Multi-Party Coordination in the Context of MOWS

R. Corchuelo, J. A. Pérez, and A. Ruiz

Universidad de Sevilla, Dpto. de Lenguajes y Sistemas Informdticos, Avda. de la Reina Mercedes, s/n. Sevilla 41.012, Spain

e-mail: corcku@lsi.us.es

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. In this paper, we show that interactions amongst a number of objects can also be described separately from functionality, which enhances reusability of functional code and interaction patterns. We present our proposal in the context of *Multi-Qrganisational Web-Based Systems (MOWS)* and also present a framework that provides the infrastructure needed to implement multiparty coordination as an independent aspect.

1. INTRODUCTION

Isolating interaction from computation has been paid much attention because it benefits from enhancing modularity, understandability or reusability, but also from being the best way to help relieve well-known problems such as the inheritance anomaly. Unfortunately, aspect-oriented languages, such as Cool, Ridl [l], AspectJ [2] or Aml [3] do not succeed in isolating computation from interaction with other objects in a system. Cool, for instance, allows us to define synchronization policies, but interactions with other objects are embedded into the functional code.

Therefore, objects are dependent on the interaction model used to coordinate them. This model usually relies on classical point-to-point communication primitives and, thus, emphasizes a number of entities exchanging binary message and requires a specific protocol for coordinating them that is usually scattered amongst functionality. Besides point-to-point communication, many other interaction models have been proposed in the literature [4]. Each one has its own strengths and weaknesses, so that there is no universally accepted interaction model, but a wide spectrum of alternatives. Thus, it is desirable for aspect-oriented languages dealing with interaction to exist. Such languages enhance reusability of functional code and coordination patterns that can be applied to similar situations as long as adequate mapping languages existed. Such languages also allow programmers to decide which interaction model better suits his or her needs, which has a direct effect on understandability, maintainability, and evolvability.

In this paper, we present a proposal that aims at describing aspect-oriented interaction [5] in the context of *Multi-Organisational Web-Based System*, which are systems composed of coarse-grained business objects that reside on different organisations and need to cooperate frequently [6, 7]. The paper is organized as follows. Section 2 presents a comprehensive description of our motivation to work on this proposal; Section 3 deals with high-level interaction models suitable to be applied in the context of *MOWS* and presents the linguistic support we have designed; Section 4 presents our framework for implementing aspect-oriented multiparty coordination. Finally, Section 5 shows our main conclusions.

2. MOTIVATION

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 that typically describe multi-organisational systems as collections of components that are exposed to programmers as services. Such services are usually implemented as a set of interrelated business objects that encapsulate a protocol for a logical unit of work, e.g., transferring money, processing an order, updating a customer record, or moving a task to the next person in a workflow queue.

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. In theory, programmers should only care about the functionality they provide. Unfortunately, this vision is far too idealistic because business objects are seldom isolated in a system, and they have to work coordinately with other objects in order to achieve a common goal. This implies that business objects are more than simple abstract descriptions, and they are tightly coupled with the underlying interaction model and the technology used to communicate them.

Furthermore, this technology is advanced enough and should be applied as automatically as possible, so that business objects remain as much abstract, handy, and reusable as possible. Thus, 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.

We think that automatization of the software development process is the key to succeeding in the Internet world, and separating interaction models from computations brings designers many opportunities to tests new approaches or ideas without modifying the functional code of their business objects.

3. A HIGH-LEVEL INTERACTION MODEL FOR MOWS

Client/server primitives such as remote procedure call or message passing are the *de facto* industrial standard for object interaction. Although this model has been proven to be appropriate in the local set, *MOWS* require careful thinking about problems, such as partial failure, increased latency, or language interoperability [7, 8]. It is not surprising then that many researchers have put a great deal of effort on researching advanced interaction models that are more appropriate to describe distributed systems [4].

JavaSpaces [9] have been recently proposed as a *Linda*-based [10] model for coordinating Java objects on the web. This technology provides a mechanism for storing a group of related objects and retrieving them based on a value-matching lookup for specified fields, and it allows the programmer to easily build distributed 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, often, workflow is not enough, specially in the field of *MOWS* and *e-commerce* applications. The reason is that many problems in this context require a number of objects to cooperate in order to achieve a global goal: transferring money from a bank to another by means of a point of sales terminal, reaching a virtual agreement in an auction sale, paying taxes, and so on.

For instance, assume we have to design a debit-card system [11], 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, and 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 a tuple space; 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.

In general, solutions that are based on the classical binary client/server model are difficult to apply in problems in which several objects need to cooperate simultaneously in order to achieve a common goal. Current technology provides transactional servers for grouping individual client/server interactions, but there is no clear separation between the problem to be solved and the underlying technology to be used.

The need for a higher-level interaction model has been recognised as one of the main goals in the design of the new *Microsoft's.NET* platform. Built on the standard integration fabric of *XML* and *Internet* protocols, the *Microsoft.NET* platform brings programmers a new way to develop advanced applications that can integrate or orchestrate any group of services on the Internet into a single solution. Unfortunately, the orchestrating component of the *.NET* platform does not succeed in separating high-level coordination from the underlying technology. Thus, this solutions are tightly coupled with the way they are implemented.

These concerns argue for an interaction model that allow to describe coordination patterns amongst an arbitrary number of object as much independently as possible from the underlying technology. Such an interaction model exists, and it is usually referred to as the *Multiparty Interaction Model* [12, 13]. It is well suited to capture the essence of problems in which several objects residing on different organisations need to cooperate coordinately and simultaneously, and 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 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 synchronization. The programmer does not need to care about these low level details, which can be generated in a completely automatic way, thus easing maintainability.

This is the reason why it is not surprising that it has also attracted the attention of the designers of the well-known *Catalysis* method [14], which is a next generation *UML*-based approach for the systematic business-driven development of component-based systems. *Catalysis* has been used by *Fortune 500* companies in fields including finance, telecommunication, insurance, manufacturing, embedded systems, process control, flight simulation, travel and transportation, or systems management, thus proving the adequacy of this novel interaction model in so different application domains.

We have focused on this interaction model because, although dozens of papers dealing with it exist, none has addressed the problems of open multiparty interactions, passive objects, or aspect-orientation.

3.1. Linguistic Support

In order to support the above-described high-level interaction model, we have designed CAL [5], which is a language that aims at describing coordination patterns amongst a number of objects in a way that is independent from computation or other aspects. Coordination patterns are not dependent on the objects they coordinate, so that they can be easily reused.

In this section, we glance at CAL and describe its main features by means of the debit-card system. 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. Figure 1 shows a description of the debit-card system in CAL that is analyzed in the following subsections.

3.1.1. Describing interactions. Interactions are defined by means of the following syntax:

`interaction` <*name*>
  [<*participating object descriptions*>]
  (<*slot descriptions*>)
  `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 coordinates 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 *IP* [13], 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 *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 play that role; (ii) those objects agree in interacting with each other, i.e., the constraints they stablish are satisfied. A set of objects which can execute an interaction is what we call an *enablement*.

Since exclusion must be guaranteed, an object cannot commit to more than one interaction at a time. But, 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*.

3.1.2. 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` <*inter face*>
`{`
  <*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*, that encapsulates the details concerning waiting for a new sale and interactions with the clerk initiating it, *Get_Price* that can be invoked after a new sale has been initiated and reports its price, *Get_Customer_Account* and *Get_Merchant_Account*, that provide references to the involved accounts, and *Report_Results*, that can be invoked to report whether a transfer has been done or not. Similarly, pattern *Account* requires that the three operations in the interface *I Account* be implemented: *Charge*, to withdraw money from an account, *Pay_In*, to pay money into it,

```
        interaction transfer;


  [term as Terminal; source, dest as Account]
  (int 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 = Authorize 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.

and *Authorise_Payment*, that 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 in the brackets identify the objects with which the object executing such a statement is interested in cooperating, and

```
interface ITerminal {
   void Wait_For_Sale();
   int Get_Price();
   OID Get_Customer_Account();
   OID Get_Merchant_Account();
   void Report .Result (boolean done);
}

interface IAccount {
   void Charge(int sum);
   void Pay_In(int sum);
   boolean Authorize_Payment(int sum);
}
```

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

comm_stat is a communication statement involving the slots of interaction *I*. The expressions between brackets are of the form of *role*!*id*, that means that the *role* must be played by the object identified by *id*. The expression *role*!*self* is used to denote the role that the object that issues the offer is playing in the interaction. If no expression is given for a role of the interaction, it is assumed that the object accepts that any object can play such a role in the interaction.

Therefore, the behaviour of a terminal can be summarized 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 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.

3.1.3. 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 want 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 *bank Account*, but it does not provide the operations this pattern requires. For instance, there is no operation for deciding whether charging a sum is affordable or not. Fortunately, the expression *getBalance*() >= *sum* implements it

This mapping allows us to write fully abstract, reusable behaviour specifications. This is important, because in other aspect languages such as Cool, Ridl, AspectJ, or Aml, aspect specifications reference the classes onto which they are applied by name, which causes dependencies between components and aspects, and makes difficult their effective reuse.

4. A FRAMEWORK FOR ASPECT-ORIENTED OPEN MULTI-PARTY COORDINATION

We have carefully designed a framework that offers a number of high-level services for implementing Cal. We have arrive at its design largely through experimentation. In [15, 16], we presented two solutions to the problem, but, unfortunately, they lacked a systematic way to change the middleware layer or the algorithm used to select amongst conflicting interactions. Given that new results in this area appear continuously, we

```
class bankAccount {
   private int balance;
   public void updateBalance(int sum) { balance +a sum; }
   public boolean getBalanceQ { return balance; }
}
map behaviour Account onto class bankAccount where
   Charge(sum) = updateBalance(-sum);
   Pay_In(sum) = updateBalance(+sum);
Authorize Payment(sum) = (getBalance() >= sum);
```

Fig. 3. Mapping for behaviour *Account* onto class *bank Account*.

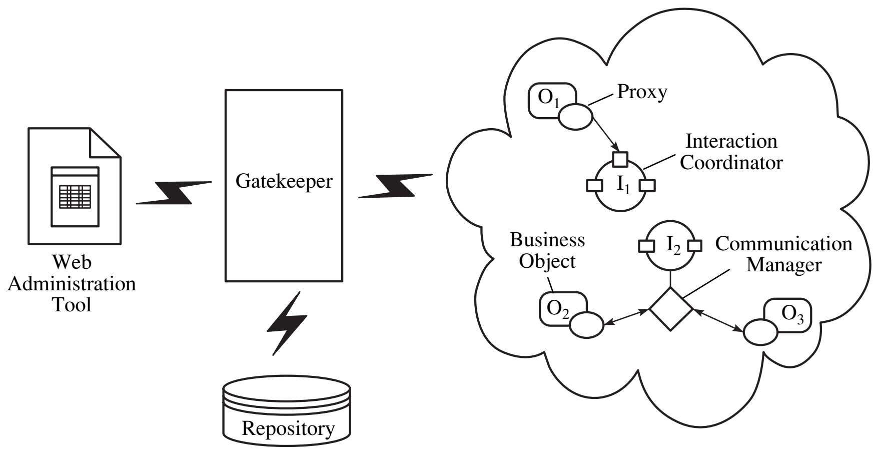

Fig. 4. The architecture of our solution.

decided to design a framework able to deal with them easily.

Our two main design goals were, thus, 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. However, objects are passive in may real-world problems, and a good framework should be able to deal with them without compromising effectiveness.

Figure 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 involve a common object.

Proxies. In our framework, 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.

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 [17].

At 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 run-time 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* [18]. This helps keep the different contexts in which a business object may participate separated, and system configuration is simplified.

Figure 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. 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 interface *IPart Proxy*, which groups the operations an object needs,

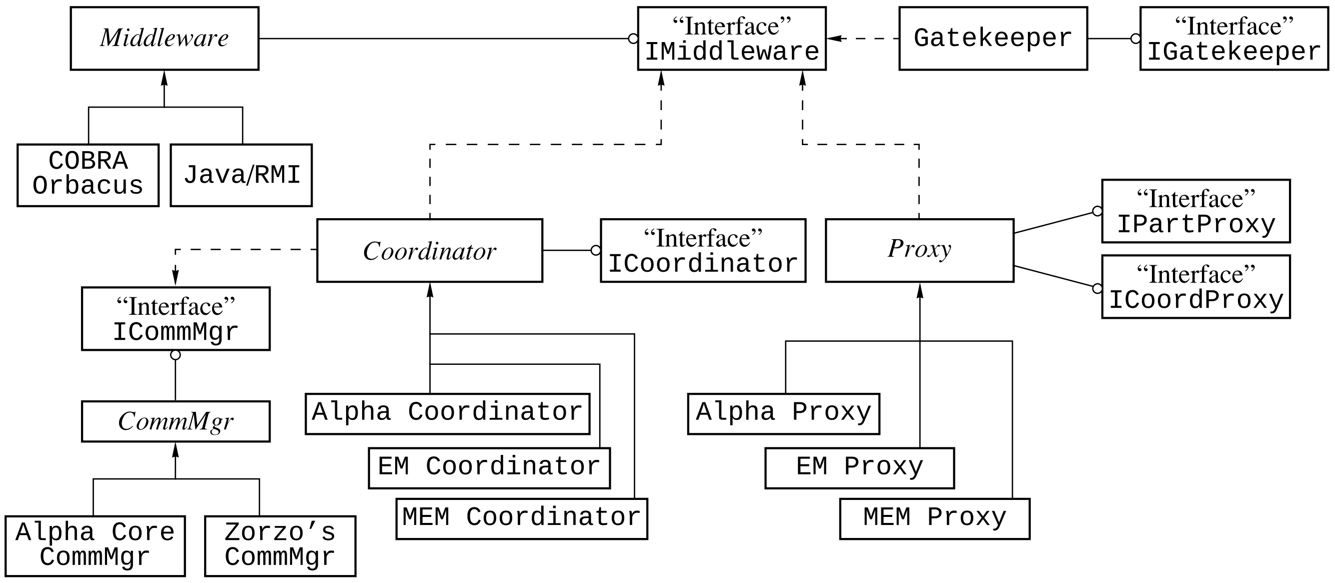

Fig. 5. Class framework.

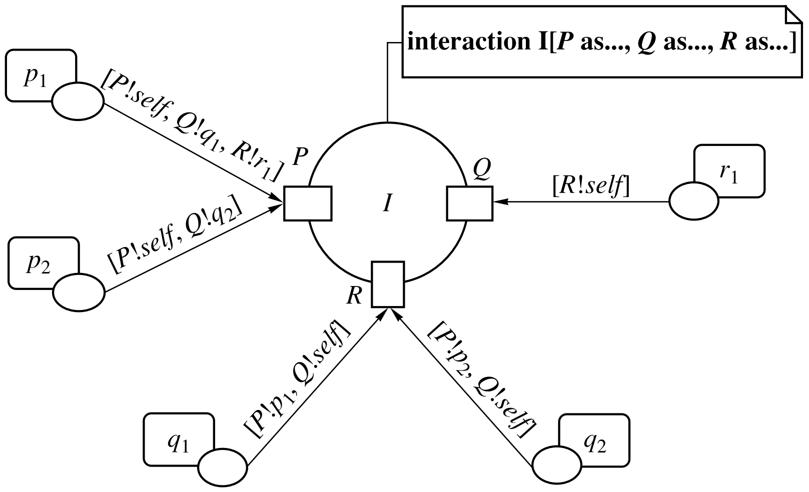

Fig. 6. A simple system.

and *ICoordProxy*, which groups the operations coordinators need.

4.1. Coordination Algorithms

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

An important feature of a good business framework is that it allows for evolution [18]. Thus, it is needed a better solution allowing for open interactions in which the set of objects participating in them is not known until run time. We have designed an algorithm for dealing with multiparty coordination in this context, which is referred to as α. 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, α was split into two parts referred to as α-Solver and α-Core [19].

We sketch the main ideas behind α-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* that may behave in some way. Let us also assume that

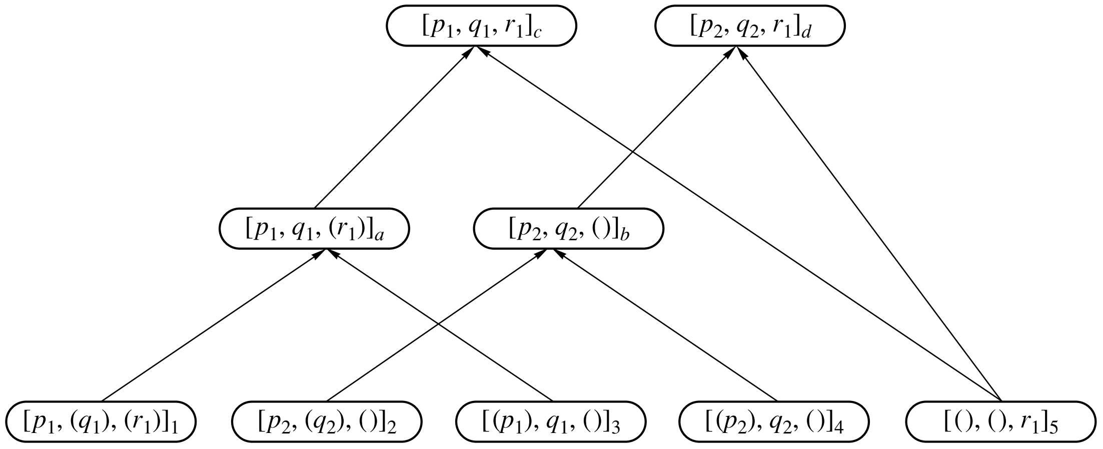

Fig. 7. Consolidation graph for the system in Fig. 6.

objects p_1 and p_2 are interested in playing role *P*, objects q_1 and q_2 make an offer to play role *Q*, and object r_1 wants 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 in the interaction, as described in section 3.1.2. 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*.

α-Solver processes offers as they arrive and forms incrementally the consolidation graph in Fig. 7 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 α-Solver constructs a consolidation graph with only one node $[p_1, (q_1), (r_1)]$ that is interpreted in 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 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 this 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 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 [20]: 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 program so that they request resources in increasing order, it has been proven to be quite effective in this context.

4.2. 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.

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 run time 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 re-processes it each time the passive object it represents changes its state. If one of the active objects willing to participate in that interaction give up, the coordinator also notifies the involved passive objects so that their proxies can stop monitoring the moment when they are ready to interact.

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 immediately replied.

Sometimes, many passive objects reside in 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 significantly degrade performance. 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 and scalability.

5. CONCLUSIONS

In this paper, we have explored the aspect-oriented paradigm, the multiparty interaction model, and how programming distributed systems in the contexts of *MOWS* may benefit from both. Separating computation from coordination encourages reuse, improves comprehension, and eases maintenance and evolution of software.

We have also presented a framework and a set of coordination algorithms that can be used to implement multiparty coordination as a separate aspect and constitutes infrastructure for implementing CAL, which is, as far as we know, the first aspect-oriented language dealing with such concern in the literature. It also deals with multi-organisational system and passive objects.

We have implemented the framework using *Java* 1.2 and *Orbacus* 3.3.2 and *Java/RMI* as middlewares, and we have also conducted some experiments and performance analysis on this prototype. Our results show that using the framework introduces a penalty that is about 20% when compared with a manually developed solution. Obviously, much work on optimization remains to be done. However, our proposal is promising because, although it introduces some performance penalty, our experiments show that the development time is approximately three times shorter than by using *.NET* orchestration or JavaSpaces. The reason is that the programmer does only need to concentrate on developing pure functionality, independently from the technology used to coordinate its objects which is dealt with by the framework and the weaver.

BIOGRAPHIES

Rafael Corchuelo, Associate Professor at the Department of Computer Languages and Systems of the University of Sevilla, Spain. Received *PhD* degree from this University. Leads Research Group on Distributed Systems since 1997. Scientific interests: instrumentation techniques to support the multiparty interaction model as well as fairness concerns. Reviewer of the *ACM Computing Reviews, Concurrency*: *Practice and Experience, Computacion y Sistemas* and *Communications of the ACM*.

Jose A. Pérez. Received Master Degree in Computer Science in 1993 and *PhD* degree in 2000. Worked as a software engineer and a consultant for companies, such as *Quimisur*, in projects concerning embedded real-time systems. Scientific interest: multiparty coordination and aspect orientation.

Antonio Ruiz. Received *B.S.E* and *M.S* degrees in Computer Engineering from the University of Sevilla. Lecturer at the School of Computer Science of the University of Sevilla since 1998. Scientific interests: non-functional requirements, software architecture, and component-oriented technologies in web applications.

REFERENCES

1. Lopes, C.V., D: A Language Framework for Distributed Programming. *PhD Thesis*, Xerox Palo Alto Research Center, 1998.
2. Lopes, C.V. and Kiczales, G., Recent Developments in AspectJ, Object-Oriented Technology: ECOOP'98 Workshop Reader Demeyer S. and Bosch J., Eds.; *Lecture Notes in Computer Science*, 1998 vol. 1543, pp. 398-401, Springer.
3. Irwin, J., Loingtier, J.-M., Gilbert, J.R., and Kiczales, G., Aspect-oriented Programming of Sparse Matrix Code, *Proc. of the 1997 Int. Sci. Computing in Object-Oriented Parallel Environ*ments; *Lecture Notes in Computer Science*, 1997, vol. 1343, pp. 249–256, Springer.
4. Papadopoulos, G. and Arbab, F., Coordination Models and Languages, *Advances in Computers*, 1998, vol. 46.
5. Corchuelo, R., Pérez, J.A., and Tow, M., A Multiparty Coordination Aspect Language, *ACM Sigplan,* 2000, vol. 35, no. 12, pp. 24–32.
6. Corchuelo, R. and Ruiz, A., Advances in Business Solutions, Nova Science, 2001.
7. Corchuelo, R., Ruiz, A., Mïhlbacher, J.R., and Garcia-Consuegra, J. Object-oriented Business Solutions, *ECOOP'2001 Workshop Reader,* LCNS, Springer, 2001.
8. Ruiz, A., Corchuelo, R., Martín, O., Durán, A., and Toro, M., Addressing Interoperability in Multi-organisational Web-based Systems, *European Conf. on Object-Oriented Programming ECOOP'OO, Workshop on Object Interoperability WOI'QO*, Sophia Antipolis, 2000, pp. 87–96.
9. Freeman, E., Hupfer, S., and Arnold, K., *JavaSpaces Principles, Patterns, and Practice*, Addison-Wesley, 1999.
10. Carriero, N., and Gelernter, D., Linda in Context, *Commun. ACM*, 1989, vol. 32, no. 4, pp. 444–458.
11. Ruiz, A., Corchuelo, R., Pérez, J.A., Durán, A., and Toro, M., An Aspect-oriented Approach Based on Multiparty Interactions to Specifying the Behaviour of a System, *Principles, Logics, and Implementations of High-Level Programming Languages PLI'99, Workshop on Object-*

Oriented Specification Techniques for Distributed Systems and Behaviours, Paris, 1999.

12. Corchuelo, R., Ruiz, D., Toro, M., and Durán, A., Avances en la coordinacidn de objetos activos, *Novatica*, 2000, vol. 143, pp. 34–37.
13. Francez, N. and Forman, I., *Interacting Processes: A Multiparty Approach to Coordinated Distributed Programming*, Addison-Wesley, 1996.
14. D'Souza, D.F. and Wills, A.C., *Objects, Components, and Frameworks with UML: The Catalysis Approach*, Reading, Mass.: Addison-Wesley, 1999.
15. Corchuelo, R., Ruiz, D., Toro, M., Prieto, J M., and Arjona J.L., A Distributed Solution to Multiparty Interaction, in *Recent Advances in Signal Processing and Communications*, World Sci., 1999, pp. 318–323.
16. Corchuelo, R., Ruiz, D., Toro, M., and Ruiz, A., Implementing Multiparty Interactions on a Network Computer, *Proc. of the XXVth Euromicro Conf. (Workshop on Network Computing)*, Milan: IEEE Press, 1999.
17. Zorzo, A.F. and Stroud, R.J., A Distributed Object-oriented Framework for Dependable Multiparty Interactions, *ACM Sigplan*, 1999, vol. 34, no. 10, pp. 435–446.
18. Building Application Frameworks: Object-Oriented Foundations of Framework Design Fayad, M. and Schmidt, D.C., Eds., Wiley, 1999.
19. Pérez, J.A., Corehuelo, R.J., Ruiz, D., and Toro, M., A Framework for Aspect-oriented Multiparty Coordination, *Working Conf. on Distributed Appieations and Interoperable Systems DAIS'01*, Krakow, 2001, Kluwer, (in press).
20. Coffman, E.G., Elphick, M.J., and Shoshani, A., System Deadlocks, *Computing Surveys*, 1971, vol. 3, no. 2, pp. 67–78.