attention of many researchers in the aspect-orientation arena. It has been dealt with from very different points of view, so there is a wide variety of related proposals in the literature. In this section, we summarise and compare some of the work that has been done, and conclude that none of them has addressed a clear separation between the interaction model used to get objects in touch and other aspects.

One of the first proposals that considered coordination as a cross-cutting aspect was COOL [61]. In COOL, a special construct called coordinator can be associated with each class (one coordinator per class) describing a synchronisation policy. The coordinator of a class describes the possible states of the class from the coordination point of view, how the state changes when methods of the class are called, and establishes constraints on how a method call is executed depending on the state.

COOL deviates from our proposal in that it does only deal with intra-object coordination, i.e., protocols that control whether a method call needs to be delayed until a synchronising condition holds. Therefore, coordination amongst a number of objects needs to be coded manually into the functional code. Furthermore, COOL assumes that the underlying interaction model is method call, which makes it difficult to change it.

Formerly, in [46], the authors presented a proposal called language framework for multiobject coordination (LFMOC). It uses multiobject constraints implemented by synchronisers that allow to describe coordination patterns. [3] Conceptually, a synchroniser is an object that constraints the invocations accepted by a set of ordinary objects that are being constrained. They allow to enforce temporal ordering constraints and atomicity on groups of invocations. Consequently, this proposal improves COOL in that it takes coordination amongst several objects into account and can also deal with atomicity.

Abstract communication types (ACT) [5] is a proposal similar to LFMOC. It is based on composition filters [4], which are used to design meta-objects that control how a set of methods in one or several objects get synchronised. LFMOC and

[3] Notice that the authors use the term 'interaction' to refer to coordination, but not the mechanism by means of which two or more objects can stimulate each other.

ACT are similar in spirit, but differ in that the language composition filters provides is much richer than the language LFMOC provides. However, LFMOC allows to generate low-level protocols automatically, and they can be customised according to specific-domain constraints. Contrarily, low-level protocols must be implemented in composition filters, but they can be reused later in similar situations.

Our proposal is similar to both LFMOC and ACT in that it also allows to coordinate groups of objects without hard-coding low-level protocols into the functional code. However, it differs in that semantic protocols in both LFMOC and ACT need to be hard-wired by means of sequences of method calls that are synchronised by the mechanisms they provide. Objects are thus dependent on the interaction model used, which is not easy to change once an object encapsulates a part of a semantic protocol.

Another interesting approach can be found in [74]. The authors present a dynamic aspect-oriented framework (DAOF) which can deal with different concerns by means of method call adaptation. In DAOF, aspects are first-order entities that can be composed at runtime by using the information stored in a middleware layer. To deal with an aspect, it must be encapsulated into a handmade class and registered in the middleware, which is responsible for applying them in a given order each time it intercepts a call to an object. The proposal is illustrated by means of the coordination aspect, which is very similar to LFMOC or composition filters. In DAOF the coordination aspect is implemented by means of an object that encapsulates a piece of logic that allows to monitor and control the invocations of a method in an object. However, it is not clear whether the authors can address multiobject synchronisation. Contrarily to LFMOC and our proposal, DAOF does not provide an automatic mechanism to generate low-level protocols automatically.

DAOF is similar in spirit to the aspect moderator framework (AMF) [20,21], which was recently enhanced and transformed into the layered aspect moderator framework (LAMF) [64,65]. The LAMF is an architectural proposal that aims at decomposing systems into a number of well-defined layers composed of functional components and aspectual properties controlled by an aspect moderator, which is similar in essence to the objects that implement aspects in DAOF.

Although the layered approach greatly enhances decomposition and abstraction, aspects need to be coded manually and coordination is not addressed explicitly, although it can be incorporated similarly to DAOF. Neither does this framework achieve a clear separation between functional code and the underlying interaction model, which is also assumed to be object-oriented method calls. In fact, the framework works by intercepting method invocations at runtime and deciding which aspects need to be applied before the method begins executing. Thus, semantic protocols also need to be hard-coded into the functional code, whereas low-level protocols are coded into specific classes and thus kept separated.

Meta-object protocols (MOP) [81] is a recent proposal that aims at adding aspects to an object by using a limited set of reflection capabilities that allow programmers to modify how an object behaves at runtime. The proposal is similar in spirit to COOL and allows to implement this language efficiently. However, the aspects that can be integrated at runtime are not limited to synchronisation because they are coded using the same programming language. Although this proposal allows to integrate multiple aspects, they need to be coded manually, and does not allow for aspects such as coordination because they cross-cut the boundaries of several classes. The underlying interaction model is also assumed to be object-oriented method calls.

MOP builds on the spirit to languages such as MAUD [2], which provide full reflection. However, full reflective languages have complicated semantics and may bring unnecessary additional complexity [5], which argues for more effective solutions such as MOP. Each object in MAUD, for instance, has three meta-objects, namely a dispatcher, a mail queue and acquaintances. The messages sent and received are handled by the dispatcher and mail queue objects, respectively. The acquaintances object contains a list of objects

that may be addressed by its owner. In MAUD, one can implement coordinated behavior by replacing the meta-objects with the objects implementing the required protocol. To install a protocol for an object, the original mail queue and dispatcher must be replaced by a pair implementing the required protocol. In MAUD, a shared protocol amongst objects is implemented by mail queues and dispatchers. Coordinated behavior is distributed amongst mail queue and dispatcher objects which are added to all participating objects.

In a sense, our proposal resembles MAUD in that we attach an object called proxy to each user object. The main difference is that we do not require the underlying language to be reflective, which has a direct effect on efficiency, and the semantic protocol is encapsulated in an independent business object that can be constructed using the most appropriate interaction model in each context. Unfortunately, MAUD objects assume that the interaction model is message passing, so that even if the meta-objects are changed, they still communicate and embed semantic protocols by means of sequences of messages.

The proposal described in [60] also deserves some attention. The authors present a solution called adaptive methods (AM) to deal with crosscutting concerns in scenarios in which a set of interrelated objects need to carry out a common task. It consists of a Java library that allows to implement traversal strategies [58]. Such strategies allow to describe high-level cooperations as a high-level description of how to reach the participants of a computation, plus a description of what to do when each participant is reached. Although AM might seem similar to ours, it is completely different because AM are targeted towards designing workflow algorithms in which participants are designed carefully to accomplish a part of the workflow. No real coordination is needed amongst a set of distributed objects, just a description of how to reach each participant and the work it has to do. As the authors point out, AM can only deal with a limited class of behavioural concerns and coordination of objects that run independently does not seem to be amongst them, except for the case of workflow coordination.

## 6. Conclusions

In this paper, we have explored the aspect-oriented paradigm, the multiparty interaction model, and how programming distributed systems may benefit from both. Separating computation from interaction encourages reuse, improves comprehension, and eases maintenance and evolution of software because classes that are purely functional can be easily reused in contexts in which specific interaction models are more suited than object-oriented method calls. We have also shown that the proposal is feasible, and the efficiency of the framework we have presented rates comparably to handmade implementations, but reduces development time significantly.

To the best of our knowledge, this piece of work is novel in that none of the references consulted aims at separating the underlying interaction model from other aspects. It is assumed that objects communicate by means of method calls, which make them dependent on this interaction model and makes it difficult to reuse them in contexts in which other interaction models are best suited.

OMIM also seems valuable in the context of MOWS because it allows several objects to interact simultaneously. Although most analysis and design methods use constructs in which several objects need to interact simultaneously, very little support is available in current programming languages, which makes our framework valuable because it can be used to reduce the gap between the constructs such methods provide and their implementation.

## Acknowledgements

We are thankful to our referees for their insightful suggestions on an earlier version of this paper, to our editors, and also to the students who served as guinea pigs to evaluate our proposal.

## References

[1] A. Adir, Compiling programs with multiparty interactions and teams, Ph.D. thesis, Technion, 1994.

[2] G. Agha, S. Frølund, R. Panwar, D. Sturman, A linguistic framework for dynamic composition of dependability protocols, in: C.E. Landwehr, B. Randell, L. Simoncini (Eds.), Proceedings of the 3rd Working Conference on Dependeble Computing for Critical Applications, IFIP Transactions, Springer, Berlin, 1993, pp. 345–363.
[3] M. Aksit, Separation and composition of concerns in the object-oriented model, ACM Computing Surveys, 28(4es) (1996).
[4] M. Aksit, L. Bergmans, S. Vural, An object-oriented language-database integration model: the composition-filters approach, in: O.L. Madsen (Ed.), Proceedings of the European Conference on Object-oriented Programming, ECOOP'92, Lecture Notes in Computer Science, Utrecht, The Netherlands, July 1992, Springer, Berlin, 1992, pp. 372–395.
[5] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, A. Yonezawa, Abstracting object interactions using composition filters, in: R. Guerraoui, O. Nierstrasz, M. Riveill (Eds.), Proceedings of the Workshop on Object-Based Distributed Programming, ECOOP'93, Lecture Notes in Computer Science, vol. 791, Springer, Berlin, 1994, pp. 152–184.
[6] E. Andersen, Conceptual modeling of objects: a role modeling approach, Ph.D. thesis, University of Oslo, 1997.
[7] R.L. Bagrodia, Process synchronization: design and performance evaluation of distributed algorithms, IEEE Transactions on Software Engineering 15 (9) (1989) 1053–1065.
[8] B. Bauer, J. Müller, J. Odell, Agent UML: a formalism for specifying multiagent interaction, in: P. Ciancarini, M. Wooldridge (Eds.), Proceedings of the 22nd International Conference on Software Engineering ICSE'01, Lecture Notes in Computer Science, vol. 1957, Springer, Berlin, 2001, pp. 91–103.
[9] B. Bauer, J. Müller, J. Odell, Agent UML: a formalism for specifying multiagent software systems, International Journal of Software Engineering and Knowledge Engineering 11 (3) (2001) 207–230.
[10] L. Bergmans, M. Aksit, Composing crosscutting concerns using composition filters, Communications of the ACM 44 (10) (2001) 51–57.
[11] G. Booch, Object-Oriented Design with Applications, Benjamin/Cummings, Redwood City, CA, 1990.
[12] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H.F. Nielsen, S. Thatte, D. Winer, Simple object access protocol (SOAP 1.1), Technical Report. Available from <http://www.w3.org/TR/SOAP>, W3C Consortium, 2000.
[13] A. Brodsky, D. Brodsky, I. Chan, Y. Coady, J. Pomkoski, G. Kiczales, Aspect-oriented incremental customization of middleware services, Technical Report TR-2001-06, Department of Computer Science, University of British Columbia, 2001.
[14] G. Caire, F. Leal, P. Chainho, R. Evans, F. Garijo, J. Gómez, J. Pavón, P. Kearney, J. Stark, P. Massonet, Agent oriented analysis using MESSAGE/UML, in: Proceedings of Agent-Oriented Software Engineering, AOSE'01, Montréal, 2001, pp. 101–108.
[15] N. Carriero, D. Gelernter, Linda in context, Communications of the ACM 32 (4) (1989) 444–458.
[16] D. de Champeaux, Object-oriented analysis and top-down software development, in: Proceedings of the European Conference on Object-Oriented Programming, ECOOP'91, Lecture Notes in Computer Science, vol. 512, Springer, Berlin, 1991, pp. 360–375.
[17] K.M. Chandy, J. Misra, Parallel Program Design: A Foundation, Addison-Wesley, Reading, MA, 1988.
[18] P. Coad, E. Yourdon, Object-Oriented Analysis, Computing Series, Yourdon Press, Englewood Cliffs, NJ, 1990.
[19] E.G. Coffman, M.J. Elphick, A. Shoshani, System deadlocks, Computing Surveys 3 (2) (1971) 67–78.
[20] C. Constantinides, A. Bader, T. Elrad, An aspect-oriented design framework for concurrent systems, in: A.M.D. Moreira, S. Demeyer (Eds.), ECOOP Workshops, 1999, p. 302.
[21] C.A. Constantinides, A. Bader, T. Elrad, P. Netinant, M.E. Fayad, Designing an aspect-oriented framework in an object-oriented environment, ACM Computing Surveys 32 (1es) (2000) 41.
[22] R. Corchuelo, O. Martín, An IP-based language for the description of distributed, reactive programs, in: Proceedings of the V European Concurrent Engineering Conference ECEC'98, Erlagen-Nuremberg, Germany, April 1998, pp. 267–270.
[23] R. Corchuelo, J.A. Pérez, M. Toro, A multiparty coordination aspect language, ACM Sigplan 35 (12) (2000) 24–32.
[24] R. Corchuelo, D. Ruiz, M. Toro, J.M. Prieto, J.L. Arjona, A distributed solution to multiparty interaction, in: Recent Advances in Signal Processing and Communications, World Scientific Engineering Society, Singapore, 1999, pp. 318–323.
[25] R. Corchuelo, D. Ruiz, M. Toro, A. Ruiz-Cortés, Implementing multiparty interactions on a network computer, in: Proceedings of the XXVth Euromicro Conference (Workshop on Network Computing), Milan (Italy), September 1999, IEEE Press.
[26] R. Corchuelo, A. Ruiz-Cortés, Advances in Business Solutions, Catedral, 2002.
[27] R. Corchuelo, A. Ruiz-Cortés, J.R. Mühlbacher, J. García-Consuegra, Object-oriented business solutions, in: Object-Oriented Technology, Lecture Notes in Computer Science, vol. 2323, Springer, Berlin, 2002, chapter 8.
[28] R. Corchuelo, A. Ruiz-Cortés, R. Wrembel, Technologies Supporting Business Solutions, Nova Science Publishers, 2002.
[29] J.A. Díaz-Pace, M.R. Campo, Analyzing the role of aspects in software design, Communications of the ACM 44 (10) (2001) 66–73.
[30] J.A. Díaz-Pace, F. Trilnik, M.R. Campo, How to handle interacting concerns, in: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications OOPSLA'00, Minneapolis, Minnesota, USA, October 2000.

[31] E.W. Dijkstra, A Discipline of Programming, Prentice-Hall, Englewood Cliffs, NJ, 1976.
[32] D.F. D'Souza, A.C. Wills, Objects, Components, and Frameworks with UML: The Catalysis Approach, Addison-Wesley, Reading, MA, 1999.
[33] T. Elrad, M. Aksit, G. Kiczales, K. Lieberherr, H. Ossher, Discussing aspects of AOP, Communications of the ACM 44 (10) (2001) 33–38.
[34] T. Elrad, R.E. Filman, A. Bader, Aspect-oriented programming, Communications of the ACM 44 (10) (2001) 29–32.
[35] R. Englander, Java and SOAP. O'Reilly and Associates, 2002.
[36] M. Fayad, E-frame: a process-based, object-oriented framework for e-commerce, in: Proceedings of the International Conference on Internet Computing IC'2001, vol. 1, Las Vegas, Nevada, USA, June 2001, CSREA Press, 2001, pp. 124–128.
[37] M. Fayad, R.E. Johnson (Eds.), Domain-Specific Application Frameworks: Frameworks Experience by Industry, Wiley, New York, 1999.
[38] M. Fayad, D.C. Schmidt (Eds.), Building Application Frameworks: Object-Oriented Foundations of Framework Design, Wiley, New York, 1999.
[39] M. Fayad, D.C. Schmidt, R.E. Johnson (Eds.), Implementing Application Frameworks: Object-Oriented Frameworks at Work, Wiley, New York, 1999.
[40] M.E. Fayad, D.S. Hamu, D. Brugali, Enterprise frameworks characteristics, criteria, and challenges, Communications of the ACM 43 (10) (2000) 39–46.
[41] M.E. Fayad, D.C. Schmidt, Lessons learned building reusable OO frameworks for distributed software, Communications of the ACM 40 (10) (1997) 85–87.
[42] P. Ferguson, W.S. Humphrey, S. Khajenoori, S. Macke, A. Matvya, Results of applying the personal software process, Computer 30 (5) (1997) 24–31.
[43] I.R. Forman, Design by decomposition of multiparty interactions in Raddle. in: 5th International Workshop on Software Specification and Design, May 1989, pp. 2–10.
[44] N. Francez, I. Forman, Interacting Processes: A Multiparty Approach to Coordinated Distributed Programming, Addison-Wesley, Reading, MA, 1996.
[45] E. Freeman, S. Hupfer, K. Arnold, JavaSpaces Principles, Patterns, and Practice, Addison-Wesley Longman, Reading, MA, 1999.
[46] S. Frølund, G. Agha, A language framework for multi-object coordination, in: O. Nierstrasz (Ed.), Proceedings of the European Conference on Object-oriented Programming ECOOP'93, Kaiserslautern, Germany, 1993, Springer, Berlin, 1993, pp. 346–360.
[47] C. Ghezzi, M. Jazayeri, D. Mandrioli, Introduction to the Personal Software Process, Addison-Wesley, Reading, MA, 1997.
[48] J. Gray, T. Bapty, S. Neema, J. Tuck, Handling crosscutting constraints in domain-specific modeling, Communications of the ACM 44 (10) (2001) 87–93.
[49] P. Grefen, K. Aberer, Y. Hoffner, H. Ludwig, Crossflow: cross-organizational workflow management in dynamic virtual enterprises, International Journal of Computer Systems Science and Engineering 15 (5) (2000) 277–290.
[50] J. Irwin, J.-M. Loingtier, J.R. Gilbert, G. Kiczales, Aspect-oriented programming of sparse matrix code, in: Proceedings of the 1997 International Scientific Computing in Object-Oriented Parallel Environments, Lecture Notes in Computer Science, vol. 1343, Springer, Berlin, December 1997, pp. 249–256.
[51] Y.J. Joung, A comprehensive study of the complexity of multiparty interaction, in: Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages POPL'92, ACM Press, January 1992, pp. 142–153.
[52] Y.J. Joung, Two decentralized algorithms for strong interaction fairness for systems with unbounded speed variability, Theoretical Computer Science 243 (1–2) (2000) 307–338.
[53] Y.J. Joung, S.A. Smolka, A comprehensive study of the complexity of multiparty interaction, Journal of the ACM 43 (1) (1996) 75–115.
[54] Y.J. Joung, S.A. Smolka, Strong interaction fairness via randomization, IEEE Transactions on Parallel and Distributed Systems 9 (2) (1998) 137–149.
[55] J. Kamatar, W. Hayes, An experience report on the personal software process, IEEE Software 17 (6) (2000) 85–89.
[56] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersen, J. Palm, W.G. Griswold, An overview of AspectJ, in: Proceedings European Conference on Object-Oriented Programming ECOOP'01, Lecture Notes in Computer Science, vol. 2072, Springer, Berlin, 2001, pp. 327–353.
[57] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin, Aspect-oriented programming, in: Proceedings of the European Conference on Object-Oriented Programming ECOOP'97, Lecture Notes in Computer Science, Springer, Berlin, 1997, pp. 220–242.
[58] K.J. Lieberherr, Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns, PWS Publishing Company, Boston, USA, 1996.
[59] K. Liebherherr, From transience to persistence in object-oriented programming: patterns and architectures, ACM Computing Surveys 28A (4) (1996) 39–41.
[60] K. Liebherherr, D. Orleans, J. Ovlinger, Aspect-oriented programming with adaptive methods, Communications of the ACM 44 (10) (2001) 39–41.
[61] C.V. Lopes, D: a language framework for distributed programming, Ph.D. thesis, Xerox Palo Alto Research Center, 1998.
[62] G.C. Murphy, R.J. Walker, E.L.A. Baniassad, Evaluating emerging software development technologies: lessons learned from assessing aspect-oriented programming, IEEE Transactions on Software Engineering 25 (4) (1999) 438–455.
[63] G.C. Murphy, R.J. Walker, E.L.A. Baniassad, M.P. Robillard, A. Lai, M.A. Kersten, Does aspect-oriented

programming work?, Communications of the ACM 44 (10) (2001) 75–77.
[64] P. Netinant, C. Constantinides, T. Elrad, M.E. Fayad, Supporting aspectual decomposition in the design of operating systems, in: J. Malenfant, S. Moisan, A.M.D. Moreira (Eds.), Proceedings of the 3rd International Workshop on Object-Orientation and Operating Systems, ECOOP 2000, Sophia, France, 2000.
[65] P. Netinant, T. Elrad, M.E. Fayad, A layered approach to building open aspect-oriented systems: a framework for the design of on-demand system demodularization, Communications of the ACM 44 (10) (2001) 83–85.
[66] J. Odell, H.V.D. Parunak, B. Bauer, Representing agent interaction protocols in UML, in: P. Ciancarini, M. Wooldridge (Eds.), Proceedings of the 22nd International Conference on Software Engineering ISCE'01, Lecture Notes in Computer Science, vol. 1957, Springer, Berlin, June 2001, pp. 121–140.
[67] H. Ossher, P. Tarr, Using multidimensional separation of concerns to (re)shape evolving software, Communications of the ACM 44 (10) (2001) 43–50.
[68] G. Papadopoulos, F. Arbab, Coordination models and languages, in: Advances in Computers, vol. 46, Academic Press, New York, 1998.
[69] D.L. Parnas, On the criteria to be used in decomposing system into modules, Communications of the ACM 15 (12) (1972) 1053–1058.
[70] R. Pawlak, L. Duchien, G. Florin, An automatic aspect weaver with a reflective programming language, in: P. Cointe (Ed.), Proceedings of the 2nd International Conference on Metalevel Architectures and Reflection, Lecture Notes in Computer Science, St. Malo, France, July 1999, vol. 1616, Springer, Berlin, 1999, pp. 147–149.
[71] J.A. Pérez, R. Corchuelo, D. Ruiz, M. Toro, A framework for aspect-oriented multiparty coordination, in: New Developments in Distributed Applications and Interoperable Systems, Kluwer Academic Publishers, Dordrecht, 2001, pp. 161–174.
[72] J.A. Pérez, R. Corchuelo, D. Ruiz, M. Toro, An enablement detection algorithm for open multiparty interactions, in: Proceedings of the 2002 ACM Symposium on Applied Computing SAC'02, Madrid, Spain, March 2002, ACM Press, 2002, pp. 378–384.
[73] J.A. Pérez, R. Corchuelo, D. Ruiz, M. Toro, An order-based, distributed algorithm for implementing multiparty interactions, in: Coordination Models and Languages. Proceedings of the 5th International Conference COORDINATION 2002, Lecture Notes in Computer Science, York, United Kingdom, 2002, Springer, Berlin, 2002, pp. 250–257.
[74] M. Pinto, L. Fuentes, M.E. Fayad, J.M. Troya, Separation of coordination in a dynamic aspect-oriented framework, in: Proceedings of the First International Conference on Aspect-Oriented Software Development AOSD'01, Enschede, The Netherlands, April 2002.
[75] D.S. Platt, Introducing Microsoft .NET, second ed., Microsoft Press, 2002.
[76] T. Reenskaug, P. Wold, O.A. Lehne, Working With Objects. The OOram Software Engineering Method, Manning Publications Co., August 1995.
[77] A. Ruiz-Cortés, R. Corchuelo, J.A. Pérez, A. Durán, M. Toro, An aspect-oriented approach based on multiparty interactions to specifying the behaviour of a system, in: Principles, Logics, and Implementations of High-Level Programming Languages PLI'99. Workshop on Object-Oriented Specification Techniques for Distributed Systems and Behaviours, Paris, France, 1999, pp. 56–65.
[78] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, Object-Oriented Modeling and Design, Prentice-Hall, Schenectady, NY, 1991.
[79] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modeling Language Reference Manual. Object Technology Series, Addison-Wesley Longman, Reading, MA, 1999.
[80] A.R. Silva, P. Sousa, J.A. Marques, Development of distributed applications with separation of concerns, in: Proceedings of the 1995 Asia-Pacific Software Engineering Conference APSEC'95. IEEE Press, 1995.
[81] G.T. Sullivan, Aspect-oriented programming using reflection and metaobject protocols, Communications of the ACM 44 (10) (2001) 95–97.
[82] B. Vanhaute, E. Truyen, W. Joosen, P. Verbaeten, Composing non-orthogonal meta-programs, in: Proceedings of the 1st Workshop on MultiDimensional Separation of Concerns at OOPSLA'99, 1999.
[83] J. Viega, D. Evans, Separation of concerns for security, in: P. Tarr, A. Finkelstein, W. Harrison, H. Nusibeh, H. Osser, D. Perry (Eds.), Proceedings of the Workshop on Multi-Dimensional Separation of Concerns in Software Engineering at ICSE'00, 2000.
[84] R. Wirfs-Brock, B. Wilkerson, Designing Object-Oriented Software, Prentice-Hall, Englewood Cliffs, NJ, 1990.
[85] A.F. Zorzo, R.J. Stroud, A distributed object-oriented framework for dependable multiparty interactions, ACM Sigplan 34 (10) (1999) 435–446.