# A Multiparty Coordination Aspect Language

Rafael Corchuelo    José A. Pérez    Miguel Toro

Universidad de Sevilla, Dpto. de Lenguajes y Sistemas Informáticos
Avda. de la Reina Mercedes, s/n. Sevilla 41.012 (Spain)
E-mail: corchu@lsi.us.es, web page: www.lsi.us.es/~corchu

**Abstract**

Coordination languages were motivated by an ever-increasing need for producing highly-reusable components, which can be partially achieved by implementing them in a way that is independent of the way they interact. Isolating interaction from computation, persistence and other concerns enhances modularity, thus promoting reusability and understandability. In this paper, we concentrate on a language aimed at describing the simultaneous coordination of a number of entities, which is a problem we are usually faced with when we have to programme bank transfers, purchases with debit cards, auctions, and so on. This language relies on the novel multiparty interaction model.

## 1 Introduction

Object-oriented languages such as Java or C++ are not adequate enough to model distributed systems because aspects such as distribution, synchronisation or replication do not usually fit into the scope of a class. In fact, much of the complexity and brittleness of existing systems stems from the way the implementation of these aspects ends up in spaghetti code that causes many problems when it needs to be maintained manually. In order to deal with these cross-cutting aspects, a number of researchers began working on several approaches that allow programmers to describe each aspect in an *ad hoc* language [KLM+97]. Aspects can then be compiled and merged to obtain an executable piece of code by means of a tool called weaver.

Recently, isolating coordination from computation has been paid much attention because it benefits from enhancing modularity, understandability or reusability, but also from being the best way to solve problems such as the inheritance anomaly. Many coordination languages have been proposed [PA98], but they usually rely on the client/server model, thus emphasising a number of entities exchanging binary messages and a specific protocol for dealing with the details concerning the achievement of such a global goal. Unfortunately, sometimes, the effort needed to design, test, implement, and debug this protocol is much bigger than the effort needed to implement the rest of the application.

This motivated several researchers to introduce multiparty interaction constructs into languages for the description of distributed systems [JS96]. Unfortunately, the languages that incorporate this powerful construct are usually intended to describe both functionality and coordination, thus producing components that are highly dependent on the environment in which they are intended to be integrated. We think that aspect-orientation is the key to describe the coordinated behaviour separately from computation, persistence or other implementation aspects so that functional code can be kept clean.

Our proposal consists of a language called CAL that is aimed at increasing the level of abstraction of a program by considering the concurrent behaviour of components as an aspect where multiparty interactions are the sole means for synchronisation and communication. Figure 1 sketches our aspect-oriented

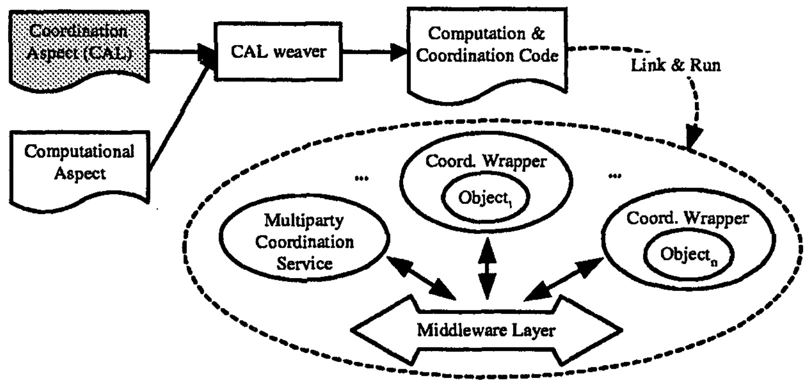


Figure 1: A framework for aspect–oriented multiparty coordination.

proposal to specifying the behaviour of a distributed system. The coordination aspect is specified in CAL, computation is described in a programming language, e.g., Java, and a weaver combines both sources into a piece of code that can then be compiled and executed on top of a middleware layer, e.g., CORBA. The code generated by the weaver is not spaghetti code, but layered code composed of a wrapper that encapsulates the details concerning coordination of the operations defined in the computational aspect.

The rest of the paper is organised as follows: section 2 introduces the multiparty interaction model, section 3 overviews CAL, section 4 analyses other authors' work, and, finally, section 5 shows our conclusions and the work we are planning on doing soon.

## 2 Multiparty Interactions

Several authors have focused on coordination primitives that model synchronous interactions amongst an arbitrary number of asynchronous entities. This form of coordination is usually referred to as the multiparty interaction model, and it is very adequate for describing problems such as bank transfers, purchases with debit cards, tax payments and so on because, in these problems, we need to coordinate simultaneously a number of entities that interact in order to achieve a common goal: transferring money from an account to another by means of a cash point or a point-of-sales terminal, reaching a virtual agreement, and so on.

A multiparty interaction consists of a set of actions aimed at exchanging information simultaneously amongst a number of entities. Obviously, each of the entities participating in a multiparty interaction must be ready to execute its corresponding actions so that it can occur. An attempt to participate in an interaction delays an entity until every participant is ready to execute it, and after the interaction is carried out, the participating entities continue their parallel, asynchronous executions.

In the following sections, we introduce some basic notation and show how multiparty synchronisation and communication work by means of classical, well–known examples.

### 2.1 Basic Notation

The notation we use to describe our examples is very similar to the notation we use in CAL. It comes from IP [FF96], which is a language aimed at describing distributed systems that is also amenable to formal reasoning. The most important statement is the interaction statement, which is of the form $a\{comm\_stat\}$. $a$ is referred to as the name of the interaction this statement tries to engage, and *comm_stat* is an optional communication statement intended to transfer information from one entity to another when multiparty synchronisation takes place. In IP, an entity that participates in an interaction has access to the state of other entities participating in the same interaction. This mechanism is not well-engineered, but it is enough to describe how multiparty communication works without taking into account too much syntactic burden.

We also use guarded non-deterministic choice statements that are of the form $[[]_{i=1}^{n} G_i \rightarrow S_i]$. Guards are of the form *B&int_stat*, where $B$ is a boolean expression and the rest is an interaction statement. A guard is said to be passable, i.e., the statements beyond the arrow can be executed, as long as $B$ holds and $a$ is enabled, i.e., all of the entities participating in it are ready to engage it. If $B = true$, it can be left out together with its corresponding ampersand. Guarded non-deterministic choices can be repeated until none of the boolean expressions in the guards holds, and the notation we use is $*[[]_{i=1}^{n} G_i \rightarrow S_i]$.

Non-deterministic choice statements à la Disjktra of the form $[[]_{i=1}^{n} B_i \rightarrow S_i]$, and loops of the form $*[[]_{i=1}^{n} B_i \rightarrow S_i]$ are also provided. They can be used to make local decisions or computations not involving coordination.

## 2.2 Multiparty Synchronisation

We illustrate multiparty synchronisation by means of the dining philosophers problem, which is a well-known multi-process synchronisation problem that consists of five philosophers sitting at a table who do nothing but think and eat. There is a single fork between each philosopher, and they need to pick both forks up in order to eat. This problem is the core of a large class of problems where an entity (the philosopher) needs to acquire a set of resources (the forks) in mutual exclusion, e.g., a point-of-sales terminal trying to have access to two bank accounts to transfer funds.

The obvious solution to this problem, using two-party interactions, consists of picking up forks in sequence. Nevertheless, a problem arises if each philosopher grabs the fork on his/her right, and then waits for the fork on his/her left to be released. In this case, a deadlock has occurred, and all philosophers will starve. If we used multiparty interactions, each philosopher would pick up his/her two forks at the same time so that no deadlock may arise.

```
behaviour Philosopher_i ::
  *[ get_fork_i{} —→
      eat ; rel_fork_i{}; think
  ]

behaviour Fork_i ::
  *[
    get_fork_i{} —→ rel_fork_i{}
  []
    get_fork_i+1{} —→ rel_fork_i+1{}
  ]
```

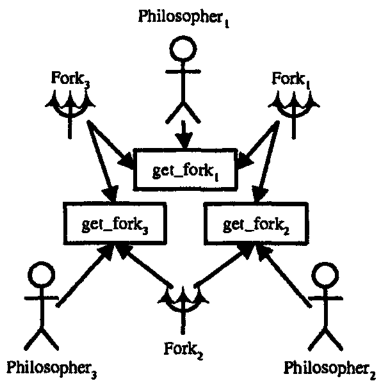


Figure 2: A solution to the dining philosophers problem.

Figure 2 shows a solution to this problem using three-party interactions where one philosopher and two forks collaborate together. The behaviour of the philosophers is described by scripts *Philosopher*$_i$, and the behaviour of the forks by scripts *Fork*$_i$ ($i = 1, 2, \ldots, n$). *Philosopher*$_i$ eternally tries to get his/her associated forks by participating in the three-party interaction *get_forks*$_i$ together with *Fork*$_i$ and *Fork*$_{i-1}$ (we assume that index arithmetic is cyclic, i.e., $1-1=n$ and $n+1=1$). Thus, acquiring a set of resources is specified as synchronising with the corresponding entities in an interaction. After *Philosopher*$_i$ has picked his/her forks up, he or she eats, releases the forks, spends some more time thinking, and the whole process is repeated again.

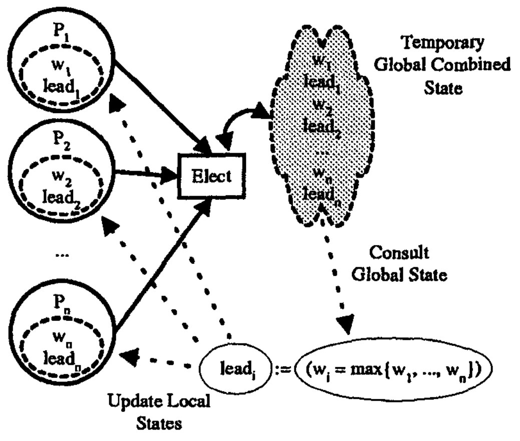
## 2.3 Multiparty Communication

We illustrate the notion of multiparty communication by means of the leader election problem, which is a multi–process communication problem that consists of a number of processes that are able to execute an algorithm, but there is no a priori candidate to run it. Therefore, an election under the processes needs to be held. The criterion they use to select a leader is quite simple: each of them is supposed to have a different natural weight $w_i$ in the system, e.g. its net address, and the leader is the process $P_i$ satisfying $w_i = \max_{1 \leq j \leq n} \{w_j\}$.

The usual solution to this problem, using two–party interactions, consists of arranging the processes in a unidirectional ring where only pairs of neighboring processes can exchange their weights and calculate a local maximum. These maximums are propagated in the ring so that after $n-1$ rounds the global maximum has been calculated. The problem here is that synchronizing the whole set of processes so that each one passes its local maximum at the right moment is quite tricky. If we used multiparty communication, all of the processes would synchronise and have access to the weights other processes have simultaneously.

```
behaviour P_i ::
  /* Local variables */
  { w_i: natural; lead_i: boolean }

  w_i := a weight;
  Elect{
    lead_i := w_i = max{w_1, ..., w_n};
  };
  [ lead_i → execute algorithm ]
```



$P_1$
$w_1$
$lead_1$
$P_2$
$w_2$
$lead_2$
...
$P_n$
$w_n$
$lead_n$
Elect
Temporary Global Combined State
$w_1$ $lead_1$ $w_2$ $lead_2$ ... $w_n$ $lead_n$
Consult Global State
$lead_i := (w_i = \max\{w_1, ..., w_n\})$
Update Local States

Figure 3: A solution to the leader election problem.

An immediate solution to this problem is shown in figure 3. Here, the multiparty interaction *Elect* synchronises all of the processes, allowing them to exchange information and decide which one has to be assigned to the role of leader. When several processes synchronise and interact, a temporary global combined state is built by combining the local states of the processes participating in that interaction so that they can read information in the state of other participants. This way, each process synchronising on *Elect* can read the weights the other processes have, compute the maximum in parallel, compare it to its own weight and store the result of this comparison in its local variable $lead_i$. After interacting, the one that finds itself having the maximum weight executes the appropriate algorithm.

This multiparty communication scheme was incorporated into IP [FF96], but it obviously suffers from several drawbacks because a process having access to the local state of another process is not well–engineered. It is not dangerous because processes are allowed to modify only their own state, but a process needs to know what is the name of the variable that has the piece of information it needs. Thus, a process definition cannot be modified without taking into account what processes depend on its variables. In CAL, we have improved this basic communication scheme by providing each interaction with a set of slots processes participating in it can use to exchange information.

## 3 A Glance at CAL

CAL is a language aimed at describing coordination patterns amongst a number of entities in a way that is independent from computation or other aspects, which are supposed to be coded in a programming language such as Java. Coordination patterns are not dependent on the components 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 well–known debit–card system, which is one of the basic behaviour patterns in a distributed electronic commercial system. 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. This is a problem that can be clearly 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 4 shows a description of the debit–card system in CAL, and its components are analysed in the following subsections.

### 3.1 Describing Interactions

Interactions are defined by means of the following syntax:

```
interaction <name>[<participant descriptions>]
                  (<slot descriptions>)
  where <read/write permissions>
```

Each interaction is given a different name, a number of participants, a number of slots, and some read/write permissions. In the example in figure 4, an interaction called *transfer* has been defined, and it is a three–party interaction that coordinates a terminal that plays the role *term* and two bank accounts that play the roles *source* and *destination*. This interaction is intended to be the channel by means of which a terminal and two bank accounts 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, being the most important difference that a participant does not need to have access to the local state of other participants 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 an adequate message on its screen; the account playing the role *source* can read slot *sum* in order to decide whether it can transfer such a sum, and it can write slot *approval* so that it can store its decision; finally, the destination account can read both slots, but it is not allowed to write any of them.

### 3.2 Describing Behaviours

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

```
interaction transfer[term as Terminal; source, destination as Account]
                    (int sum, boolean approval)
where
  term writes sum, reads approval;
  source writes approval, reads sum;
  destination reads approval, sum;

behaviour Terminal requires
  void  Wait_For_Sale();
  int   Get_Price();
  OID   Get_Customer_Account();
  OID   Get_Merchant_Account();
  void  Report_Result(boolean done);
{
  *[ true →
       Wait_For_Sale();
       /* Try to engage interaction 'transfer' together with customer's
          account as source and merchant's account as destination */
       transfer[-, Get_Customer_Account(), Get_Merchant_Account()] {
         sum = Get_Price();
         Report_Result(approval);
       }
  ]
}

behaviour Account requires
  void Charge(int sum);
  void Pay_In(int sum);
  boolean Authorize_Payment(int sum);
{
  *[ /* Behave as a source account: Try to engage interaction 'transfer'
        together with any terminal or destination account */
     transfer[*, -, *] {
          approval = Authorize_Payment(sum);
          [approval → Charge(sum);];
     } → skip;
    []
    /* Behave as a destination account: Try to engage interaction
       'transfer' together with any terminal or source account */
    transfer[*, *, -]  {
      [approval → Pay_In(sum);];
    } → skip;
  ]
}
```

Figure 4: A description of the debit–card system in CAL.

```
behaviour <name> requires
  interaction <required interactions>
  <required operations>
{
  <behaviour statement>
}
```

Each one is given a different name, and requires a number of interactions to be available in the system, and a number of operations to be implemented by the entities onto which it can be mapped. In the example in figure 4, two behaviour patterns are described: *Terminal*, that describes the behaviour of a terminal, and *Account*, that describes the behaviour of a bank account, both as debtor and creditor.

Pattern *Terminal* requires five operations to be implemented by the entities 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 customer's account and the merchant's account, and *Report_Result*, that can be invoked to report whether a transfer has been done or not. Pattern *Account* requires three operations: *Charge*, to withdraw money from an account, *Pay_In*, to pay money into it, 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 the statements we described in section 2, being the only difference with regard to the interaction statement. In CAL, they are of the form $a[id_1, id_2, \ldots, id_n]\{comm\_stat\}$, where $id_1$, $id_2$, ..., $id_n$ identify the entities with which the entity executing such a statement is interested in cooperating. There are two special identifiers that are denoted by "–" and "*". The former refers to the entity executing this interaction statement, and the latter to any entity.

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 entity that models 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 to inform the clerk if the transfer was carried out.

The behaviour of an *Account* also consists of an infinite loop where engaging in interaction *transfer* is offered twice simultaneously: 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 entity 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 can be viewed as statements that are executed in a critical region where no race conditions can occur. We think that this approach is suitable to describe problems where several entities need to agree and collaborate simultaneously in order to solve a problem because we do not need to design a specific protocol to avoid race conditions. On the contrary, it is implicitly included in the multiparty interaction mechanism.

```
class bankAccount  {
  private int balance;

  public void updateBalance(int sum) {
    balance += sum;
  }

  public boolean getBalance() {
    return balance;
  }
}

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

Figure 5: Mapping for behaviour *Account* onto class *BankAccount*.

### 3.3 Mapping Behaviours onto Object Classes

Behaviour patterns are abstract because they describe how an entity 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 figure 5. 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. Fortunately, the expression *getBalance*() >= *sum* implements it.

## 4 Related Work

There are several coordination languages aimed at coordinating a number of entities so that they can achieve a common goal, but they usually rely on the client/server model. We think that this model might not be adequate in some cases because a protocol is usually needed in order to coordinate the activity of a number of entities so that they can achieve their common goal. The programmer is responsible for ensuring that the entities involved in such cooperative activities do not interfere with or suffer interference from other entities. Designing such protocols is sometimes more time and effort consuming than implementing local computations. The multiparty interaction model has been paid much attention because it can deal with such situations in a simple, elegant way, and this is the reason why we bet on it.

Other authors have proposed coordination techniques based on event notification protocols. For instance, in [MHSA99] a model for coordination called Coordinated Roles is presented. It is based on event notification protocols that allow to detect events that occur in active objects. In this model, a behaviour pattern is organised as a hierarchy of coordination components that are responsible for coordinating the activities of several computational or coordination components. Events can occur when an object receives a message, when it begins or finishes processing a message or when an abstract state is reached, and coordination components may control the sequences of events that can occur, thus describing a coordination protocol. Furthermore, a coordination component may delay one of the components it coordinates until a certain event occurs, thus synchronising it with other components.

Proposals such as [Frø96] or [Arb96] have some similarities with the one we have described above. They are usually based on defining coordinating components that coordinate the activities of a number of objects. Our proposal diverges from such approaches in the sense that we describe coordination as an aspect that can be mapped onto entities as if it was a wrapper. Furthermore, multiparty communication is also a simple yet powerful mechanism we can use to exchange information amongst coordinated entities.

## 5 Conclusions and Future Work

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

The main drawback of our proposal is that the mapping between behaviour patterns and object classes is one–onto–one. Thus, related patterns can not be developed, improved or maintained separately. For instance, we cannot describe patterns for debtor accounts and creditor accounts separately, but it would be desirable. Currently, we are working on composing and specialising behaviour patterns, but our results are not concluding, yet.

We have implemented CAL using Java and a CORBA Service for dealing with multiparty interaction. Its main drawback is that it is not fault–tolerant. Unfortunately, very little research has been carried out in this field. [ZS99] is the only article we know that presents a general scheme for implementing fault-tolerant multiparty interactions in a distributed object–oriented environment. We are working so that we can incorporate the ideas in this article into our CAL prototype.

## References

[Arb96] F. Arbab. The IWIM model for coordination of concurrent activities. *Lecture Notes in Computer Science*, 1061, 1996.

[FF96] N. Francez and I. Forman. *Interacting processes: A multiparty approach to coordinated distributed programming*. Addison–Wesley, 1996.

[Frø96] S. Frølund. *An Actor-Based Approach to Synchronization – Coordinating Distributed Objects*. The MIT Press, Cambridge, Massachusetts, 1996.

[JS96] Y.J. Joung and S.A. Smolka. A comprehensive study of the complexity of multiparty interaction. *Journal of the ACM*, 43(1):75–115, January 1996.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97 Proceedings*, pages 220–242. Lecture Notes in Comnputer Science, Springer-Verlag, 1997.

[MHSA99] J.M. Murillo, J. Hernández, F. Sánchez, and L.A. Álvarez. Coordinated roles: Promoting reusability of coordinated active objects using event notification protocols. In *Proceedings of COORDINATION '99*, number 1594 in LCNS, pages 53–68, Amsterdam, The Netherlands, 1999. Springer–Verlag.

[PA98] G. Papadopoulos and F. Arbab. Coordination models and languages. In *Advances in Computers*, volume 46. Academic Press, 1998.

[ZS99] A.F. Zorzo and R.J. Stroud. A distributed object-oriented framework for dependable multiparty interactions. In L. Meissner, editor, *Proceeings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34.10 of *ACM Sigplan Notices*, pages 435–446, N.Y., November 1999. ACM Press.