A FRAMEWORK FOR ASPECT–ORIENTED MULTIPARTY COORDINATION

José A. Pérez, Rafael Corchuelo, David Ruiz and Miguel Toro
Dep. de Lenguajes y Sistemas Informáticos
Universidad de Sevilla, Spain
{jperez,corchu,druiz,mtoro}@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 no fit into the scope of a class must be satisfied. In this paper, we show that interaction amongst a number of objects can also be described separately from functionality by means of the CAL language, and present a framework that provides the needed infrastructure. It is innovative because it supports open multiparty interactions.

Keywords: Multiparty interactions, coordination algorithms, aspect–oriented languages .

1. INTRODUCTION

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 difficult problems such as the well-known inheritance anomaly. Unfortunately, aspect–oriented languages such as COOL, RIDL [Lopes, 1998], ASPECTJ [Lopes and Kiczales, 1998] or AML [Irwin et al., 1997] do not succeed in isolating computation from interaction with other objects in a system. COOL, for instance, allows us to define synchronisation 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 relies on classical point–to–point communication primitives and, thus, emphasises a number of objects exchanging binary messages 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 [Papadopoulos and Arbab, 1998], and many researchers have centred their effort on the novel multiparty interaction model, which has been introduced in many languages [Joung and Smolka, 1996]. It has also attracted the attention of the designers of the well–known Catalysis method

[D'Souza and Wills, 1999]. 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.

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 coordinated behaviour separately from computation so that functional code can be kept clean.

In this paper, we present a framework for implementing multiparty coordination as an aspect. It has been used to implement the CAL language [Corchuelo et al., 2000], which is, to the best of our knowledge, the first multiparty coordination–aspect language to appear in the literature. The rest is organised as follows: section 2 sketches the interaction model on which CAL relies; section 3 presents the framework that provides the needed run–time infrastructure to implement it, and section 4 shows a brief description of the underlying algorithms; section 5 glances at other authors' work, and section 6 shows our conclusions and future work.

2. CAL INTERACTION MODEL

CAL [Corchuelo et al., 2000] is a language aimed 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.

To achieve this, CAL uses multiparty interactions as the sole mean to express coordination. In CAL, each interaction has a name, a number of roles and a number of slots associated to it.

The name of the interaction is a string which unambiguously identifies an interaction in the system. When an object is ready to coordinate with other objects, it offers to participate in one or more interactions by mean of their interaction names.

So, 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 a *enablement*.

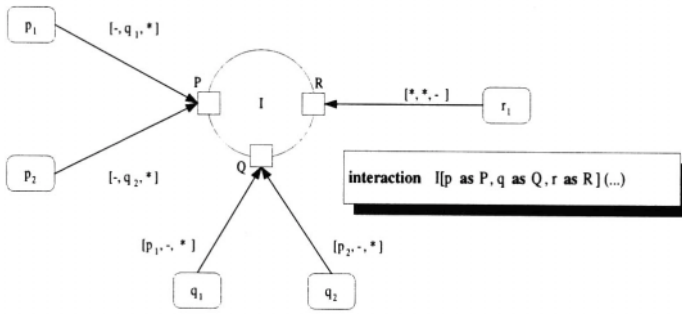

Figure 1. Offers made to the coordinator of an interaction I.

Figure 1 shows an example system with an interaction called I amongst three objects that must play roles P, Q and R. Objects p_1 and p_2 make offers to play role P, objects q_1 and q_2 make offers to play role Q, and object r_1 makes an offer to play role R. The objects p_1 and p_2 require that role Q must be played by q_1 and q_2 respectively, and vice versa. Neither p_1, q_1, p_2 nor q_2 establish constraints about what object should play role R. In the other hand, the object r_1 accepts that roles P and Q could be played by any object.

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.*

When an enablement of an interaction is executed, the objects in it can communicate by means of the interaction slots. A slot is a shared variable among the objects in the enablement which is created when the enablement is executed. These slots make up a local state that simulates the temporary global combined state in IP [Francez and Forman, 1996], 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. Obviously, a multiparty interaction delays an object that tries to read a slot that has not been initialized yet by another object.

3. A FRAMEWORK FOR ASPECT–ORIENTED MULTIPARTY COORDINATION

We have carefully designed a framework that offers a number of high-level services for providing CAL run–time support. This framework is extensible so that new middlewares or coordination algorithms may be easily incorporated.

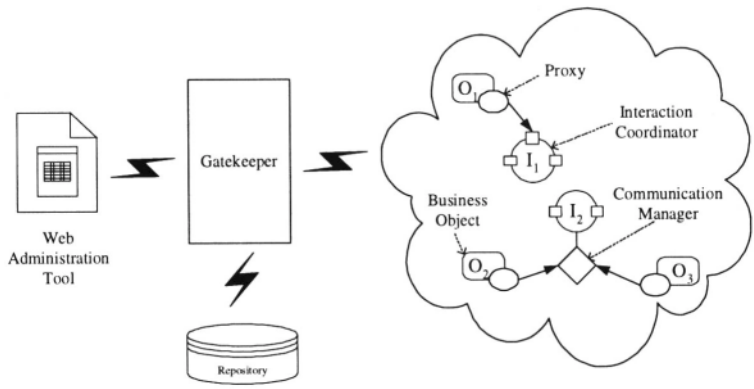

Figure 2. The architecture of our solution.

Figure 2 shows a snapshot of a running system that sketches the architecture of our implementation, which 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 umpiring amongst conflicting ones, i.e., interactions that cannot be executed simultaneously because they involve a common object. The algorithms we use are presented in section 4.

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 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. They are also responsible for coping with faults during multiparty communication [Zorzo and Stroud, 1999].

At a first glance, it might seem that the gatekeeper is a bottleneck component of our framework, 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 objects at run–time is worthwhile because this draws a clear line between the functionality they encapsulate and the way they interact with others.

4. IMPLEMENTING MULTIPARTY COORDINATION

In this section, we describe the algorithm we have devised to implement multiparty coordination. It is called α, and it is scattered amongst coordinators and proxies. It is responsible for the following tasks:

Enablement detection: The offers received from proxies are analysed sequentially to find sets of objects that agree in participating in an interaction, i.e, enablements.

Enablement selection: When one or more enablements have been detected, as many as possible should be executed simultaneously. Thus, an election under conflicting enablements needs to be held.

That is the reason why we have split α into two parts called α*–solver* and α*–core* that are further explained in the following subsections.

4.1. The α*-solver* Algorithm

α*–solver* is responsible for enablement detection, and the ideas behind it can be presented by means of the example in figure 1 at page 163.

α*–solver* processes offers as they arrive and form a consolidation graph that consists of tuples such as $[p_1, (q_1), ()]$. This tuple represents the offer made by p_1 and it means that it wants to play role P in interaction I, requires q_1 to play role Q, and does not care about which object should play role R. We say that role P is consolidated in this tuple, whereas role Q requires object q_1 and role R accepts any object.

Figure 3 shows the consolidation graph built by our algorithm as the offers made by the objects in our example arrive at the coordinator responsible for interaction I. Assume that the offer made by p_1 arrives first so that α*–solver* constructs a consolidation graph with only one node $[p_1, (q_1), ()]$. 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 $[p_1, (q_1), ()]$, a connecting node of the form $[p_1, q_1, ()]$ is

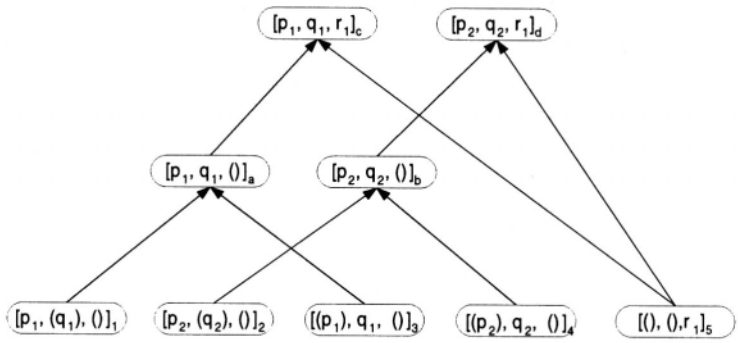

Figure 3. Consolidation graph for the system in figure 1.

added. It indicates that both p_1 and q_1 want to participate in interaction I and agree in committing to it together with any object playing role R. Notice that no enablement is found until object r_1 makes its offer. When this happens, two enablements are found simultaneously, but, unfortunately, they are conflicting because they share r_1.

In order to present α–*solver* code, we first need to define a consolidation operator that is defined on both the tuples of the consolidation graph and its elements. We refer to this operator as $\odot$ and it is defined on tuples as $[e_1, e_2, \ldots, e_n] \odot [e'_1, e'_2, \ldots, e'_n] = [e_1 \odot e'_1, e_2 \odot e'_2, \ldots, e_n \odot e'_n]$. It is defined on the elements of a tuple by means of the following axioms:

1 $p_1 \odot (p_2) = (p_2) \odot p_1 = p_1$, as long as $p_1 = p_2$

2 $(p_1) \odot (p_2) = (p_2) \odot (p_1) = (p_1)$, as long as $p_1 = p_2$

3 $p \odot () = () \odot p = p$

4 $(p) \odot () = () \odot (p) = (p)$

5 $() \odot () = ()$

The entry–point to α–*solver* is the routine called *ProcessOffer*(T, G) presented in figure 4. (T is the offer being processed, and G is the current consolidation graph, which is built incrementally as new offers are received.) It simply iterates over the set of roots of graph G and calls routine *Search*(T, R) presented in figure 5 on each one. (T represents the current offer, and R a root of graph G.) This routine first tries to consolidate tuples T and R, and if it is possible, the consolidated tuple is returned and inserted in the graph as a parent of both T and R. Otherwise, a recursive search is performed in the subgraph whose root is the left child of R. If a consolidation $left$ is found there, it then recursively tries to find a new consolidation of $left$ with a tuple in the subgraph

```
ProcessOffer (T: Tuple; G: Graph): Set of Tuple
 enablements: Set of Tuple
 roots: Set of Tuple
 C: Tuple
 enablements ← ∅
 roots ← the roots of G
 add T to G as an unconnected leaf
 for every root R in graph G do
  C ← Search (T, R)
  if C is not null and every role in C is consolidated then
   enablements ← enablements ∪ {C}
  end if
 end for
 return enablements
end ProcessOffer
```

Figure 4. α*–solver* entry–point.

```
Search (T: Tuple; R: Tuple): Tuple
 result: Tuple;
 left, right: Tuple;
 if T and R can consolidate then
  result ← T ⊙ R
  let result be the parent tuple of T and R
 else
  if T is not a leaf then
   left ← Search (T, left child of R)
   if left is not null then
    right ← Search (left, right child of R)
    result ← (right is not null ? right: left)
   else
    right ← Search (T, right child of R)
    result ← (right is not null ? right: null)
   end if
  else
   result ← null
  end if
 end if
 return result
end Search
```

Figure 5. α*–solver* recursive consolidation function.

whose root is the right child of *R*. If such a consolidation if found, then it is returned because it is the most consolidated tuple that has been found; else, *left* is returned. If no consolidation is found while examining the left subgraph of *R*, then the right subgraph is also explored. If no consolidation is found, then *null* is returned.

4.2. The α–*core* Algorithm

The α–*core* algorithm takes the enablements detected by α–*solver* and selects for execution as many as possible. If a enablement is rejected, that is because it conflicts with another that has already been selected. An enablement can be conflicting in two different ways:

- It may be *locally* conflicting with another enablement of the same interaction. This is the case of enablements c and d in the example in figure 3.
- It may be *remotely* conflicting with another enablement of another interaction.

For the sake of simplicity, in this section, we assume that each coordinator is responsible for only one enablement. We drop this restriction in the next section.

The ideabehind α–*core* is quite simple. Shared objects are considered to be shared resources amongst the coordinators which coordinate the enablement where they appear. In order for an enablement to be selected, its coordinator must ensure exclusive access to every shared object participating in it. So, a coordinator must *lock* every shared object in its enablement before it can be selected for execution. For instance, consider the example in figure 6: there are two coordinators for two interactions I_1 and I_2 that are conflicting because p_2 is offering participation in both. Figure 7 shows a scenario for this system, and table 1 describes the messages α–*core* uses.

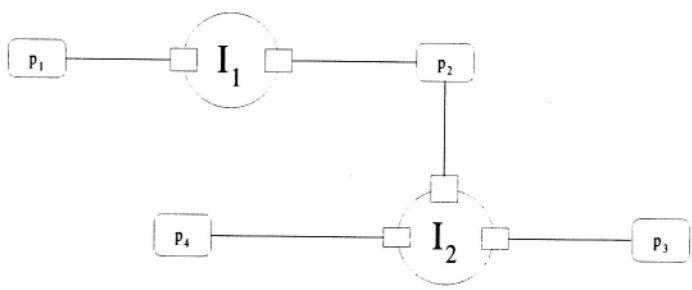

Figure 6. Two conflicting coordinators with one enablement each one.

In this scenario, p_1 is the first object ready to participate in I_1. Since it is only interested in this interaction, it notifies its offer to coordinator I_1 by means of a *PARTICIPATE* message, and then waits for a *START* message before beginning the execution of this interaction. Assume that p_2 gets then ready to participate in either I_1 or I_2. Since it offers participation to more than one coordinator, it sends two *OFFER* messages by means of which the

Table 1. Messages used by α–*core*.

Message	*Description*
$ACKREF$	Message sent from a coordinator to acknowledge it has got a $REFUSE$ message from an object.
$LOCK$	Message sent from a coordinator to a shared object to request exclusive access.
$OFFER$	Message sent from an object to a number of coordinators to offer them participation in the interactions they manage.
OK	Message sent from an object to a coordinator to notify that it grants it exclusive access. This message is sent as a reply to a $LOCK$ message.
$PARTICIPATE$	Message sent from an object to only one coordinator to inform it that it is only interested in the interaction it manages.
$REFUSE$	Message sent from an object to a coordinator to cancel an offer.
$START$	Message sent from a coordinator to a locked object to notify it that the interaction it manages may start.
$UNLOCK$	Message sent from a coordinator to a shared object to release exclusive access.

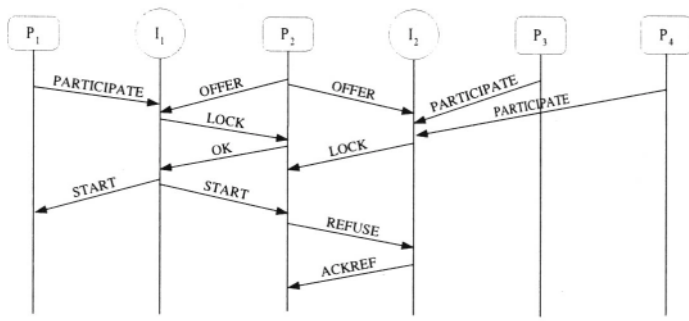

Figure 7. A possible scenario for the system in figure 6.

coordinators that receive them can infer that this object is shared with others, although they need not know each other directly.

When coordinator I_1 processes this offer, it detects enablement $[p_1, p_2]$. Since p_2 is a shared object it tries to lock p_2 by sending it a *LOCK* message. There is no need to lock p_1 because this object is interested in only one interaction; thus it is not a shared object. Assume that p_3 and p_4 decide then to participate in I_2 and send a *PARTICIPATE* message to its coordinator. It then detects enablement $[p_2, p_3, p_4]$ and tries to lock p_2, too. Unfortunately, the *LOCK* message sent to p_2 by I_1 is received before the *LOCK* from I_2 arrives. Thus, p_2 notifies I_1 that it accepts to be locked by means of an *OK* message,

but it shall not acknowledge the lock message received later from coordinator I_2, but shall record it, just in case I_1 cannot be executed. Coordinator I_2 waits until it gets an answer from p_2 before going on, thus it cannot lock an object if another lock is still pending. When I_1 receives the *OK* message, it knows that it has exclusive access to its shared object, and thus sends a *START* message to p_1 and p_2. When the shared object p_2 receives the *START* message from I_1, it knows that it can execute that interaction and cancels the offer made to I_2 by sending it a *REFUSE* message that is acknowledged by means of an *ACKREF* message.

Therefore, the idea behind *α-core* consists of making the coordinators compete to lock their shared objects, allowing an enablement to be executed as long as its corresponding coordinator has acquired exclusive access to all of its shared objects. The problem is that locks need to be carried out carefully in order to avoid deadlocks. We use an idea proposed in [Coffman et al., 1971]: *α-core* assumes that coordinators may sort their objects according to a given immutable property, e.g., their net address or UUID, so that lock attempts are made in increasing order. This idea was proven not to produce deadlocks and it is quite effective.

4.3. Putting *α-solver* and *α-core* Together in an Efficient Way

In the previous section, we sketched the way *α-core* resolves conflicts amongst conflicting enablements, but we assumed an important restriction: every coordinator coordinated just one enablement. This is not realistic because, as we showed in the example in section 4.1, an offer may lead to several enablements.

The solution to this problem is straightforward because *α-core* can easily be generalised to coordinate an arbitrary number of enablements: if a coordinator finds more than one enablement, it just executes *α-core* for every enablement. For example, the coordinator of interaction I in figure 1 would execute *α-core* for the enablement $[p_1, q_1, r_1]$ and for the enablement $[p_2, q_2, r_1]$.

Notice that the coordinator of I should send two *LOCK* messages to object r_1: one for the enablement $[p_1, q_1, r_1]$ and another one for the enablement $[p_2, q_2, r_1]$. Obviously, it does not make sense that a coordinator needs to send more than one *LOCK* message to the same object because it compromises efficiency. The solution to this problem is that every coordinator associates a *lock count* to every shared object that is initialised to zero. When it needs to lock an object for the first time, it sends it a *LOCK* message and set its lock count to one. When the coordinator needs to lock again an object with a lock count greater than zero, it just increases by one its counter, and no *LOCK* message is sent again to it. Symmetrically, when an object needs to be unlocked, its

lock count is decreased by one. If the resulting counter is greater than zero, no *UNLOCK* message should be sent since the object is already locked. If the lock count reaches zero, then no enablement is locking the object, so an *UNLOCK* message must be sent then.

5. RELATED WORK

Several solutions to implement multiparty interactions have been proposed in the literature. We have found a variety of centralised and distributed techniques for dealing with multiparty synchronisation and exclusion. For instance, synchronisation may be solved by means of polling, message–counts [Bagrodia, 1989], or auxiliary resources such as tokens [Chandy and Misra, 1988]; the exclusion problem may be solved by using time stamps, auxiliary resources [Bagrodia, 1989], probabilistic techniques [Joung, 2000], and so on. Our α algorithm solves synchronisation by means of its enablement detection algorithm (α–*solver*), and the exclusion problem by the selection algorithm (α–*core*), which locks objects in a given order. This idea was presented in [Coffman et al., 1971] in the context of operating systems. Although it did not work well in this field, because resources of an operating system are difficult to sort and usually cannot be requested in increasing order, it has been successfully applied in α–*core*.

The simplest algorithm can be found in [Francez and Forman, 1996], for instance, and it consists of using a central scheduler responsible for every interaction. In [Corchuelo et al., 1999], a slightly modified version of the basic algorithm was presented. In this solution, there is a manager per interaction responsible for detecting enablement, but also a central scheduler responsible for umpiring amongst conflicting managers. Although this solution is suitable for some problems in the traffic control arena, the central conflict resolutor is not adequate in the general case.

The first distributed algorithms for coordination were produced in the context of CSP, but they were restricted to two–party interactions. Later, the problem became of great interest, and Bagrodia devised the EM and MEM algorithms [Bagrodia, 1989], which are the most cited in this field. EM uses a number of interaction managers, each one responsible for managing a subset of interactions. When an object wants to participate in a number of interactions, it sends *READY* messages to the corresponding managers, which use a message-count technique for detecting enablement; mutual exclusion is achieved by means of a circulating token that allows the manager having it to execute as many non–conflicting interactions as possible. Having a circulating token has several drawbacks because it amounts to additional network load, even if no interaction is enabled, which may be quite problematical in bus networks. The token also needs to circulate amongst managers in a given order, thus organising

them in a unidirectional ring, which may lead to a situation in which a manager can never execute one of the interactions for which it is responsible, because it never gets to have the token at the right time.

For these problems, Bagrodia devised a modified version of EM that was called MEM. It combines the synchronisation technique used in EM with the idea of using auxiliary resources to arbitrate between conflicting interactions. The exclusion problem is solved by mapping the multiparty exclusion problem onto the well–known dining philosophers problem. Thus, conflicting managers are considered to be philosophers that need to acquire shared forks placed between them in mutual exclusion. MEM has an important drawback because the number of forks a manager has to acquire to guarantee mutual exclusion increases steadily as the number of potentially conflicting interactions increases. This implies that the probability of acquiring all the forks decreases accordingly, even if the managers are not conflicting at run–time.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have explored the aspect–oriented paradigm, the multiparty interaction model, and how programming distributed systems may benefit from both. The multiparty interaction model captures three main issues in the design of distributed systems: synchronisation, communication and exclusion, and we have presented a framework to implement it as an aspect.

A variety of solutions exist in the literature, and ours is innovative in the sense that we do not require the set of active objects in a system to be fixed and known in advance. In addition, coordinators need not know all of the processes in a system, and objects are not directly dependent on each other, which is an important drawback in other proposals. This way, our solution can be easily applied in open contexts such as the Internet where multiparty interactions can be used to coordinate an arbitrary number of objects.

References

[Bagrodia, 1989] Bagrodia, R. (1989). Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Transactions on Software Engineering*, 15(9): 1053–1065.

[Chandy and Misra, 1988] Chandy, K. and Misra, J. (1988). *Parallel Program Design: A Foundation*. Addison–Wesley.

[Coffman et al., 1971] Coffman, E., Elphick, M. J., and Shoshani, A. (1971). System deadlocks. *Computing Surveys*, 3(2):67–78.

[Corchuelo et al., 2000] Corchuelo, R., Pérez, J., and Toro, M. (2000). A multiparty coordination aspect language. *ACM Sigplan*, 35(12):24–32.

[Corchuelo et al., 1999] Corchuelo, R., Ruiz, D., Toro, M., Prieto, J., and Arjona, J. (1999). A distributed solution to multiparty interaction. In *Recent Advances in Signal Processing and Communications*, pages 318–323. World Scientific Engineering Society.

[D'Souza and Wills, 1999] D'Souza, D. and Wills, A. (1999). *Objects, Components, and Frameworks with UML: The Catalysis Approach.* Addison–Wesley, Reading, Mass., 1 edition.

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

[Irwin et al., 1997] Irwin, J., Loingtier, J.-M., Gilbert, J. R., and Kiczales, G. (1997). Aspect–oriented programming of sparse matrix code. In Springer-Verlag, editor, *Proceedings of the 1997 International Scientific Computing in Object–Oriented Parallel Environments*, volume 1343 of *Lecture Notes in Computer Science,* pages 249–256. Springer–Verlag.

[Joung, 2000] Joung, Y. (2000). Two decentralized algorithms for strong interaction fairness for systems with unbounded speed variability. *Theoretical Computer Science*, 243(1–2):307–338.

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

[Lopes, 1998] Lopes, C. (1998). *D: A Language Framework for Distributed Programming*. PhD thesis, Xerox Palo Alto Research Center.

[Lopes and Kiczales, 1998] Lopes, C. and Kiczales, G. (1998). Recent developments in AspectJ. In Demeyer, S. and Bosch, J., editors, *Object-Oriented Technology: ECOOP'98 Workshop Reader*, volume 1543 of *Lecture Notes in Computer Science*, pages 398–401. Springer–Verlag.

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

[Zorzo and Stroud, 1999] Zorzo, A. F. and Stroud, R. J. (1999). A distributed object-oriented framework for dependable multiparty interactions. *ACM Sigplan*, 34 (10):435–446.