An order-based algorithm for implementing multiparty synchronization

José A. Pérez[*,†], Rafael Corchuelo and Miguel Toro

The Distributed Group, University of Seville, Spain

SUMMARY

Multiparty interactions are a powerful mechanism for coordinating several entities that need to cooperate in order to achieve a common goal. In this paper, we present an algorithm for implementing them that improves on previous results in that it does not require the whole set of entities or interactions to be known at compile- or run-time, and it can deal with both terminating and non-terminating systems. We also present a comprehensive simulation analysis that shows how sensitive to changes our algorithm is, and compare the results with well-known proposals by other authors. This study proves that our algorithm still performs comparably to other proposals in which the set of entities and interactions is known beforehand, but outperforms them in some situations that are clearly identified. In addition, these results prove that our algorithm can be combined with a technique called synchrony loosening without having an effect on efficiency.

KEY WORDS: multiparty synchronization; coordination; concurrency control

1. INTRODUCTION

Since Hoare's work on how to coordinate sequential processes [1], interactions have become a fundamental feature in many frameworks for distributed computing because they allow entities to synchronize and communicate in an abstract way, independently from the underlying primitives used to implement interactions. (Hereafter, the term 'entity' refers to any single-threaded computing artefact, e.g. a thread, a process, an object or a component.) Thus, programmers can focus on the higher-level protocols their business objects have to implement, instead of the low-level details concerning synchronization and communication primitives.

*Correspondence to: José A. Pérez, ETSI Informática, Avda. de la Reina Mercedes s/n, Sevilla E-41012, Spain.
†E-mail: jperez@lsi.us.es

Contract/grant sponsor: Spanish Ministry of Science and Technology; contract/grant number: TIC-2003-02737-C02-01, FIT-150100-2001-78

Languages such as CSP [1], Ada [2], SR [3,4] and some Java concurrency frameworks [5,6] support bipartite interactions only, but this can be easily extended to an arbitrary number of entities [7–19]. Such interactions are usually said to be multiparty, and they provide a higher level of abstraction because they allow the expression of complex cooperations as atomic actions that can be refined automatically into efficient message-passing or shared-memory protocols [20–28].

Since entities are single-threaded, they cannot execute more than one interaction at a time. Implementing multiparty interactions thus requires an algorithm to be devised to synchronize several entities and execute interactions in mutual exclusion. The execution of an interaction usually entails the exchange of data amongst the entities that have synchronized on that interaction, and this is usually carried out by means of a temporary shared state that acts as a blackboard [12]. In the literature, there are a variety of centralized and distributed techniques for dealing with multiparty synchronization and exclusion. For instance, synchronization may be solved by means of polling [22], message-counts [20], or auxiliary resources like tokens [21]; the exclusion problem may be solved by using priorities [26], time stamps [23], auxiliary resources [20,26], probabilistic techniques [24,25], queues [27], and so on. Very little work, however, has been reported on how to implement the multiparty exchange of data [28].

Although the implementation techniques have reached a rather elaborate status, most of them suffer from a common drawback: the set of entities and interactions needs to be known at compile time, and, as reported in [23], none of them are well suited to be adapted to a dynamic setting in which new entities may be created or some of them may terminate. The proposal in [23] differs in that the set of entities may change at run-time, but they all need to know each other, which simply moves the gathering of information from compile-time to run-time. Care must be taken to ensure that new entities that are created or destroyed concurrently do not interfere with each other, which complicates the solution. Furthermore, some algorithms, e.g. [25], cannot be applied to systems in which an entity works locally for increasing periods of time.

In this paper, we propose α-core[‡]. It is an algorithm for implementing multiparty synchronization that addresses the problems mentioned above and solves them efficiently. It improves on other approaches in that it does not require the whole set of entities or interactions to be known either at compile-time or at run-time; interactions only need to be known by the entities that may participate in them. Furthermore, it can deal with both terminating and non-terminating systems or systems in which entities can perform local computation for arbitrarily long periods of time. We also present the results of an experimental analysis on the performance of the algorithm, and compare it with some state-of-the-art proposals. The experiments show that our proposal performs comparably to other algorithms, but outperforms them in some situations that are clearly identified. In addition, our algorithm can be used in a system to which synchrony loosening [12] has been applied, but, in contrast to other proposals, this does not have an effect on its performance.

The rest of the paper is organized as follows. In Section 2 we motivate the need for multiparty interactions, provide a thorough description of their semantics, and present an example that may help realize their benefits. In Section 3 we present a formal definition of the algorithm, and prove that it is

[‡] α-core is the first algorithm in a series of proposals we are developing to implement $\mathcal{CAL}$ [29]. This aspect-oriented coordination language supports the Partners-Named Enrolement Model [23,30,31], and its implementation relies on an algorithm called α, which is composed of two subalgorithms referred to as α-core and α-solver. Subsequent versions of the algorithm are referred to as β and γ, and they deal with fault tolerance and fairness issues, but they are still under hot development.

correct in Section 4. We compare our proposal with other authors' work in Section 5, and evaluate it empirically in Section 6. Finally, we report on our main conclusions and future research directions in Section 7.

2. MULTIPARTY INTERACTIONS IN A NUTSHELL

Our goal in this section is to provide the reader with a good understanding of multiparty interactions. First, we motivate the need for such interactions, then we present a description of their semantics, and, finally, we present a classical example that may help realize their benefits.

2.1. Motivation

Primitives such as remote procedure call or message passing are the *de facto* industrial standard for communicating entities, and new materializations such as SOAP [32,33] are sprouting out at an increasing pace. Roughly speaking, such primitives may be viewed as directional binary interactions. Unfortunately, they are difficult to apply in a context in which several entities need to cooperate simultaneously in order to achieve a common goal [11,12]. Such problems include: transferring money from one bank to another by means of a point of sales terminal (three entities) [11,34]; paying taxes on-line (three entities in Spain: a taxpayer, the Exchequer, and Spain's Certification Authority); filtering in e-commerce [35] (a customer, a filter system, and several service providers); or reaching a virtual agreement in an auction sale (multiple entities).

Furthermore, most object-oriented analysis and design methods recognize the need for coordinating several entities and provide designers with tools to model such multi-object collaborations. Different terms are used to refer to them: entity diagrams [36]; process models [37]; message connections [38]; data-flow diagrams [39]; collaboration graphs [40]; scenario diagrams [41]; or collaborations [8,18]. These proposals are accompanied by a rich set of examples that show the adequacy of such modelling tools in fields including finance, telecommunication, insurance, manufacturing, embedded systems, process control, flight simulation, travel and transportation, or systems management. Recently, this need for multi-entity interaction has also been recognized in the field of multiagent systems [42–46] and parallel programming [19], where they have been proven to achieve optimal parallelism.

It is thus not surprising that multiparty interactions have also attracted the attention of researchers in the field of Java concurrency frameworks. For instance, in [15] the authors proposed a framework that supports barrier synchronization, which may be viewed as a simple form of multiparty interaction; this framework has become the basis for the Java Specification Request #166 within the Java Community Process [47]. In [11] the authors presented a framework that supports one-to-*n* interactions in which an object may request several objects to interact with it atomically. The proposal in [6] leverages SR [3,4] and extends Java to provide a rich concurrency model that includes Ada-like biparty interactions. Such interactions were generalized to the multiparty case in [48]. In [5], another Java framework that supports CSP-like biparty interaction is also presented.

Unfortunately, none of these Java frameworks can be viewed as a full implementation of multiparty interactions since either interactions are not symmetric [6,11], or they cannot guard the execution of an alternative in a multi-choice command [11,15]. Ada-like interactions are not symmetric since they involve two roles, one assumed by a fixed server and the other by various clients. If interactions cannot

guard an alternative, the resulting entities can offer to participate in only one interaction at a time, which results in a reduction of concurrency in the general case [23,49].

Beyond Java and Ada, there are many research languages that support multiparty interactions. A taxonomy can be found in [13], but the most recent are IP [12] and $\mathcal{CAL}$ [29]. IP is intended to have a dual role because, on the one hand, it provides a flexible notation to describe distributed, concurrent or parallel systems [19], but, on the other hand, it is also amenable to formal reasoning. $\mathcal{CAL}$ is a language that aims at separating the interaction model from the basic functionality an object encapsulates, thus promoting reusability [7].

Summing up, the idea behind multiparty interactions is to allow a number of entities to cooperate so that they can carry out a common task. Although such cooperations can be described in terms of low-level primitives, the concerns of reusability and abstractness argue for a solution in which such cooperations can be expressed in terms of higher-level multiparty interactions that can be refined automatically into low-level, efficient protocols, thus relieving the programmer from the burden of coding, testing and maintaining them.

2.2. Semantics

In this section, we introduce the main features of the multiparty interaction model with which we deal. It is a standard description except for the fact that the identities of the entities that may participate in an interaction need not be known beforehand, which makes the model more general and introduces several problems that have not been addressed by other authors.

The terms *entity* or *participant* refer to any computing artefact that is able to perform local computations and decide autonomously when to participate in an interaction. Multiparty interactions are usually provided as guards in multi-choice commands, so that an entity may offer to participate in several interactions at a time, although only one can be executed at a time since entities are single-threaded. This is not a shortcoming, since an object with two threads may be viewed as two related entities. This model is thus also well suited to coordinate processes, threads, objects or components.

Each interaction is identified by a unique *name*, and has a fixed number of participants referred to as its *cardinality*. Note that the set of entities that may participate in an interaction need not be known in advance. Often, we refer to interactions with cardinality n as n-party interactions. In general, n can be greater than or equal to two, although the results we show work well with single-party interactions. For an n-party interaction to become *enabled*, n participants need to be *offering participation* in it. Once several entities have been coordinated, communication depends completely on the constructs provided by the language under consideration, but most have been designed so that it is relatively easy to determine data communication requirements [12].

Roughly speaking, a multiparty interaction can be viewed as a set of data exchange actions that need to be executed jointly and coordinately by a number of entities, each of which must be ready to execute its own action so that the interaction can occur. An attempt to participate in an interaction delays an entity until all other participants are available, and after an interaction is executed, the participating entities exchange some data and continue their local computations on their own accord.

It is worth noting that an interaction being enabled does not amount to its execution. Figure 1 depicts a simple system composed of four participants, a two-party interaction and a three-party interaction. Note that participant P_1 is offering participation in I_1, whereas P_3 and P_4 are offering participation in I_2, and P_2 is offering participation in either I_1 or I_2. This means that these interactions cannot be

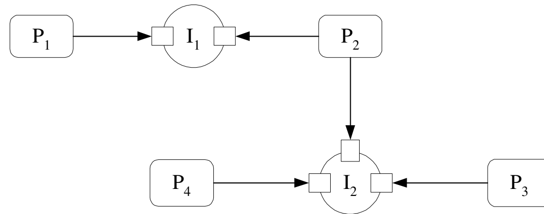


Figure 1. A system composed of four participants and two conflicting interactions.

executed simultaneously and they are said to be *conflicting*. Thus an election needs to be held to decide which should be executed. The one that is executed is referred to as the *winner interaction* and the others as the *loser interactions*.

In summary, every interaction has three associated events: (i) *enablement*, which occurs when all of its participants are offering to participate in it; (ii) *execution*, which occurs when an enabled interaction is selected for execution (probably out of a set of conflicting interactions); (iii) *disablement*, which occurs either after execution or when a participant withdraws its offer because it executes a conflicting interaction.

In order to implement multiparty interactions, we need to devise an algorithm that satisfies the following properties [20].

- *Exclusion:* when an interaction in a group of conflicting interactions is executed, the rest become disabled.
- *Progress:* if an interaction is enabled, it will eventually become disabled, because it is executed or an entity offering participation in it executes another interaction.
- *Synchronization:* if an entity executes an interaction, the remaining entities participating in that interaction will execute it.
- *Idleness:* an entity that is performing local computations cannot execute any interaction, i.e. it has to be idle to offer any of them.

2.3. A classical example

We illustrate multiparty synchronization by means of the well-known Dining Philosophers problem [50], which is a classic multiparty synchronization problem. It consists of a number of 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 up both forks to eat. This problem is the core of a large class of problems in which an entity needs to acquire a set of resources in mutual exclusion. This situation can be the case of a debit card system in which there is a set of point-of-sales terminals and several computers that hold customer or merchant accounts. When a clerk inserts a debit card into a terminal (philosopher), a three-party interaction needs to be carried out to transfer funds from a customer's account (left fork) to the corresponding merchant's account (right fork).

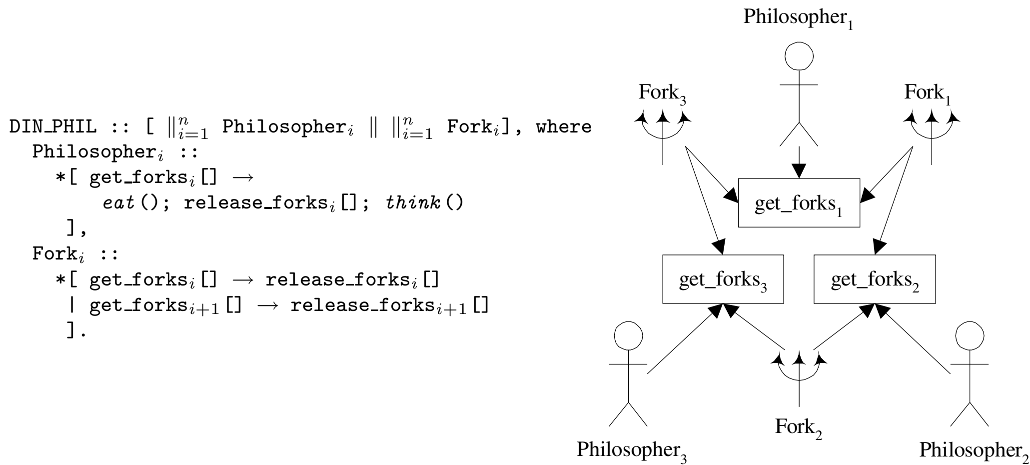


Figure 2. A solution to the Dining Philosophers problem in IP.

The obvious solution to this problem, using two-party interactions, consists of picking up both forks in sequence. Nevertheless, a problem arises if each philosopher grabs the fork on their right, and then waits for the fork on their left to be released. In this case, a deadlock has occurred, and all philosophers starve. In [51], the authors proved that assuming no means of communication amongst philosophers other than through information attached to their forks, any solution in which all philosophers are programmed identically must have a possibility of deadlock. Thus, correct solutions must rely on some distinction to be made amongst the philosophers. These solutions are usually not scalable or reusable since the distinction a philosopher has to implement depends heavily on the topology of the problem under consideration.

If we used multiparty interactions, the solution would be conceptually simpler since each philosopher would pick up their two forks at a time when no deadlock could arise. Figure 2 shows a solution to this problem in IP. The philosophers are represented by processes $Philosopher_i$, and the forks by $Fork_i$ ($i = 1, 2, \ldots, n$). $Philosopher_i$ eternally tries to get their associated forks by interacting 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 resource is specified as synchronizing with the corresponding processes in an interaction. After $Philosopher_i$ has picked their forks up, they eat, release the forks and spend some more time thinking. (Note that interactions get_forks_i and get_forks_{i+1} are always in conflict when they are both enabled, but only one can be executed.)

3. OUR PROPOSAL: α-CORE

In this section we present a comprehensive description of our algorithm. First, we present an overall picture of the protocols it implements, then we describe the notation we use and the algorithm.

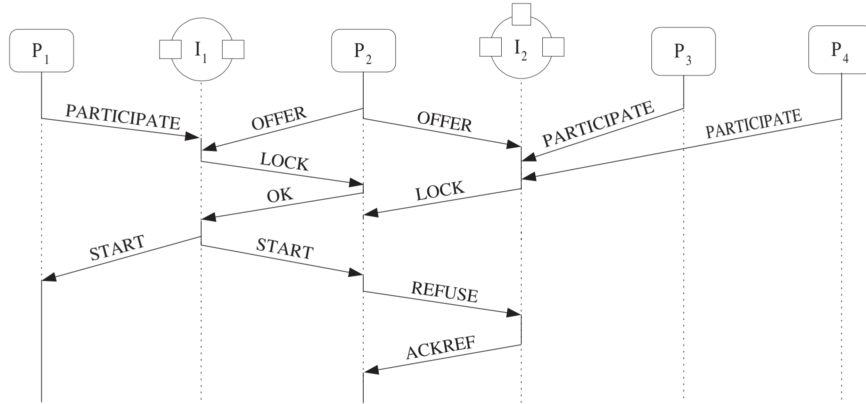


Figure 3. A possible scenario for the system in Figure 1.

Finally, we present some remarks on some subtle details. The introduction to the algorithm in Section 3.1 was presented previously in [52].

3.1. The overall picture

α-core was devised to detect enabled interactions and select amongst conflicting interactions in a simple, effective way. We assume that each entity participating in an interaction runs independently, and that each interaction is managed by its own *coordinator*. (We refer to coordinators using the same name assigned to the interaction they manage wherever this is not misleading.) The communication between coordinators and participants is modelled by means of *asynchronous messages* because this communication primitive is available on almost every platform. An entity that offers participation in more than one interaction is considered as a *shared resource* amongst the coordinators responsible for managing those interactions. For an interaction to be executed, its corresponding coordinator must ensure exclusive access to all of its participants, i.e. coordinators must compete for their shared participants so that they can execute the interaction they manage.

Figure 3 shows a typical scenario for the system depicted in Figure 1, and Table I describes the messages α-core uses. In this scenario, P_1 is the first entity 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 then gets 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 those coordinators can infer that this participant is shared with others, although they need not know each other directly.

As soon as coordinator I_1 receives the offers from its two participants, it detects that I_1 is enabled and tries to lock P_2 by sending it a *LOCK* message. There is no need to lock P_1 because this participant is interested in only one interaction; thus it is not a shared participant. Assume that P_3 and P_4 then decide

Table I. Messages used by α-core.

Message	Description
PARTICIPATE	If a participant is interested in only one interaction, it uses this message to inform its corresponding coordinator.
OFFER	If a participant is interested in more than one interaction, it then sends *OFFER* messages to their coordinators.
LOCK	Message sent from a coordinator to a shared participant to request exclusive access to it.
OK	Message sent from a participant to a coordinator to notify that it grants it exclusive access. This message is sent as a reply to a *LOCK* message.
START	Message sent from a coordinator to a locked participant to notify it that the interaction it manages may start.
UNLOCK	Message sent from a coordinator to a shared participant to release exclusive access.
REFUSE	Message sent from a participant to a coordinator to cancel an offer or to inform that it cannot be locked.
ACKREF	Message sent from a coordinator to acknowledge it has got a *REFUSE* message from a participant.

to participate in I_2 and send a *PARTICIPATE* message to its coordinator. It then detects that it is also enabled, 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 does not acknowledge the *LOCK* message received later from coordinator I_2, although it has to record it just in case I_1 cannot be executed. Coordinator I_2 waits until it gets an answer from P_2 before continuing; that is, it cannot lock a participant if another lock is still pending. When I_1 receives the *OK* message, it knows that it has exclusive access to its shared participants, and sends a *START* message to both P_1 and P_2. When the shared participant 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 with an *ACKREF* message.

Therefore, the idea behind α-core consists of locking shared participants, and an interaction may be executed as long as its corresponding coordinator has acquired exclusive access to all of its shared participants. The problem is that entities have to be locked carefully to avoid deadlocks. We use the Hierarchical Ordering of Sequential Processes Principle presented in [50,53] to solve this problem since α-core assumes that coordinators may sort their participants according to a given immutable property, e.g., their net address, their universally unique identifier (UUID) [54] or their universal/uniform resource identifier (URI) [55], so that lock attempts are made in increasing order. This idea is also the basis of the well-known two-phase lock protocol in the database world [11,56].

3.2. Notation and assumptions

We describe our algorithm by means of state transition diagrams, which are finite deterministic automata composed of a finite number of states and atomic transitions amongst them. The initial state

is labelled by an arrow without origin, and every transition is labelled by a specification of the form

$$\frac{[guard]\ message}{action}$$

message denotes the message that causes a transition to occur if the guard holds. If it is omitted, a transition depends solely on its guard. *action* is the list of sentences to be executed if a transition occurs. We use assignments, and sentences of the form $Send(recipient, message)$ to send messages. The word *Sender* refers to the identifier of the coordinator or participant that sent the message that caused a transition to occur. Finally, *guard* is a boolean expression over the state of the automaton that evaluates it. The state of an automaton is a set of local variables that we declare in a box. For a transition to occur, its corresponding message must be received and the guard must hold. If a guard is omitted it is assumed to be *true* by default.

We only need to make the three usual safety and liveness assumptions about the underlying message-passing system: (i) if a message is sent, then it is received at the destination within a finite period of time; (ii) if a message is received, then it was sent previously; (iii) if messages m_1 and m_2 are received in this order from same origin, then m_1 was sent before m_2. Regarding the system itself, we only need to assume that the number of entities and interactions is finite, although it may be arbitrarily large.

3.3. α-core in a participant

The state transition diagram for participants is shown in Figure 4, and a short description of its variables is shown in Table II. Initially, participants are performing local computations in state *ACTIVE* until they assign the set of interactions in which they are interested to variable *IS*. If $|IS| = 1$, a participant is interested in one interaction only, so it sends a *PARTICIPATE* message to its coordinator (transition 2). Since the target state of this transition is *LOCKED*, this means that the participant gets locked automatically once the offer is made. If $|IS| > 1$, it then sends an *OFFER* message to each coordinator whose identifier is in *IS* and reaches the *WAITING* state (transition 1).

In the *WAITING* state, a participant waits to be locked and leaves it when it receives a *LOCK* message from a coordinator that then becomes a *prospective winner* (transition 3). On receipt of this message, participants acknowledge with an *OK* message and reach the *LOCKED* state. Subsequent *LOCK* messages are temporarily stored in *locks* without acknowledging their receipt (transition 4). This behaviour allows the prospective winner to have exclusive access to a shared participant until it can reach a decision and decide whether the interaction it manages may start or not.

In the *LOCKED* state, a participant waits for the prospective winner coordinator to send it an *UNLOCK* message indicating that it is a loser and the interaction it manages will not be executed, or a *START* message indicating that it is the winner and the interaction it manages can start. We have to deal with the former case depending on the availability of other coordinators in the *locks* set. If there is at least one coordinator in this set (transition 5), one of them is chosen arbitrarily and becomes the new prospective winner. (It is important to choose randomly because if it is not the case, variability amongst consecutive executions would only depend on network latency.) Then, an *OK* message is sent to it, and the participant remains in the *LOCKED* state. If $unlocks = \emptyset$ (transition 6), this means that no coordinator can be elected as a new prospective winner, and the participant has to return to the *WAITING* state. During this transition, offers are re-sent to the coordinators that have already rejected the participant because it has not executed any interaction and is still interested in them.

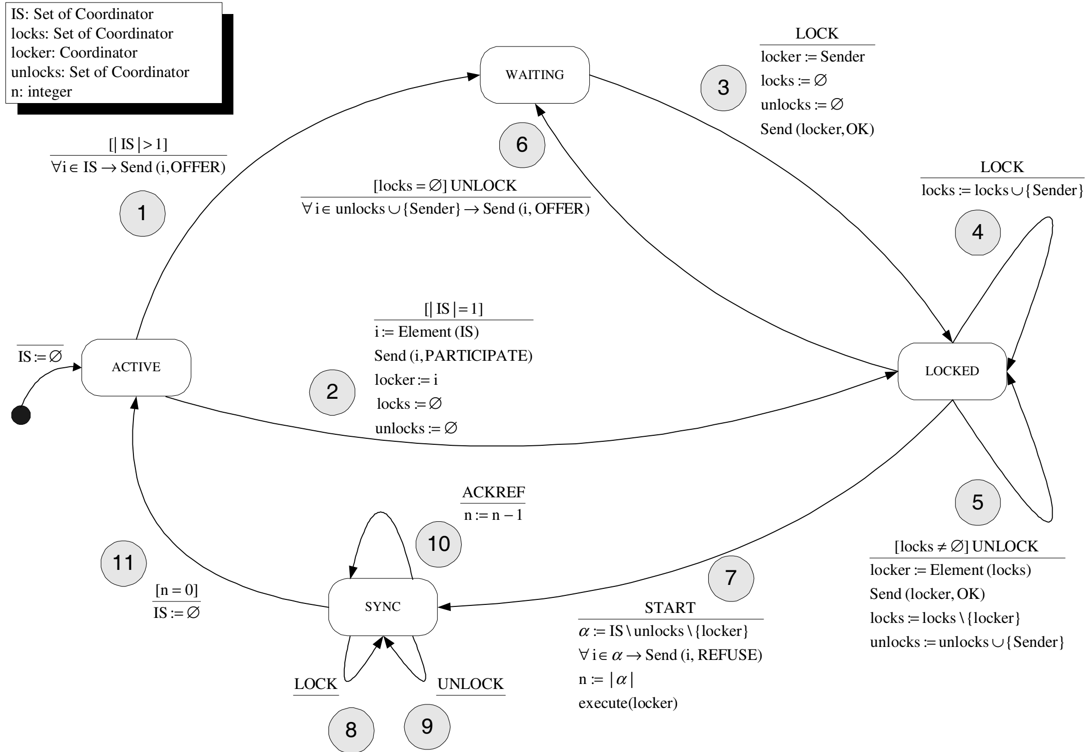


Figure 4. α-core state diagram for participants.

When a *START* message is received from the current prospective winner coordinator in the *LOCKED* state (transition 7), the interaction this coordinator manages has been selected for execution and may start. However, a *REFUSE* message has to be sent to the coordinators of the interactions in the set *IS*, except for the winner and those that are known to be losers (those that sent an *UNLOCK* message), in order to inform them that this participant is no longer interested in them. Note that on receiving a *START* message, the participant cannot return to the *ACTIVE* state immediately because it first has to wait for the coordinators to which it has sent a *REFUSE* message to acknowledge its receipt by means of an *ACKREF* message. This is essential, because if the participant executed the interaction and offered participation in other interactions immediately after, *ACKREF* messages from the coordinators that were sent a *REFUSE* message before might be misunderstood if another interaction was executed too soon. Consequently, the participant has to remain in an intermediate state called *SYNC* until every coordinator that was sent a *REFUSE* message acknowledges its receipt.

Table II. Variables used by α-core in participants.

Variable	Description
IS	A set with the identifiers of the coordinators that manage the interactions in which a participant is interested.
locker	A variable that identifies the coordinator that has locked a participant, i.e. the *current prospective winner coordinator*.
locks	A set of identifiers that allow us to refer to each coordinator from which a participant has received a *LOCK* message *after* the one received from the current prospective winner coordinator.
n	A counter used to determine when every involved coordinator has acknowledged a *REFUSE* message from a participant.
unlocks	A set of identifiers that allow us to refer to prospective winner coordinators that had to release a participant. When a participant receives an *UNLOCK* message from its prospective winner coordinator, it is said that it has been *rejected* by the coordinator.

LOCK and *UNLOCK* messages may also be received in state *SYNC*. For the moment, it is difficult to realize the reason why, because they stem from some intricacies in the coordinators that will become clear in Section 3.5.

3.4. α-core in a coordinator

Figure 5 shows the state transition diagram for coordinators, and Table III gives a brief explanation of its variables. A coordinator may be in two different states called *ACCEPTING* and *LOCKING*, and accepts offers from participants whilst it is in the former. Those offers can be made by means of *PARTICIPATE* messages, in the case of non-shared participants (transition 2), or by means of *OFFER* messages, in the case of shared participants (transition 1). In the former case, since non-shared participants lock themselves automatically, they are directly stored in the set *locked*. In the latter, the participant that made the offer is waiting to be locked, so the coordinator stores its identifier in set *shared*. In both cases, n, the offer counter, is increased.

ACCEPTING state is a bit more complex than it might seem, because whilst a coordinator is still waiting for offers, it may receive a *REFUSE* message from a participant that wants to cancel its offer because another coordinator in which it was interested has reacted more quickly (transition 3). Since this participant is shared, the coordinator needs to remove it from the *shared* set because no *LOCK* message has been sent to it. An *ACKREF* message is sent in order to acknowledge the *REFUSE* message, and the offer counter is decreased. The reason why n is decreased conditionally stems from an intricacy of α-core, and we explain this further in Section 3.5. When n equals the cardinality of the interaction a coordinator manages in the *ACCEPTING* state, it becomes enabled and there are two possible continuations depending on the existence of shared participants. If there are no such participants (transition 4), it does not have to compete for any of them; thus, it can send them a *START* message immediately so that they can start executing the interaction it manages. Otherwise (transition 5), it needs to lock shared participants carefully to avoid deadlocks. The solution we use

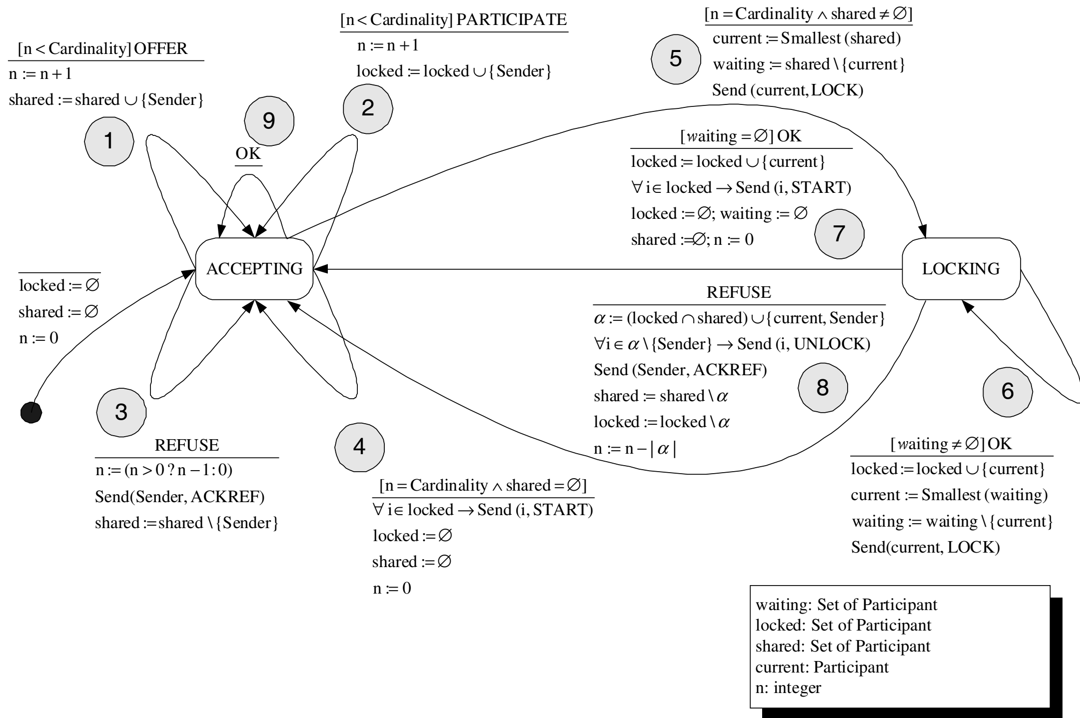


Figure 5. State transition diagram for coordinators.

was suggested in [50,53], and it consists of establishing a strict total order relationship amongst shared resources so that they have to be acquired in increasing order. Thus, for a coordinator to lock participant P, it must have previously locked every shared participant Q such that $Q \prec P$, $\prec$ being the total order under consideration; if it cannot lock it because it receives a *REFUSE* message, then it must unlock every preceding participant. In both cases, the answer may be delayed for some time if the participant is currently locked by another coordinator. Therefore, in transition 5, the smallest participant in the *waiting* set is removed from it and sent a *LOCK* message. This participant may reply with an *OK* message, indicating that it has been locked, or a *REFUSE* message, indicating that it is no longer interested in the interaction it manages.

Processing an *OK* message depends on the number of participants waiting to be locked. If there is a participant in the *waiting* set (transition 6), the participant that has sent the *OK* message is stored in the *locked* set, and the next waiting participant in the *waiting* set is selected to be locked. Otherwise, every shared participant has been locked. Thus, the interaction can be executed and a *START* message is sent to every participant (transition 7).

Table III. Variables used by α-core in coordinators.

Variable	Description
current	The participant that a coordinator is currently trying to lock, i.e. a *LOCK* message has been sent to it, but no reply has yet arrived.
n	An offer counter, which is used to detect when the interaction a coordinator manages is enabled.
shared	The set of participants shared with other coordinators.
waiting	The set of participants not yet locked.

Processing a *REFUSE* message is quite different (transition 8), because this means that one of the shared participants that has not yet been locked cancels its offer or the shared participant currently being locked refuses to be locked because it has committed to another interaction. Thus, all the shared participants that are already locked have to be unlocked in order to prevent deadlocks. The coordinator then sends an *UNLOCK* message to every locked participant, to the participant currently being locked (unless it is the sender), and an *ACKREF* to the participant that sent the *REFUSE* message before reaching the *ACCEPTING* state again.

In the preceding discussion, we have intentionally left out transition 9 and some details of transition 3 that are further explained in the following section.

3.5. Remarks

Transitions 8 and 9 in participants, as well as transitions 3 and 9 in coordinators stem from some intricate features of our algorithm. In this section, we justify the reason why such transitions are necessary.

Figure 6 shows a scenario that proves that *LOCK* messages may be received whilst a participant is in the state *SYNC* (transition 8). Note that once coordinator I_2 has received the *OFFER* message from P_2 and the proper *PARTICIPATE* message from P_3, it sends a *LOCK* message to try to lock its first participant, but this message may take an arbitrary long time to reach its recipient. Thus, when P_2 receives the *START* message from coordinator I_1, it enters state *SYNC* and sends a *REFUSE* message to coordinator I_2 to cancel its offer. Note that when P_2 sends this message, the *LOCK* message from I_2 has not yet arrived to P_2 due to transmission delays, thus proving that a *LOCK* message may be received whilst a participant is in the state *SYNC*. Note also that such a message may not be processed after the *ACKREF*, because we are assuming that messages sent from the same origin to the same destination are processed in order.

Figure 7 shows a scenario that proves that *UNLOCK* messages may also be received whilst a participant is in the state *SYNC* (transition 9). Note that once coordinator I_2 has received the *OFFER* messages from P_1 and P_2, it sends a *LOCK* message to try to lock P_1, which is its smallest participant. Assume that this message takes an arbitrarily long time to reach its recipient. In this context, both I_1 and I_3 might succeed in locking their shared participants, and they would send *REFUSE* messages to I_2 in order to cancel their offers. When I_2 receives the first *REFUSE* message from P_2, it executes transition 8 and sends an *UNLOCK* message to P_1 and an *ACKREF* message to P_2. Note that the

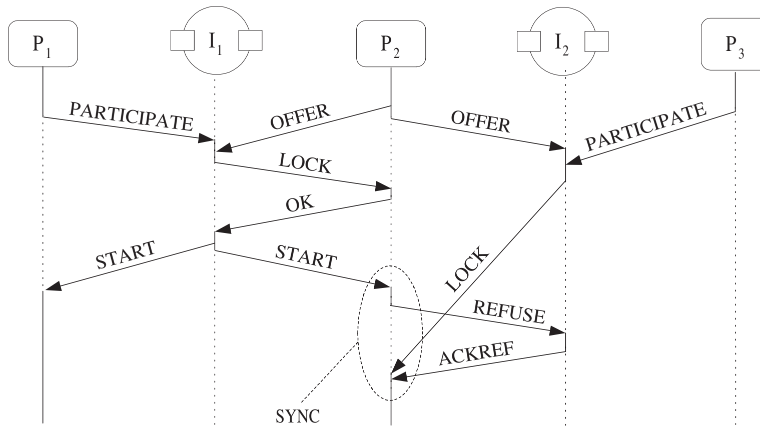


Figure 6. A scenario that proves that a *LOCK* message may be received whilst a participant is waiting for *ACKREF* messages in state *SYNC*.

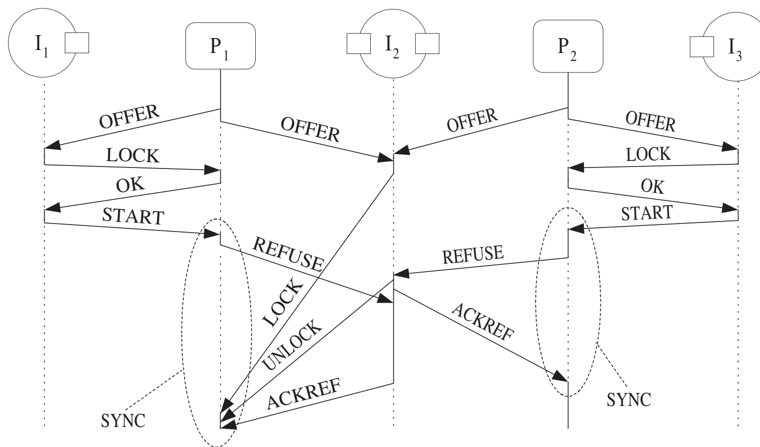


Figure 7. A scenario that proves that *UNLOCK* messages may be received whilst a participant is in the state *SYNC*, as well as *REFUSE* or *OK* messages whilst a coordinator is in the state *ACCEPTING*.

UNLOCK message must necessarily arrive at P_1 after the *LOCK* message, and that they both must be received whilst P_1 is in the state *SYNC*, because messages sent from the same origin are processed in order and P_1 will not leave the state *SYNC* until it receives an *ACKREF* message.

The scenario in Figure 7 also justifies the need for transitions 3 and 9 in coordinators. On receipt of the first *REFUSE* message, I_2 reaches state *ACCEPTING* immediately. Note that on receipt of a *REFUSE* message from P_2, coordinator I_2 does not know if participant P_1 has accepted to be locked or not, thus its answer must be processed in the state *ACCEPTING*, which justifies the need for transitions 3 and 9 in coordinators.

It should also be pointed out that counter n is decreased in transition 3 if and only if it is strictly greater than zero, which might seem superfluous as long as we have defined n as a counter that records the number of offers a coordinator has got, so that each offer should be cancelled by exactly one *REFUSE* message. The reason is that if transition 8 occurs in a situation in which $current \neq Sender$, then the answer from participant *current* has not yet been processed. However, *current* is removed from set *shared* and n is decreased as if the answer from *current* had been processed. This is the reason why we need to be careful before updating counter n.

It is also worth noting that a coordinator in state *LOCKING* can only receive an *OK* message from the participant to which it sent a *LOCK* message. In other words, a coordinator in state *LOCKING* cannot receive an *OK* message from a participant other than *current*. The reason is that a coordinator may receive an *OK* message whilst it is in state *ACCEPTING* if it received a *REFUSE* message from a participant other than *current* whilst it was in the state *LOCKING*. Then, if the *current* participant had already sent its *OK* message, such message would be received by the coordinator when it is already in the state *ACCEPTING*. The coordinator cannot change again to the *LOCKING* state without having received this *OK* message because to do so, it needs to receive again an *OFFER* message from every shared participant (including that which sent the *OK* message) and such an *OFFER* is sent *after* the *OK* message. (Recall that we assume that the network does not alter the order of messages between a source and a destination.)

In order to avoid these sets of intricate transitions, we might have added a *CANCEL* message to sort out the difference between a *REFUSE* message indicating that a participant does not accept to be locked and another indicating that a participant decides to cancel an offer because it commits to another interaction. The problem is that adding this message leads to a solution with more states and transitions, which is undesirable.

4. CORRECTNESS

In this section, we prove that α-core is correct, i.e. it guarantees the exclusion, progress, synchronization and idleness properties presented in Section 2.2. We begin with a number of preliminary definitions and lemmas, and then present and prove the theorems we need to state that our algorithm is correct.

4.1. Preliminaries

Definition 1. Let $\{P_1, P_2, \ldots, P_n\}$ be a finite set of participants. $\prec$ is a total order over this set, or simply an order, as long as $\prec$ is a transitive, irreflexive binary relation over this set.

Definition 2. We define a predicate that is denoted by $locks(I, P)$ and holds as long as coordinator I has sent a *LOCK* message to participant P and has received an *OK* message from it.

Definition 3. We define a predicate that is denoted by $waits(I, P)$ and holds as long as coordinator I has sent a *LOCK* message to participant P, but has not yet received a reply. Note that $waits(I, P)$ holds if P has replied, but its answer has not yet been received by I.

As a referee pointed out, it is important to realize that $locks(I, P)$ implies $\neg wait(I, P)$, and $waits(I, P)$ implies $\neg locks(I, P)$.

Definition 4. We denote the number of *LOCK* messages sent out by a given coordinator since it left state *ACCEPTING* for the last time as η_{LOCK}, i.e. η_{LOCK} is reset to zero every time a coordinator leaves state *ACCEPTING* and reaches the state *LOCKING*.

Definition 5. We denote the number of *OK* messages received by a given coordinator since it left the *ACCEPTING* state for the last time as η_{OK}, i.e. η_{OK} is reset to zero every time a coordinator leaves state *ACCEPTING* and reaches state *LOCKING*.

Lemma 1. $\eta_{LOCK} = \eta_{OK} + 1$ *is an invariant when coordinator I is in state LOCKING.*

Proof. Proving it when I enters state *LOCKING* after leaving state *ACCEPTING* is straightforward if we analyse transition 5. I sends out its first *LOCK* message during this state transition, thus $\eta_{LOCK} = 1$ and $\eta_{OK} = 0$ by definition, because when I enters state *LOCKING* for the first time after leaving state *ACCEPTING* it has not yet received an *OK* message.

From now on, transition 6 is the only transition that keeps coordinator I in state *LOCKING*. This transition is executed on receipt of an *OK* message, but coordinator I sends out a *LOCK* message during it. Therefore, each time η_{OK} increases, so does η_{LOCK}, thus keeping the property $\eta_{LOCK} = \eta_{OK} + 1$ invariant. □

Lemma 2. $\eta_{LOCK} + |waiting| = |shared|$ *is an invariant when coordinator I is in state LOCKING.*

Proof. According to transition 5, it is clear that $\eta_{LOCK} = 1$ and $waiting = shared \setminus \{current\}$ when I enters state *LOCKING* after leaving state *ACCEPTING* (being *current* its smallest shared participant). Consequently, $|waiting| = |shared| - 1$ in this situation, so $|waiting| = |shared| - \eta_{LOCK}$ and $\eta_{LOCK} + |waiting| = |shared|$.

From now on, transition 6 is the only transition that keeps coordinator I in state *LOCKING*. This transition sends out a *LOCK* message and also removes a participant from set *waiting*. Therefore, $|waiting|$ decreases by one and η_{LOCK} increases by one, thus keeping $\eta_{LOCK} + |waiting| = |shared|$ invariant. □

Lemma 3. *The maximum number of OK messages received by coordinator I before leaving state LOCKING is* $\eta_{OK}^{\max} = |shared| - 1$.

Proof. According to Lemmas 1 and 2, we can infer that $\eta_{OK} + 1 + |waiting| = |shared|$, so $\eta_{OK} = |shared| - |waiting| - 1$. Given that *shared* does not change in state *LOCKING*, η_{OK} reaches its maximum value when $waiting = \emptyset$ and I leaves state *LOCKING*. Therefore, $\eta_{OK} = |shared| - 1$ in this situation. □

Lemma 4. *Let $\prec$ be an order over the participants of our system, and let I be a coordinator whose set of shared participants is $\{P_1, P_2, \ldots, P_k\}$ $(k \geq 1)$. Without loss of generality, we can assume that $P_1 \prec P_2 \prec \ldots \prec P_k$. In this context, if $waits(I, P_i)$, then $\forall 1 \leq j < i \cdot locks(I, P_j)$.*

Proof. *LOCK* messages are sent out during transitions 5 and 6. In transition 5, the recipient is the smallest participant in set *waiting*. Note that each time an *OFFER* message is received in state *ACCEPTING*, its sender is added to set *shared*. Thus, when coordinator I leaves state *ACCEPTING* during transition 5, the smallest participant in set *shared*, i.e. P_1, is set as the *current* participant and a *LOCK* message is sent to it. The rest of participants in set *shared* are stored in set *waiting*. Therefore, when coordinator I enters state *LOCKING* after leaving state *ACCEPTING*, this lemma holds because before receiving an answer from P_1, predicate $waits(I, P_1)$ holds and there is no participant P_0 such that $P_0 \prec P_1$, i.e. $\forall 1 \leq j < 1 \cdot lock(I, P_j)$ holds trivially.

In transition 6, the recipient is also the smallest participant in set *waiting*, but note that each time transition 5 or transition 6 is executed, the smallest participant in set *waiting* is removed from it. Also note that a new *LOCK* message is not sent unless an *OK* message has been received from the participant that was sent the previous *LOCK* message. Let P_i be the smallest participant in set *waiting* before transition 6 is executed. Consequently, on receipt of an *OK* message, if there is at least one participant in set *waiting*, coordinator I sends a new *LOCK* message and $waits(I, P_i)$ holds until it receives an answer from P_i, as well as $locks(I, P_1), locks(I, P_2), \ldots, locks(I, P_{i-1})$. □

Lemma 5. *Let P be a participant that has offered participation to a number of coordinators $\{I_1, I_2, \ldots, I_n\}$ $(n \geq 1)$ and receives its first LOCK message since it left state ACTIVE from coordinator I_i $(1 \leq i \leq n)$. In this context, P will eventually send an OK message to I_i.*

Proof. This lemma follows directly from transition 3. If participant P is willing to participate in several interactions, it has then reached state *WAITING* by means of transition 1, and it remains in that state until it receives a *LOCK* message from a coordinator. Let this coordinator be I_i $(1 \leq i \leq n)$. On receipt of this message, P leaves state *WAITING* immediately and sends an *OK* message to I_i during transition 3. Consequently, P eventually replies, and I_i will eventually get an *OK* message. □

Lemma 6. *Let I be a coordinator that sends a LOCK message to participant P, and let $\prec$ be an order over the set of participants of our system. In this context, coordinator I will receive a REFUSE or an OK message from P within a finite period of time.*

Proof. Given that we are assuming that the underlying network is reliable and every message reaches its recipient within a finite period of time, if coordinator I does not receive an answer from P after sending it a *LOCK* message, it implies that P has received this message whilst it was in state *LOCKED* and remains in that state eternally, i.e. it does not receive any *UNLOCK* messages. (Otherwise, Lemma 5 guarantees that a participant in state *WAITING* replies in finite time.) We prove that this is impossible by *reductio ad absurdum*.

If P does not answer in a finite time, then there must be a coordinator I_1 such that $locks(I_1, P)$. If I_1 does not release its lock in a finite time, then there exists a participant P_1 such that $waits(I_1, P_1)$, and P_1 does not send an answer to I_1 in a finite time. Consequently, we can infer that there exists a coordinator I_2 and a participant P_2 such that $locks(I_2, P_1)$ and $waits(I_2, P_2)$. In other words, there is

an infinite wait chain of the following form:

$$\begin{aligned} & waits(I, P) \wedge locks(I_1, P) \wedge \\ \wedge \quad & waits(I_1, P_1) \wedge locks(I_2, P_1) \wedge \\ \wedge \quad & waits(I_2, P_2) \wedge locks(I_3, P_2) \wedge \\ \wedge \quad & \dots \\ \wedge \quad & waits(I_i, P_i) \wedge locks(I_{i+1}, P_i) \wedge \\ \wedge \quad & \dots \end{aligned}$$

Given that the number of coordinators and participants in a system is finite, although it may be arbitrarily large, this sequence must be circular, i.e. there must exist a $k \geq i$ such that $waits(I_k, P) \wedge locks(I_k, P_{k-1})$. According to Lemma 4, we can infer the following properties:

$$\begin{aligned} & waits(I_1, P_1) \wedge locks(I_1, P), \text{ then } P \prec P_1 \\ & waits(I_2, P_2) \wedge locks(I_2, P_1), \text{ then } P_1 \prec P_2 \\ & \dots \\ & waits(I_k, P) \wedge locks(I_k, P_{k-1}), \text{ then } P_{k-1} \prec P \end{aligned}$$

This is a contradiction because we assume that $\prec$ is an order amongst the participants of our system. Thus, there cannot exist a P such that $P \prec P$. In other words, the contradiction stems from the fact that using the transitivity of $\prec$, we conclude that it is reflexive, which contradicts Definition 1. □

4.2. Exclusion property

Theorem 1. *Let $\{I_1, I_2, \dots, I_n\}$ be a set of conflicting enabled coordinators. (Hereafter, we say that a coordinator is enabled/disabled if the interaction it manages is found to be enabled/disabled.) α-core guarantees that only one of the interactions is executed and the rest become disabled.*

Proof. In short, what we have to prove is that if coordinator I_i finds itself enabled, succeeds in locking its shared participants and executes transition 7, then any other enabled, conflicting coordinator I_j must become disabled and execute transition 8. We prove this by *reductio ad absurdum*.

Let S be the set of participants shared between coordinators I_i and I_j $(i \neq j, 1 \leq i, j \leq n)$. If both coordinators execute transition 7, then every shared participant in S must have sent an *OK* message to both I_i and I_j. This is not possible because a participant cannot dispatch two messages at a time. Furthermore, we can assume that the *LOCK* message from I_i is dispatched before the *LOCK* message from I_j without loss of generality. Now, we can sort out two cases.

- On the one hand, if P had been in state *WAITING* when it received this message, it would have left this state, it would have sent an *OK* message to I_i, and it would have dispatched the *LOCK* message from I_j in state *LOCKED*. In this state, this *LOCK* would not have been answered with an *OK* message unless I_i had unlocked P, but, in this case, I_i would have not executed, which contradicts our hypothesis.
- On the other hand, if P had been in state *LOCKED* when it received the *LOCK* message from I_i, the answer would have been delayed until the current locker would have released its lock because

of a *START* message or an *UNLOCK* message. In the former case, participant P would have sent I_i and I_j a *REFUSE* message, which contradicts the hypothesis because I_i and I_j would not have been able to execute in this case. In the latter case, P would have selected another prospective winner coordinator. If neither I_i nor I_j had been selected, the situation would have been the same as that described in the previous scenario. If I_i had been selected, it would have been sent an *OK* message, but, in that case, when I_i had sent a *START* message (recall that we are assuming that both I_i and I_j execute transition 7), P would have executed transition 7 (do not confuse transition 7 in participants with transition 7 in coordinators) and would have sent a *REFUSE* message to I_j, which contradicts our hypothesis because, in that case, I_j would not have executed. □

4.3. Progress property

Theorem 2. *Let I be a coordinator that finds itself enabled. α-core guarantees that it will eventually become disabled.*

Proof. An enabled coordinator may become disabled because it executes in mutual exclusion or because one of its participants commits to another interaction. When a coordinator finds itself enabled and has no shared participants ($shared = \emptyset$) it immediately sends *START* messages to its participants and then becomes disabled (transition 4). Otherwise, if the enabled coordinator has shared participants, it enters state *LOCKING* (transition 5), and it remains enabled until it leaves this state, because it receives an *OK* message from each of its shared participants or a *REFUSE* message. Roughly speaking, we need to prove that α-core does not deadlock.

The proof follows directly from Lemmas 3 and 6. Each time coordinator I enters state *LOCKING*, being it because it executes transition 5 or 6, it sends a *LOCK* message to a shared participant. Lemma 6 guarantees that coordinator I will receive an answer in a finite time. Thus, if it receives a *REFUSE* message, it executes transition 8 and returns to state *ACCEPTING* immediately; if it receives an *OK* message, it remains in state *LOCKING*, but this cannot happen more than $\eta_{OK}^{\max} = |shared| - 1$ times according to Lemma 3.

Therefore, once coordinator I finds itself enabled and enters the *LOCKING* state, it will eventually abandon this state and will thus become disabled in finite time. □

4.4. Synchronization property

Theorem 3. *α-core guarantees that if participant P executes interaction I, then all of the entities participating in this interaction will execute it.*

Proof. In short, what we have to prove is that if participant P_i of interaction I executes it, then the rest of participants of this interaction will also execute it. If P_i executes interaction I, then it must have received a *START* message from the coordinator that manages I. If this happens, then the coordinator has executed either transition 4 or transition 7, because they are the only transitions in which this message may be sent out. Note that a *START* message is sent to each participant in set *locked* in both transitions.

- If we assume that transition 4 was executed, then the cardinality of the interaction has been reached ($n = Cardinality$), and the interaction has no shared participants ($shared = \emptyset$). In other words, this means that transition 2 has executed *Cardinality* times, and transition 1 has not executed. If transition 2 has executed *Cardinality* times, this means that set *locked* includes every participant of the interaction. Let P_j be another participant of this interaction. Then, P_j must be in set *locked* and, therefore, P_j is also sent a *START* message.
- If we assume that transition 7 was executed, note that another participant P_j may be: (i) the shared participant that sent the current *OK* message (denoted by *Sender*); (ii) another shared participant; or (iii) a non-shared participant.
  - If P_j is *Sender*, then P_j is included in set *locked* just before sending a *START* message to each participant in the set, so a *START* message is also sent to it.
  - If P_j is a shared participant other than *Sender*, then it was added to set *shared* in transition 1 when it sent the *OFFER* message to the coordinator. Since the interaction had at least one shared participant, transition 5 was executed and, after a number of transitions over state *LOCKING* (possibly none), transition 7 was finally executed. Note that each time transition 6 is executed, the participant that has just been locked is added to set *locked*, and a new participant is removed from set *waiting* and assigned to *current*. Thus, when transition 7 is executed, $waiting = \emptyset$ and *locked* contains the whole set of shared participants except for *current*, which is added immediately before the *START* messages are sent out. So, P_j is also sent a *START* message.
  - If P_j is a non-shared participant, then it was added to set *locked* in transition 2; therefore, it is also sent a *START* message. □

4.5. Idleness property

Theorem 4. *α-core guarantees that a participant can execute an interaction as long as it is not executing local computation.*

Proof. Although proving this property may be quite subtle in other algorithms, it is straightforward in our case. A participant can do its local computation whilst it is in state *ACTIVE*. On completing its local activities, it assigns the set of interactions in which it is interested to set *IS* and transits to state *WAITING* or *LOCKED*. A *START* message can only be received from a coordinator in which this participant was interested and this message must be received in state *LOCKED*. The theorem follows from the fact that a participant cannot simultaneously be executing local computations in state *ACTIVE* and interacting during transition 7, since we assume that entities are single threaded. □

5. COMPARISON

In this section we compare our solution with other authors' work. Some of the arguments we present during this discussion are corroborated with the results of the experimental study shown in Section 6.

The simplest algorithm to implement multiparty interactions can be found in [12], for instance, and it consists of using a central scheduler responsible for every interaction. Each participant sends it a

READY message when it arrives at a point where it needs to coordinate its activities with others, and the manager uses a counter per interaction to determine if they become enabled on receipt of a *READY* message. If more than one becomes enabled by the same *READY* message and they are conflicting, a random variable may be used to select one of them. Although this algorithm may be suitable for certain systems, the concerns of performance and reliability argue for a distributed solution.

In [57], a slightly modified version of the basic algorithm was presented. In this solution, there is one 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, and also in the context of multiparty cryptography [58], the central conflict resolutor is problematical in the general case.

The first distributed algorithms for coordination were produced in the context of CSP [1], but they were restricted to two-party interactions. Later, the problem gained great interest, and Chandy and Misra [21] developed two algorithms that are the basis of Bagrodia's MEM algorithm [20], which is currently one of the most cited in this field because it can be configured to achieve optimal performance in a particular system.

Bagrodia first devised a distributed version of the basic centralized algorithm called EM. It uses a number of interaction managers, each responsible for managing a subset of interactions. (Note that these subsets need not be disjoint.) When a participant 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 organizing 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 moment.

In α-core, there is one coordinator, i.e. one manager, per interaction and exclusion is achieved by locking participants in a given order. α-core reduces the possibility of an interaction never being executed because when a participant is in state *LOCKED* and it is unlocked by a coordinator, it selects the next prospective winner coordinator randomly, thus introducing some additional variability in the exclusion problem. In the experimental analysis, we show that EM produces the highest execution deviation and that α-core reduces it.

For these problems, Bagrodia devised a modified version of EM that was called MEM. It combines the synchronization 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 or onto the Drinking Philosophers problem [59]. Thus, conflicting managers are considered as philosophers that need to acquire shared forks placed between them in mutual exclusion.

MEM has an important drawback because the number of forks that 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. In α-core, there is no need for virtual resources. Instead, shared entities are considered to be resources that coordinators need to acquire. Thus, the probability of gaining mutual exclusion decreases as the number of shared participants increases, but,

in general, this is less problematical because most practical multiparty interactions are three-, four- or five-party[§], whereas the number of potentially conflicting interactions may be greater in typical scenarios.

A technique known as synchrony loosening was proposed in [12] to reduce the cardinality of an interaction at compile-time. In general, the speed at which a system may run should increase as the cardinality of its interactions decreases, but, unfortunately, this is not the case if we use MEM to implement multiparty interactions. The reason is that when synchrony loosening is applied, some interactions are split into a number of interactions with smaller cardinality that share many common participants. Thus, this technique increases the degree of potential conflict. As shown in our comparative study in Section 6.1, both EM and MEM are quite sensitive to increases in the degree of potential conflict, which may sharply affect the performance of a system. In contrast, α-core is completely insensitive to this parameter, which makes synchrony loosening the best choice to optimize a system.

A difficulty also arises in both EM and MEM because between enablement detection and acquisition of the token or the forks, a conflicting interaction may have started executing. The complex part of both algorithms is the way they use the information communicated during mutual exclusion to detect that an enablement is no longer current. This implies that each token or fork needs to carry an array with information about the entities, which makes messages larger as the number of entities increases and impossible to re-adapt at run-time if a new interaction or participant sprouts.

An additional problem with EM and MEM is that the technique they use for detecting enablement uses message-counters that need to be reset periodically so that they do not overflow. In EM, when a manager detects that the message counters are about to overflow, it becomes an initiator that sends extra messages to all of the entities in the system so that they reset their counters and inform the managers that coordinate the interactions in which they participate about this fact. The initiator cannot forward the token until every entity and manager has reset its counters. As for MEM, the procedure is similar but an additional interaction I_{R} needs to be added to the system to stop it and reset counters when they are about to overflow. I_{R} is conflicting with every other interaction so that when it executes, no other interaction may be executing at the same time. It remains to ensure that I_{R} will be executed within finite time from the instant a counter is about to overflow. For this purpose, MEM introduces a special entity for each interaction I, so that it participates in both I and I_{R}. (Only special entities can participate in I_{R}.) In general, a special entity is willing to participate in its two interactions; however, if the counter reset procedure needs to be executed, the special entities wait to execute only I_{R} so that it becomes the only enabled interaction in the system. Given that the set of interactions for which managers are responsible need not be disjoint, additional care must be taken so as to prevent different managers that coordinate I_{R} from starting the counter reset procedure simultaneously.

Clearly, both EM and MEM require the initiator manager to know every entity in the system, which is an important drawback. α-core does not require the set of participants to be known in advance nor does it require the set of coordinators to know each other or the whole set of entities, which is an important advantage.

[§]See [12], for instance. There it is shown that more than fifty applications of multiparty interactions, but none of them has a greater cardinality, except for the case of the leader election problem in which an interaction may coordinate an arbitrarily large number of participants that need to elect a leader.

The three algorithms examined so far can guarantee that an interaction cannot be enabled forever (cf. Theorem 2), i.e. they guarantee weak interaction fairness [12]. Unfortunately, they cannot guarantee that an interaction that is infinitely often enabled will be executed, i.e. they cannot guarantee strong interaction fairness. Given that an entity may autonomously decide when it is ready for interaction with others, and an entity ready for interaction cannot instantaneously be known by a manager, strong interaction fairness cannot be implemented by a deterministic algorithm unless arbitrarily large delays are introduced before deciding which interaction should be executed out of a set of conflicting interactions [60,61].

Recently, two new algorithms called TB and SM have been presented in [25]. They both are equivalent, the difference being that TB uses message-passing primitives, whereas SM uses shared-memory primitives. Furthermore, they have both been proven to schedule multiparty interactions in a strongly fair manner [62] with probability one, i.e. they are probabilistic algorithms. They improve on a previous result [24] in that the time an entity spends on local computation or the transmission delays need not be bounded by any predetermined constant, although they cannot deal with systems in which an entity monotonically increases the time it spends on local computation. Both algorithms are based on the idea of 'attempt, wait, and check' to establish an interaction. For instance, if participant P is interested in $I_1, I_2, \ldots, I_n$, it selects one of them randomly, and waits for some time. If after that P has detected that the other participants are also ready for that interaction, it may start immediately; otherwise, a new random attempt is made.

Although TB and SM can schedule interactions in a strongly fair manner, the cost of a random attempt–wait–check cycle may be considerably high. The expected time it takes for a participant in an enabled interaction I to start it is no greater than $m\eta/\prod_{p_i \in \mathcal{P}(I)} \psi_{p_i,I} + (m-1)\eta + \epsilon$, where m is its cardinality, $\psi_{p_i,I}$ is the probability that participant p_i chooses I in its random draw, $\mathcal{P}(I)$ is the set of participants interested in that interaction, and $\eta - \epsilon$, $\eta > \epsilon > 0$ is the length of a monitoring window. The time complexity increases sharply as m increases and makes it impractical for applications with four- or five-party interactions or applications in which participants offer more than three interactions at a time.

Finally, EM, MEM, TB or SM are not ready to deal with systems in which an entity may loop forever whilst executing local computation or finish. Since α-core requires coordinators to communicate with only the entities that are interested in them, it can deal with both terminating and non-terminating systems.

6. EXPERIMENTAL RESULTS

In this section, we report on the results of a simulation analysis that was undertaken using the Casale I simulation language [63] in order to compare EM, MEM and α-core. The experiments measured a number of metrics as a result of variations in the level of potential or run-time conflict and show that it performs comparably to other algorithms, but outperforms them in some situations that are clearly identified. We also present the results of an experimental study that was carried out using our CORBA-based implementation [64].

The results show that α-core is quite effective when compared with EM and MEM, and our implementation also achieves a good performance.

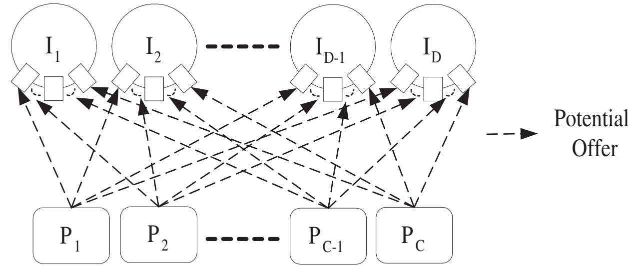


Figure 8. Synchronization pattern for simulations.

6.1. Comparative analysis

Figure 8 depicts the structure of the systems that we used in our experiments. Each was composed of C participants and D interactions. Note that they are all potentially conflicting because they are C-party and they share every participant in the system. When one or more interactions are potentially conflicting, we say that there is a *static conflict*. However, the existence of a static conflict amongst a set of interactions does not imply that they must be necessarily conflicting at run-time. They will not be conflicting at run-time unless their shared participants offer to participate in them. Once a shared participant offers to participate in more than one interaction, those interactions become conflicting at run-time. In Figure 8, each dotted arrow is a potential offer, and the level of actual participation is controlled by means of parameter P ($1 \leq P \leq D$). For instance, if $D = 10$ and $P = 2$, then only two interactions will be conflicting at a time, although they are all potentially conflicting with each other.

This pattern is quite flexible and it allows us to observe how the algorithms under consideration perform in different situations: in order to study how they are affected by the level of static conflict amongst interactions, we only need to vary D; in order to study how they are affected by the level of conflict at run-time, we only need to vary P; finally, in order to study how the cardinality of the interactions affect the algorithms, we only need to vary C.

We considered the system run on a point-to-point network with links connecting all coordinators with their participants and *vice versa*, which is adequate for modelling networks such as the Internet where several messages may be in transit simultaneously.

Several metrics were used in order to compare EM, MEM and α-core, as follows.

- Elapsed time, which is the total time taken to complete an experiment.
- Selection time, which is the time taken by an algorithm to select an interaction for execution, possibly out of many conflicting interactions. It is measured from the instant that an interaction becomes enabled (as viewed by its corresponding coordinator) to the time that it is selected for execution by a specific algorithm.
- Interaction time, which is the time elapsed from the instant a participant finishes local computation to the instant it executes one of the interactions it offers.
- Message count, which is the average number of messages needed to schedule one interaction for execution.

- Execution deviation, which is the mean deviation from the expected number of times each interaction should have been executed in an environment in which every interaction would have had the same probability of being selected to be executed.

These metrics were studied as a function of a number of parameters, including the level of participation, the cardinality of each interaction, the degree of potential conflict, and the average time taken to transmit a message between processes.

We varied C and D in the range $[2\ldots10]$, P in the range $[1\ldots10]$, and each experiment was run 100 times for a duration required to schedule 10 interactions using different random seeds. Each communication link was modelled as a first-come first served (FCFS) server with service time (i.e. transmission time) sampled from an exponential distribution $\exp(\mu)$ with $\mu = 2$ ms. We considered that the time participants are performing local computation or interacting is not negligible, and we sampled them from two exponential distributions with mean 20 ms and 2 ms, respectively.

We found out that other things being equal, the metrics vary almost linearly with μ, except for the message count, which is obviously independent from this parameter. This behaviour was expected because the communication medium was modelled by a FCFS server with mean service sampled from an exponential distribution with small scale. It is also worth mentioning that modelling communication links as FCFS servers with service time sampled from another distribution varied the timings of each algorithm, although the relative ranking remained essentially unchanged.

Furthermore, for the purpose of comparison, each manager in the EM and MEM algorithms managed only one interaction. Anyhow, none of these assumptions are an inherent feature of the simulation, and alternative hypothesis may easily be simulated.

6.2. The results of the simulation

Figures 9–12 summarize the average percentage increase or decrease of each metric when each parameter was increased by one unit. For instance, in Figure 9, if we increase D from 7 to 8, the elapsed time needed to complete the simulation is expected to increase 0.00% in the case of α-core, 1.19% in the case of EM and 1.42% in the case of MEM.

In Figure 9, we can see that both EM and MEM are quite sensitive to the degree of potential conflict D, with MEM having a larger increase. The reason for this is that an increase in D increases the number of managers that have a shared fork. Thus, an increase in D increases the probability that a hungry manager will have to request forks from its neighbours, i.e. the managers that are potentially conflicting with it. If a given manager has $D-1$ neighbours, we can assume that the probability that a manager owns k forks is uniform for all $0 \leq k \leq D-1$; thus, the probability of the manager owning all $D-1$ forks will be $1/D$, which decreases as D increases. It follows that a manager has to request forks more often as D increases, thus increasing the selection time and the number of messages exchanged to achieve mutual exclusion. For the EM algorithm, increasing D amounts to increasing the distance the token needs to travel before finding an enabled manager, thus increasing both the selection time and the number of messages needed to pass the token until it arrives at an enabled manager. α-core is completely insensitive to D because coordinators are not directly dependent on each other to achieve mutual exclusion.

Note that the impact on the elapsed time is not as large as might be expected. The reason is that this metric is hundreds of times greater than the selection time, thus reducing the impact on the

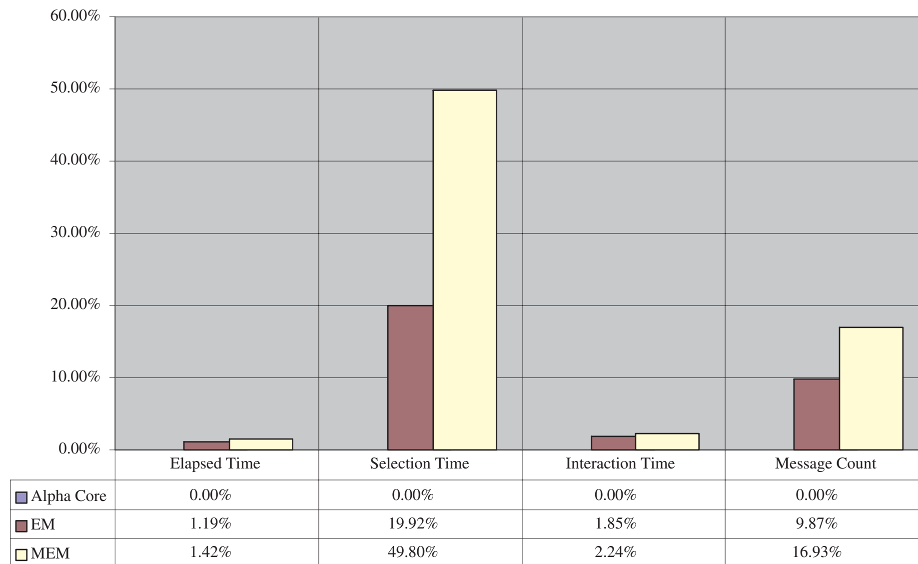


Figure 9. Results of the simulation analysis: sensitivity to D.

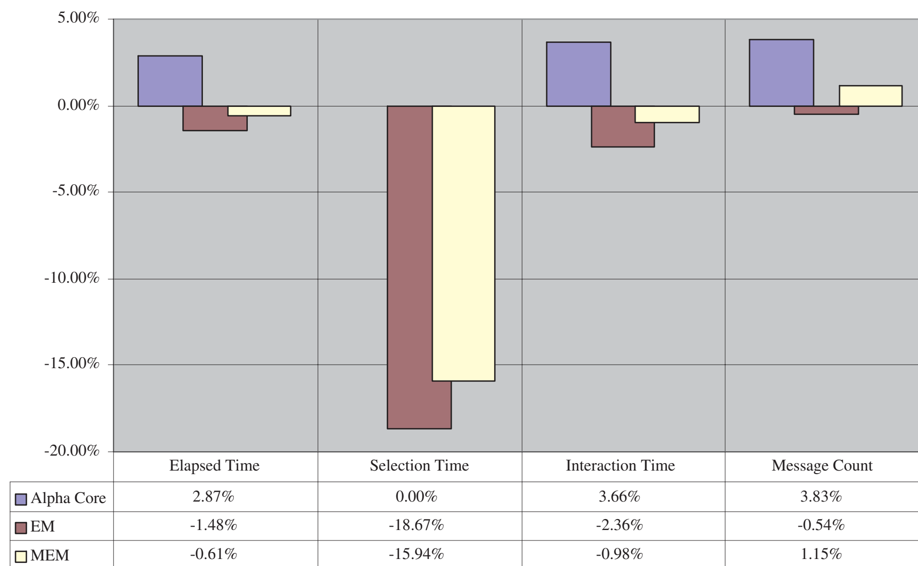
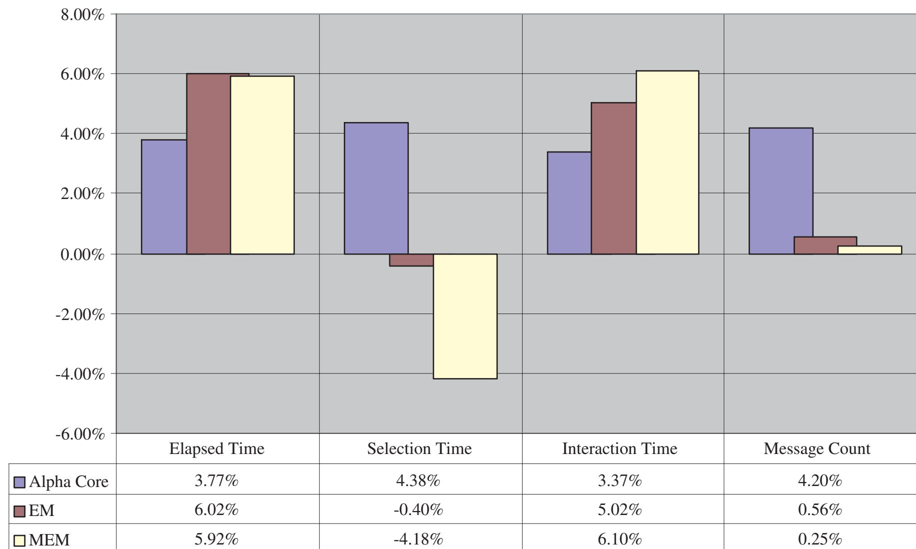
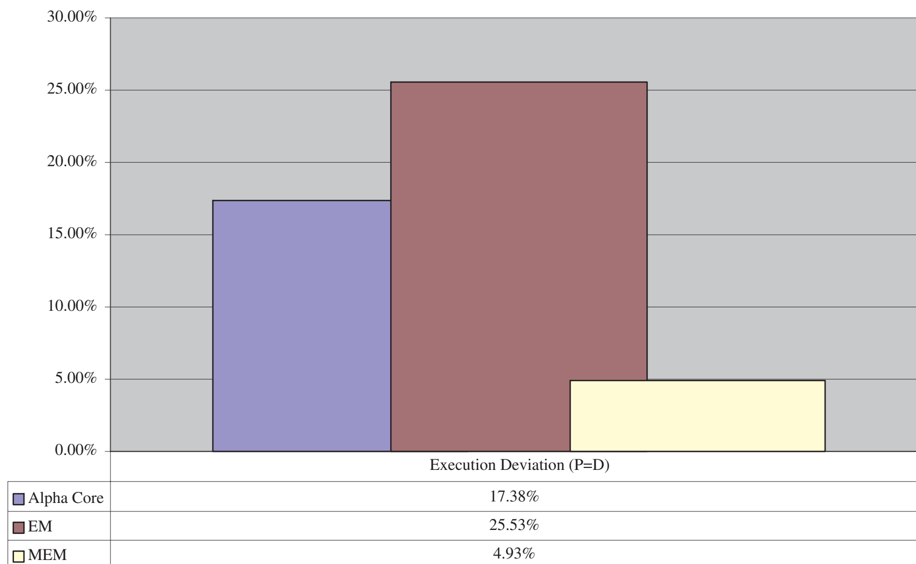


Figure 10. Results of the simulation analysis: sensitivity to P.



	Elapsed Time	Selection Time	Interaction Time	Message Count
Alpha Core	3.77%	4.38%	3.37%	4.20%
EM	6.02%	-0.40%	5.02%	0.56%
MEM	5.92%	-4.18%	6.10%	0.25%

Figure 11. Results of the simulation analysis: sensitivity to C.



	Execution Deviation (P=D)
Alpha Core	17.38%
EM	25.53%
MEM	4.93%

Figure 12. Results of the simulation analysis: execution deviation.

average percentage. For instance, if $C = 6$ and $P = 5$, the selection time for MEM increased 4.40 ms when we increased D from 9 to 10, i.e. it increased by 31.18%. However, the total elapsed time increased from 5277 to 5470 ms, which amounts to only 2.65%.

As for parameter P, it is interesting to observe in Figure 10 that the selection time of α-core is also insensitive to it, but EM improves its selection time as the degree of run-time conflict increases. The reason is that the selection time for EM is quite sensitive to the average distance between enabled managers. If this distance decreases, the selection time improves. Given that the number of managers is fixed, this distance decreases as P increases because the number of enabled managers increases as the degree of run-time conflict increases. The selection time of MEM also decreases with P because the more managers are enabled, the greater the probability of a manager having all its shared forks being enabled. In fact, the selection time for both EM and MEM is at a minimum when the degree of run-time conflict is maximum. The time α-core takes to select an enabled interaction is completely insensitive to P, which is clearly desirable, although the number of messages needed to achieve mutual exclusion increases due to the extra messages needed to refuse coordinators that have not succeeded in achieving mutual exclusion.

As for parameter C, we can see in Figure 11 that α-core is sensitive to it because it achieves mutual exclusion by locking shared participants. Thus, the selection time increases steadily as C increases. In contrast, both EM and MEM are almost insensitive to C. With respect to the number of messages needed to schedule an interaction, α-core needs more messages to achieve mutual exclusion when C increases, which justifies the rise shown in the plot.

Finally, we examine the execution deviation for each algorithm in Figure 12. This metric can easily be measured when $D = P$ because all interactions are conflicting permanently in these cases. Thus, during a run long enough to schedule N interactions, each one should be executed approximately N/D times. It is interesting to observe that α-core ranks between EM and MEM. This was predictable because, in the case of EM, the chances of an interaction being executed depend solely on its chance of receiving the token at the right time. As for α-core, it depends on the chances of a coordinator being able to lock all of its participants in order. In contrast, the algorithm for implementing the Diner Philosophers problem on which MEM relies guarantees that every philosopher that becomes hungry will eventually have a chance to eat, thus producing a better distribution of executed interactions in this set of experiments.

6.3. The performance of our prototype

We have implemented α-core in the laboratory using Java 1.2 and the CORBA implementation Orbacus 3.3.2. In this section, we present a performance analysis carried out using this prototype. The tests were run on an isolated network composed of 200 MHz Pentium computers with 64 MB RAM. Participants and coordinators were distributed on the available machines so that each one hosted the same number of processes.

We used multiparty implementations of several well-known coordination problems as benchmarks, and we measured the average number of interactions per second of every experiment. The results we obtained ranged from 100 to 180 interactions per second, which we think shows that our prototype performs quite well, and that α-core is suitable for use in practical applications. Table IV summarizes the benchmarks we used and Figure 13 shows the results.

Table IV. Description of our benchmarks.

Name	Description
The Dining Philosophers problem	This is the paradigm of those situations in which an entity needs to get exclusive access to several entities simultaneously. The version that we have run was presented in Section 2.3.
The Leader Election problem	This is another well-known problem in the field of distributed systems. In the field of e-commerce, it may be viewed as a basic synchronization pattern in round-based auctions on the Internet. The number of participants (auctioneers) is given in parenthesis.
The Bank Transfer problem	This is a typical problem in an e-commerce setting, since the goal is to coordinate a point-of-sales terminal and two banks to transfer money from one of them to the other. The problem we implemented was described in detail in [34], and is very similar to that described in [11].
The Matrix Multiplication problem	This is a typical example of a task that needs to be divided into several subtasks on which several workers work loosely coupled until the results need to be aggregated. This is a typical pattern in systems that need to search for goods or services using several providers [35].
The Traveling Salesman problem	This is a classical optimization problem in which the goal is to find the cheapest way of visiting a set cities and returning to the starting point [65]. This problem is of utmost importance when planning the distribution of goods in e-commerce problems.
The Towers of Hannoi problem	Although this pattern is not likely to arise in practice, it is a good example of a problem in which a relatively small number of entities need to interact very frequently using complex coordination patterns. We used the implementation presented in [12] to test our prototype in an extreme situation.

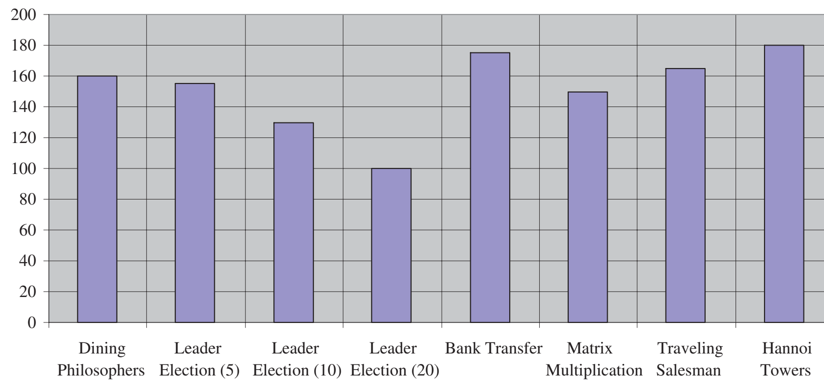


Figure 13. Experimental results using the benchmarks described in Table I.

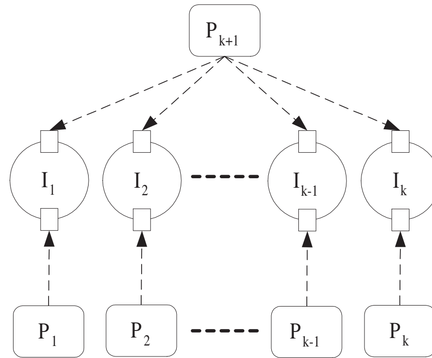


Figure 14. Synchronization pattern for experiments.

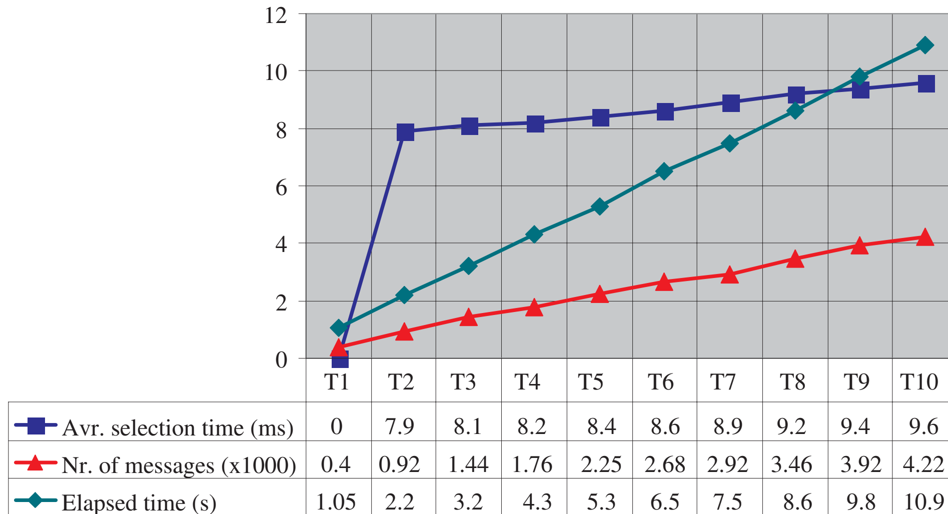
Java is a portable, flexible language, and this is the reason why we have selected it to implement our prototype. Nonetheless, its performance is a clear drawback, chiefly if we take into account that communication delays are comparable to the time the Java Virtual Machine takes to execute most instructions.

In order to study how the implementation language affects the experimental results, we carried out a series of tests using the synchronization pattern sketched in Figure 14. We prepared a set of programmes $T_1, T_2, \ldots, T_n$, and each T_k consisted of k biparty interactions, referred to as I_i ($i = 1, 2, \ldots, k$), and $k + 1$ participants, referred to as P_i ($i = 1, 2, \ldots, k + 1$). $P_1, P_2, \ldots, P_k$ offer participation in interactions $I_1, I_2, \ldots, I_k$, respectively, and P_{k+1} offers participation in any interaction, thus making them conflicting. Experiment T_1 is trivial because it consists of two entities and a biparty interaction, but it shows how our algorithm performs in the absence of conflicts.

Each test was run until each interaction was executed 100 times, and we measured the average selection time, the number of messages exchanged during the experiments, and the elapsed running time.

Figure 15 shows the results of our tests. Theoretically, the average selection time is independent from the number of conflicting interactions in our scenario, because when coordinator I_i becomes enabled in test T_k (participants P_i and P_{k+1} have offered to participate in I_i), it sends a *LOCK* message to its shared participant P_{k+1}. The first *LOCK* message received by P_{k+1} is replied with an *OK* message, so the period of time from when I_i detects that it is enabled until the instant it knows it is the winner should be about 2μ, μ being the average time taken to transmit a message. In our system, $\mu \simeq 2$ ms. If the *LOCK* message sent by coordinator I_i is not the first message P_{k+1} receives, this coordinator will eventually get a *REFUSE* message from P_{k+1} as a reply. Since every time P_{k+1} participates in one interaction it must refuse $k - 1$ coordinators, the average refuse time is more variant, because the first coordinator that is refused has to wait less time than the coordinators that are refused after it. The reason for this is that our implementation was written in Java, and the loop we used to send $k - 1$ *REFUSE* messages takes a time that is comparable to the time a message takes to get to its destination.

When k coordinators are competing to lock the shared participant P_{k+1}, each has a success ratio of $1/k$. In other words, the bottleneck of our system is participant P_{k+1}. Since the number of interactions



	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
Avr. selection time (ms)	0	7.9	8.1	8.2	8.4	8.6	8.9	9.2	9.4	9.6
Nr. of messages (x1000)	0.4	0.92	1.44	1.76	2.25	2.68	2.92	3.46	3.92	4.22
Elapsed time (s)	1.05	2.2	3.2	4.3	5.3	6.5	7.5	8.6	9.8	10.9

Figure 15. Experimental results using the synchronization pattern sketched in Figure 14.

it can run per unit of time depends on the underlying computer and is constant, it will participate less times per unit time in each one as the number of offered interactions increases. Theoretically, the average number of times it executes an interaction per unit of time should be $1/k$ times as often as it would execute it in the absence of conflicts. This means that the time taken by a coordinator to execute a number of interactions sharing a participant with $k-1$ coordinators should be nearly k times the time taken to execute the same number of interactions without conflicts.

From Figure 15, we can appreciate an important difference between the first column and the other columns. Test T_1 is a particular case, since coordinator I_1 does not have to lock any participants, so its selection time is zero. To execute an interaction, four messages need to be sent (two *PARTICIPATE* and two *START*), so 400 messages are transmitted. Although this should only require about 800 ms, the coordinator elapses 1.05 s due to computational overhead.

Test T_1 is the best case. When there are conflicts, coordinators must lock their shared participant P_{k+1}. Then, six messages are needed to execute an interaction (one *PARTICIPATE*, one *OFFER*, one *LOCK*, one *OK* and two *START*); but every time a coordinator is refused, a penalty of four messages is added (one *REFUSE*, one *UNLOCK*, another *OFFER* and another *LOCK*). Since the selection time is measured from the time that an interaction becomes enabled (after the last offer) until the time the *OK* message is received, the selection time should be $2\mu \simeq 4$ ms (the transmission time of one *LOCK* and one *OK*). Furthermore, the time each coordinator takes to execute 100 interactions should be the time needed to transmit 600 messages plus a penalty every time that it is refused. The number of times that a coordinator is refused should be proportional to the number of coordinators.

The measures we obtained are slightly greater than theoretically expected, due to computation overhead. With two coordinators, the selection time is about 8 ms (four more than expected) and the time taken to run 100 interactions is also a bit larger than expected (double the time elapsed by

only one coordinator). This difference can be justified if we take the number of messages transmitted into account. Recall that 600 messages are needed to execute 100 interactions, so 320 extra messages have been added when a coordinator has been refused. Furthermore, the time needed to transmit 920 messages is 1.840 ms, so a penalty of 360 ms appears due to computational overhead.

As the number of conflicting coordinators increases, the selection time increases slightly due to computational overhead. The time taken to execute 100 interactions increases almost proportionally to the number of coordinators. We think that these results are satisfactory because the system behaves as we expected it, and the selection time increases at a ratio that is about 2.47%. Those results are very good if we take into account the limitations on execution speed of our prototype, where computation times are relevant with respect to transmission times.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented an algorithm for implementing multiparty synchronization. A variety of solutions exist in the literature, and our approach is innovative because we do not require the set of entities in a system to be known beforehand. Furthermore, coordinators do not need to know all of the entities in a system, and entities are not directly dependent on each other, which is an important drawback in other proposals. In addition, our algorithm is not sensitive to the degree of potential or actual conflict, but to the cardinality of the interactions under consideration, which seems desirable, and allows us to apply synchrony loosening [12]. To the best of our knowledge, our approach is the first algorithm to which this technique can be applied without a negative impact on its efficiency.

We have also presented the results of an experimental study that shows that our solution achieves a good performance. From these results, we can conclude that α-core performs well enough to be used in applications in which the multiparty interaction model helps the programmer to design an adequate solution to their problem [19]. In fact, we are using our implementation of α-core as the basis of the run-time support needed to animate the aspect-oriented language $\mathcal{CAL}$ [7,29,66], which aims at increasing the level of abstraction of a programme by considering the concurrent behaviour of components as an aspect where multiparty interactions are the sole means for synchronization and communication.

Currently, we are working to improve our algorithm so that participants do not need to re-send their offers when they are rejected by a coordinator and no other coordinator has tried to lock them. We are also working on reducing the execution deviation of our algorithm [67,68].

REFERENCES

1. Hoare CAR. *Communicating Sequential Processes*. Prentice-Hall: Englewood Cliffs, NJ, 1985.
2. Barnes J. *Programming in Ada'95*. Addison-Wesley: Reading, MA, 1995.
3. Andrews GE, Olsson RA. *The SR Programming Language*. Benjamin-Cummings: Redwood City, CA, 1993.
4. Hartley SJ. *Operating Systems Programming: The SR Programming Language*. Oxford University Press: Oxford, 1997.
5. Hilderink GH. Communicating Java threads reference manual. *Proceedings of the 20th World Occam and Transputer User Group Technical Meeting, WoTUG'20* (*Concurrent Systems Engineering*, vol. 50), Bakkers A (ed.). World Occam and Transputer User Group (WoTUG), IOS Press: Amsterdam, 1997; 283–325.
6. Keen A, Ge T, Maris J, Olsson R. JR: Flexible distributed programming in an extended Java. *Proceedings 21st International Conference on Distributed Computing Systems, ICDCS'01*. IEEE Computer Society Press: Los Alamitos, CA, 2001; 575–584.
7. Corchuelo R, Pérez JA, Ruiz-Cortés A. Aspect-oriented interaction in multi-organizational Web-based systems. *Computer Networks* 2003; **41**(4):385–406.
8. D'Souza DF, Wills AC. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley: Reading, MA, 1999.

9. Ehrich H-D, Caleiro C. Specifying communication in distributed information systems. *Acta Informatica* 2000; **36**:591–616.
10. Evangelist M, Francez N, Katz S. Multiparty interactions for interprocess communication and synchronization. *IEEE Transactions on Software Engineering* 1989; **15**(11):1417–1426.
11. Felber P, Reiter MK. Advanced concurrency control in Java. *Concurrency and Computation: Practice and Experience* 2002; **14**(4):261–285.
12. Francez N, Forman I. *Interacting Processes: A Multiparty Approach to Coordinated Distributed Programming*. Addison-Wesley: Reading, MA, 1996.
13. Joung Y-J, Smolka SA. A comprehensive study of the complexity of multiparty interaction. *Journal of the ACM* 1996; **43**(1):75–115.
14. Katz S, Forman IR, Evangelist WM. Language constructs for distributed systems. *IFIP TC2 Working Conference on Programming Concepts and Methods*, Galilea, Israel, April 1990.
15. Lea D. *Concurrent Programming Using Java: Design Principles and Pattern* (2nd edn). Addison-Wesley: Reading, MA, 1999.
16. Odell J, Van Dyke H, Bauer B. Extending UML for agents. *Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National Conference on Artificial Intelligence*, Wagner G, Lesperance Y, Yu E (eds.). 2000; 3–17.
17. Pérez JA, Corchuelo R, Ruiz D, Toro M. An enablement detection algorithm for open multiparty interactions. *Proceedings of the 2002 ACM Symposium on Applied Computing SAC'02*, Madrid, Spain, March 2002. ACM Press: New York, 2002; 378–384.
18. Rumbaugh J, Jacobson I, Booch G. *The Unified Modeling Language Reference Manual* (*Object Technology Series*, vol. 1). Addison-Wesley/Longman: Reading, MA, 1999.
19. Tang P, Muraoka Y. Parallel programming with interacting processes. *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing, LCPC'99* (*Lecture Notes in Computer Science*, vol. 1863), Carter L, Ferrante J (eds.). Springer: Berlin, 2000; 201–218.
20. Bagrodia RL. Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Transactions on Software Engineering* 1989; **15**(9):1053–1065.
21. Chandy KM, Misra J. *Parallel Program Design: A Foundation*. Addison-Wesley: Reading, MA, 1988.
22. Corchuelo R. Prototyping constraint-based specifications of distributed systems. *PhD Thesis*, Facultad de Informática y Estadística, Dpto. de Lenguajes y Sistemas Informáticos, University of Sevilla, 1999.
23. Joung Y-J, Smolka SA. Coordinating first-order multiparty interactions. *ACM Transactions on Programming Languages and Systems* 1994; **16**(3):954–985.
24. Joung Y-J, Smolka SA. Strong interaction fairness via randomization. *IEEE Transactions on Parallel and Distributed Systems* 1998; **9**(2):137–149.
25. Joung Y-J. Two decentralized algorithms for strong interaction fairness for systems with unbounded speed variability. *Theoretical Computer Science* 2000; **243**(1–2):307–338.
26. Kumar D. An implementation of N-party synchronization using tokens. *Proceedings 10th International Conference on Distributed Computing Systems*. IEEE Computer Society Press: Los Alamitos, CA, 1990; 320–327.
27. Lynch NA. Fast allocation of nearby resources in a distributed system. *Proceedings of the 12th ACM Symposium on Theory of Computing*. ACM Press: New York, 1980; 70–81.
28. Zorzo AF, Stroud RJ. A distributed object-oriented framework for dependable multiparty interactions. *ACM SIGPLAN Notices* 1999; **34**(10):435–446.
29. Corchuelo R, Pérez JA, Toro M. A multiparty coordination aspect language. *ACM SIGPLAN Notices* 2000; **35**(12):24–32.
30. Francez N, Hailpern BT, Taubenfeld G. Script: A communication abstraction mechanism and its verification. *Science of Computer Programming* 1986; **6**(1):35–88.
31. Joung Y-J. On the design and implementation of multiparty interactions. *PhD Thesis*, State University of New York at Stony Brook, New York, 1992.
32. Box D, Ehnebuske D, Kakivaya G, Layman A, Mendelsohn N, Nielsen HF, Thatte S, Winer D. Simple object access protocol (SOAP 1.1). *Technical Report*, W3C Consortium. http://www.w3.org/TR/SOAP [7 June 2004].
33. Englander R. *Java and SOAP*. O'Reilly & Associates: London, 2002.
34. Ruiz-Cortés A, Corchuelo R, Pérez JA, Durán A, 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; 56–65.
35. Fayad M. E–Frame: A process-based, object-oriented framework for e-commerce. *Proceedings of the International Conference on Internet Computing, IC'2001*, vol. 1. CSREA Press: Las Vegas, NV, 2001; 124–128.
36. Booch G. *Object-Oriented Design with Applications*. Benjamin-Cummings: Redwood City, CA, 1990.
37. de Champeaux D. Object-oriented analysis and top-down software development. *Proceedings of the European Conference on Object-Oriented Programming, ECOOP'91* (*Lecture Notes in Computer Science*, vol. 512). Springer: Berlin, 1991; 360–375.

38. Coad P, Yourdon E. *Object-Oriented Analysis (Computing Series)*. Yourdon Press: Englewood Cliffs, NJ, 1990.
39. Rumbaugh J, Blaha M, Premerlani W, Eddy F, Lorensen W. *Object-Oriented Modeling and Design*. Prentice-Hall: Schenectady, NY, 1991.
40. Wirfs-Brock R, Wilkerson B. *Designing Object-Oriented Software*. Prentice-Hall: Englewood Cliffs, NJ, 1990.
41. Reenskaug T, Wold P, Lehne OA. *Working With Objects. The OOram Software Engineering Method*. Manning: Greenwich, CT, 1995.
42. Andersen E. Conceptual modeling of objects: A role modeling approach. *PhD Thesis*, University of Oslo, 1997.
43. Bauer B, Müller J, Odell J. Agent UML: A formalism for specifying multiagent interaction. *Proceedings of the 22nd International Conference on Software Engineering ICSE'01* (*Lecture Notes in Computer Science*, vol. 1957), Ciancarini P, Wooldridge M (eds.). Springer: Berlin, 2001; 91–103.
44. Bauer B, Müller J, Odell J. Agent UML: A formalism for specifying multiagent software systems. *International Journal of Software Engineering and Knowledge Engineering* 2001; **11**(3):207–230.
45. Caire G, Leal F, Chainho P, Evans R, Garijo F, Gómez J, Pavón J, Kearney P, Stark J, Massonet P. Agent oriented analysis using MESSAGE/UML. *Proceedings of Agent-Oriented Software Engineering, AOSE'01*, Montréal, 2001; 101–108.
46. Odell J, Parunak HVD, Bauer B. Representing agent interaction protocols in UML. *Proceedings of the 22nd International Conference on Software Engineering ISCE'01* (*Lecture Notes in Computer Science*, vol. 1957), Ciancarini P, Wooldridge M (eds.). Springer: Berlin, 2001; 121–140.
47. Lea D, Bowbeer J, Goetz B, Holmes D, McCorvey C, Peierls T. JSR 166: Concurrency Utilities. http://www.jcp.org/en/jsr/detail?id=166 [7 June 2004].
48. Charlesworth A. The multiway rendezvous. *ACM Transactions on Programming Languages and Systems* 1987; **9**(2): 350–366.
49. Joung Y-J, Smolka SA. A comprehensive study of the complexity of multiparty interaction. *Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages POPL'92*. ACM Press: New York, 1992; 142–153.
50. Dijkstra EW. Hierarchical ordering of sequential processes. *Operating Systems Techniques*, Hoare CAR, Perrot RH (eds.). Academic Press: New York, 1972; 72–93.
51. Lynch NA, Merritt M, Weihl WE, Fekete A. *Atomic Transactions*. Morgan Kaufmann: San Mateo, CA, 1994.
52. Pérez JA, Corchuelo R, Ruiz D, Toro M. An order-based, distributed algorithm for implementing multiparty interactions. *Proceedings of the 5th International Conference on Coordination Models and Languages, COORDINATION'02* (*Lecture Notes in Computer Science*, vol. 2315), Arbab F, Talcott CL (eds.). Springer: Berlin, 2002; 250–257.
53. Coffman EG, Elphick MJ, Shoshani A. System deadlocks. *Computing Surveys* 1971; **3**(2):67–78.
54. Rogerson D. *Inside COM*. Microsoft Press: New York, 1997.
55. Berners-Lee T, Fielding R, Masinter L. *RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax*, August 1998.
56. Gray J, Reuter A. *Transaction Processing*. Morgan Kaufmann: San Mateo, CA, 1993.
57. Corchuelo R, Ruiz D, Toro M, Prieto JM, Arjona JL. A distributed solution to multiparty interaction. *Recent Advances in Signal Processing and Communications*. World Scientific: Singapore, 1999; 318–323.
58. Canetti R. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology* 2000; **13**(1):143–202.
59. Chandy KM, Misra J. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems* 1984; **6**(4):632–646.
60. Joung Y-J. Characterizing fairness implementability for multiparty interaction. *Proceedings of the 23rd International Colloquium on Automata, Languages and Programming*, Paderborn, Germany, 1996 (*Lecture Notes in Computer Science*, vol. 1099), Meyer F, Monien B (eds.). Springer: Berlin, 1996; 110–121.
61. Tsay YK, Bagrodia RL. Some impossibility results in interprocess synchronization. *Distributed Computing* 1993; **6**(4):221–231.
62. Ruiz D, Corchuelo R, Toro M. Fairness in systems based on multiparty interactions. *Concurrency and Computation: Practice and Experience* 2003; **15**(11–12):1093–1116.
63. Corchuelo R, Pérez JA. Casale I: A new object–based language for discrete simulation. *Proceedings of the 5th European Concurrent Engineering Conference ECEC'98*, Erlagen–Nuremberg, Germany, 1998. The Society for Computer Simulation: San Diego, CA, 1998; 310–312.
64. Pérez JA, Corchuelo R, Ruiz D, Toro M. A framework for aspect-oriented multiparty coordination. *New Developments in Distributed Applications and Interoperable Systems*. Kluwer: Dordrecht, 2001; 161–174.
65. Lawler EL, Lenstra JK, Rinooy Kan AHG, Shmoys DB. *The Traveling Salesman Problem*. Wiley: New York, 1985.
66. Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier J, Irwin J. Aspect-oriented programming. *Proceedings of the European Conference on Object-Oriented Programming ECOOP'97* (*Lecture Notes in Computer Science*). Springer: Berlin, 1997; 220–242.
67. Ruiz D, Corchuelo R., Pérez JA, Toro M. An algorithm for ensuring fairness and liveness in non-deterministic systems based on multiparty interactions. *Proceedings of the 8th International Euro-Par Conference, EUROPAR'02*, Paderborn, Germany, August 2002 (*Lecture Notes in Computer Science*, vol. 1845). 2002; 563–572.
68. Ruiz D, Corchuelo R, Toro M. Fairness in systems based on multiparty interactions. *Concurrency and Computation: Practice and Experience* 2003; **15**(9):1093–1116.