Distributed Implementation of Systems with Multiparty Interactions and Priorities

Imene Ben-Hafaiedh[1], Susanne Graf[1], and Nejla Mazouz[2]

[1] VERIMAG. 2, avenue de Vignate
38610 Gieres, France
[2] Tunisia Polytechnic School
{benhfaie,graf,mazouz}@imag.fr

Abstract. Rich interaction models are a powerful mechanism allowing to synchronize several entities in order to achieve some common goal and to specify global properties in an abstract manner. In this paper we focus on two types of interaction models, namely *multiparty* interactions and *priorities* where priorities may be used to specify different scheduling policies. We propose a protocol for building distributed implementation of component-based models with multiparty interactions and priorities. We also present a set of experiments providing a performance analysis of the protocol.

Keywords: priorities, multiparty interaction, distributed systems.

1 Introduction

Providing a distributed implementation of component-based systems while preserving global properties is a very challenging task [5,18], as we cannot determine exactly the global state of distributed systems, but we can only approximate it [11]. Interaction models in component-based systems are a means for abstracting global properties of these systems. In this paper we focus on two types of interaction models, namely *multiparty interactions* and *priorities.*

Multiparty interactions provide a convenient means for describing the global behavior of a distributed system. Thus they can be later refined into efficient low-level protocols with respect to the platform in use. A multiparty interaction consists of a set of actions that need to be executed jointly by a number a components.

Priorities between interactions in component-based systems are widely used in system design as a way of defining different scheduling policies. They are expressive enough to enforce safety properties without inducing any deadlock in the system [13]. In fact, enforcing priorities means that when two interactions can be fired simultaneously, the one with higher priority must be executed. Thus, they restrict the behavior of the initial system which means that they preserve deadlock freedom if the initial system is deadlock-free.

The main challenge in enforcing priorities in a distributed setting is that components need to obtain a common and precise knowledge about the enabledness of interactions so the interaction with higher priority can be executed.

In [5], a partially distributed implementation has been proposed for component-based systems with priorities, but where a centralized engine manages interactions and enforce priorities. Existing distributed protocols implementing multiparty interactions [1,17] do not handle priorities. In [6], we have proposed a protocol for distributed implementation of systems with *binary* interactions and priorities. In a binary interaction exactly two components are involved. Which means that it is sufficient that each of the involved components gets information about the other to decide the execution of an interaction. Thus, the protocol presented in [6], is completely symmetric, which means that it does not distinguish between the two participants of an interaction. In the case of a multiparty interactions a completely symmetric protocol may, in the case where all partners of an n-ary interaction initiate the protocol almost simultaneously, lead to a huge number of messages. For this reason, most existing solutions for multi-party interactions (e.g. [17]) are asymmetric and predefine an "initiator" for each interaction; here we also choose this asymmetric approach.

In this paper, we propose a protocol providing a distributed implementation of component-based systems with multiparty interactions and priorities.

The paper is organized as follows. In Section 2, we present the basic semantics of a distributed system with multiparty interactions and priorities. Section 3 is dedicated to a thorough description of the protocol, and we prove its correctness in Section 4. We discuss in Section 5 some experimental results. In Section 6, we compare our proposal with other existing approaches.

2 Distributed System with Priorities

In this section, we present our notion of *distributed systems* defined by a set of *components.*

Definition 1 (component). *A* component K *is a* labeled transition system *(LTS)* $(Q, q^0, \mathcal{I}_K, \delta)$*:* Q *is a set of states with initial state* $q^0 \in Q$*,* $\mathcal{I}_K$ *is the label set.* $\delta \subseteq Q \times \mathcal{I} \times Q$ *is a transition relation.* $(q, a, q') \in \delta$ *is denoted* $q \xrightarrow{a} q'$ *and we denote by* $q \xrightarrow{a}$ *the fact that* $\exists\ q' \in Q$ *such that* $q \xrightarrow{a} q'$*.*

Definition 2 (distributed system). *A distributed system* $DS^{\mathcal{K}}$ *is defined by a set of components* $\mathcal{K} = \{K_i\}_{i=1}^n$*. It defines an LTS* $(\mathcal{S}, \mathcal{I}, \Delta)$ *where:*

- $\mathcal{I}$ *is a set of* multiparty interactions*. An interaction* a *may belong to more than one component and* $\mathcal{K}_a = \{K_i \in \mathcal{K} | a \in \mathcal{I}_{K_i}\}$*.*
- $\mathcal{S}$ *is the set of* global states *where* $S_0 \in \mathcal{S}$ *is the initial state. If* $S \in \mathcal{S}$*, then* $S = (q_1, \ldots, q_n)$ *where* $q_i \in Q_i$*. A* local state $S_{K_i} = q_i$ *is the restriction of the global state* S *to the states of* K_i*.*
- $\Delta \subseteq \mathcal{S} \times \mathcal{I} \times \mathcal{S}$ *is a transition relation.*

If a transition $\tau \in \Delta$, $\tau = (S, a, S')$ is fired, which means that the interaction a is executed, then the new global state S' is reached and we denote this by $S \xrightarrow{a} S'$. Δ is the least set of transitions satisfying the following rule:

$$\frac{a \in \mathcal{I} \qquad \forall i \text{ s.t. } K_i \in \mathcal{K}_a,\ q_i \xrightarrow{a} q_i' \qquad \forall i \text{ s.t. } K_i \notin \mathcal{K}_a,\ q_i = q_i'}{(q_1, \cdots, q_n) \xrightarrow{a} (q_1', \cdots, q_n')}$$

Intuitively this rule means that a distributed system DS can execute an interaction in a global state S, if all components involved in this interaction can execute it in their local state corresponding to S. This means that firing a corresponds to synchronously firing the corresponding interactions a of the components in $\mathcal{K}_a$.

Priorities. Given a set of interactions $\mathcal{I}$, a priority between two interactions specifies which one is preferred over the other when both can be executed. Priorities are defined as partial orders $<\subseteq \mathcal{I} \times \mathcal{I}$ and we write $a < b$ means that a has less priority than b.

Definition 3 (distributed system with priorities). *A distributed system $DS^{\mathcal{K}} = (\mathcal{S}, \mathcal{I}, \Delta)$ with a priority order $<\subseteq \mathcal{I} \times \mathcal{I}$ is a distributed transition system $DS^{\mathcal{K}}_{<} = (\mathcal{S}, \mathcal{I}, \Delta_<)$ where: $\Delta_< \subseteq \mathcal{S} \times \mathcal{I} \times \mathcal{S}$ is the largest transition relation satisfying the following rule:*

$$\frac{S \xrightarrow{a} S',\ \nexists b \in \mathcal{I}\ s.t.\ a < b\ and\ S \xrightarrow{b}}{S \xrightarrow{a}_< S'}$$

Definition 4 (locally ready, globally ready, enabled interaction). *Let $DS^{\mathcal{K}}_{<}$ be a distributed system with priorities as above. Consider a global state $S = (q_1, \ldots, q_n) \in \mathcal{S}$, and an interaction $a \in \mathcal{I}$.*

- *a is* locally ready *in the* local state *q_i iff $q_i \xrightarrow{a}_i$*
- *a is* globally ready *in S iff $\forall i$ s.t.$K_i \in \mathcal{K}_a$, $q_i \xrightarrow{a}_i$*
- *a is* enabled *in S iff a is globally ready in S and no interaction with higher priority is also globally ready in S.*

Definition 5 (global and local priorities). *Consider a system $DS^{\mathcal{K}}_{<} = (\mathcal{K}, \mathcal{S}, \mathcal{I}, \Delta_<)$. A priority rule $a < b$ is called* local *if the interactions a and b have a common component, i.e., $\mathcal{K}_a \cap \mathcal{K}_b \neq \emptyset$. Otherwise, we call this priority rule* global.

We can now define the usual notions of *concurrency* and *conflict* of interactions, where in a distributed setting we want to allow the independent execution of concurrent interactions (so as to avoid global sequencing). We distinguish explicitly between the usual notion of conflict which we call structural conflict, and a conflict due to priorities.

Definition 6 (concurrent interactions, conflicting interactions). *Let a, b be interactions of $\mathcal{I}$ and $S \in \mathcal{S}$ a global state in which a and b are globally ready.*

- *a and b are called* concurrent *in S iff $K_a \cap K_b = \emptyset$. That is, when a is executed then b is still globally ready afterward, and vice versa, and if executed, both interleavings lead to the same global state.*
- *a and b are called* in structural conflict *in S iff they are not concurrent in S, that is a and b are alternatives disabling each other.*
- *a and b are in (purely)* prioritized conflict *in S iff a and b are concurrent in S but $a < b$ or $b < a$ holds.*

Note that in case of prioritized conflict, it is known which interaction cannot be executed, whereas in case of structural conflict, the situation is symmetric. We use the notations $Concurrent_S(a)$, $Conflict_S(a)$, $PrioConflict_S(a)$ to denote the set of interactions that in state q are concurrent with a, respectively in structural or prioritized conflict with a.

In distributed systems [9,8] the detection of some situations is important for designing correct protocols. Confusion is such a situation occurring when concurrency and conflict are mixed. More precisely, confusion arises in a state where two interactions a_1 and a_2 may fire concurrently, but firing one modifies the set of interactions in conflict with the other (see Definition 7). In presence of priorities, confusion situations may compromise the correctness of a distributed implementation of a specification.

Definition 7 (confusion). *Let a and b be interactions, and S a global state of $DS_<$. We suppose that a and b are concurrent — and thus globally ready — in S.*

- *a is in* structural confusion *with b iff $\exists S' \in \mathcal{S}, S \xrightarrow{b} S'$ implies $Conflict_S(a) \neq Conflict_{S'}(a)$*
- *a is in* prioritized confusion *with b iff $\exists S' \in \mathcal{S}, S \xrightarrow{b} S'$ implies $PrioConflict_S(a) \neq PrioConflict_{S'}(a)$*

In Section 3, we propose a distributed implementation of systems $DS_<$ in which concurrent interactions are executed independently, based on the notion of concurrency of Definition 6 and our implementation does not support systems $DS_<$ with prioritized confusion situations. To realize a distributed implementation of $DS_<^{\mathcal{K}}$ we use a *distributed controller* obtained by a set of local controllers exchanging messages with each other. By a *local controller* of a component K we understand a component that may allow or disallow interactions b of K. b is allowed by offering an interaction synchronizing with b and b is forbidden by not offering it.

Definition 8 (distributed controller). *A finite set of* local controllers *for a distributed system $DS^{\mathcal{K}}$, is a set of labeled transition systems $\{LC_{K_i}\}_{i=1}^n$ each associated to one component K_i of DS. LC_{K_i} restricts the behavior of K_i. In a state $q_i \in Q_i$ of K_i, LC_{K_i} decides which interaction to execute together with other local controllers, and synchronizes with K_i on this interaction. The set of local controllers $\{LC_{K_i}\}_{i=1}^n$ communicate amongst each others, by message passing, to decide which interaction to execute (see Figure 1).*

This definition of controller ensures a part of a safety property stating that only interactions specified by the local behavior of components can be executed as each local controller and its corresponding controlled component synchronize on the interaction a chosen to be executed. Moreover all components in $\mathcal{K}_a$ must also synchronize and execute a and this is what we will prove in Section 4. Thus interactions not specified by $DS_<$ cannot be executed.

The behavior of a given local controller is described by the protocol proposed in Section 3, where we describe how local controllers communicate using messages exchange to schedule an interaction for execution.

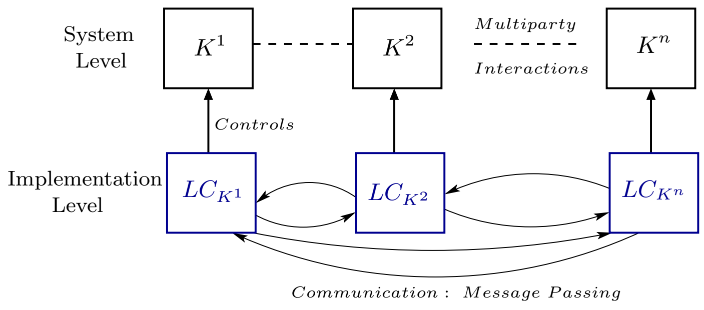

Fig. 1. Local Controllers and their Controlled Components

3 Protocol Description

In this section, we provide an overall picture of the distributed controller that is a *protocol* describing the global behavior of the set of local controllers. The behavior of the controller LC_K depends in each state q of K on the set of interactions locally ready in this state. LC_K exchanges messages with other local controllers of to decide when and which interaction to fire. Once an interaction a is chosen, LC_K synchronizes with K, and thus implicitly also with all other components involved in a to execute it.

In the following, we refer to K instead of LC_K whenever this is not misleading.

To decide when and which interaction to fire, local controllers communicate using *message passing*. We assume that the message passing mechanism, used by the controllers, ensures the following basic properties: (1) any message is received at the destination within a finite delay; (2) messages sent are received in the order in which they have been sent; (3) there is no duplication nor spontaneous creation of messages.

Table 1. Messages used by the Protocol

Message	Description
$POSSIBLE(a)$	If a is locally ready, the controller uses this message to inform the negotiator of a.
$NOTPOSSIBLE(a)$	Respond a negotiator that a is not locally ready.
$COMMIT(a)$	Message sent by the negotiator of a to its peers to lock them or sent back by a peer to respond the negotiator.
$READY(a)$	Sent by a negotiator to ask about the global readiness of a.
$NOTREADY(a)$	Sent by a negotiator to inform that a is not globally ready.
$START(a)$	Message sent by the negotiator to all the peers of a to order the execution of a.
$REFUSE(a)$	Sent by a controller to inform that it cannot commit to a.

The behavior of each local controller LC_K is given by a labeled transition system (see Figure 2) where the states of this LTS represent the different *phases* of the protocol. Transitions represent reactions to messages received by peer components which may consist in a local phase change and/or transmission of messages to some (other) peers. Thus, every transition of Figure 2 is specified by a *message* received, a *guard* and an *action* (see Table 3). Here *message* denotes the message that triggers a transition if the *guard* holds. If there is no message, the transition depends only on its *guard*. The role of each message is described in Table 1 where we use expressions of the form MESSAGE(interaction, sender) to denote the message triggering a given transition. The *action* of a transition denotes the list of statements to be executed if the transition occurs and may include message sending expressions of the form Send(message, recipient).

Some parts of the behavior of LC_K can be performed independently, thus we choose to describe it as a set of labeled transitions systems running in parallel and called *activities* (see Figure 3). These activities share the set of variables depicted in Table 2 and treat a set of disjoint messages. The transitions of Table 3 are performed by these activities where actions of the transitions may describe variable assignments, message sending or creating and killing new *activities*.

For each interaction of the to be controlled system $DS^{\mathcal{K}}_{<}$, we associate the role of *negotiator* to a local controller of one of the components involved in this interaction. Thus, a local controller may be the negotiator of a set of interactions in which its involved and each interaction has exactly one negotiator. The choice of negotiators of interactions and how it may affect the performance of the protocol is discussed in Section 5.1. This particular role of negotiator corresponds to the notion of *coordinator* defined in [17] and to the notion of *manager* presented in [3,2]. Note that comparing to the protocol for binary interactions, presented in [6], here negotiators are assigned to all interactions and not only to interactions involved in some priority rule as is the case in [6]. The reason is that in addition to the role of checking the enabledness of an interaction, the negotiator here checks also its global readiness. Thus, a controller presents a two phase behavior. A first phase is collecting knowledge about the global readiness of possible interactions and a second negotiating the enabledness of ready interactions. The phase *Active* is first entered by the transition 0, providing the set of interactions locally ready ($possibleSet(q_0)$) of the initial state of the controlled component. The controller LC_K looks for a next interaction to fire by proceeding as follows:

■ Once in phase *Active*, the activities $Main$ and $WaitingForCommit$ are created and run in parallel. $Main$ starts by checking its locally ready interactions (*possibleSet*) for interactions that are globally ready and for which it is the *negotiator*. For interactions in *possibleSet* for which LC_K is not the negotiator, it only informs their corresponding negotiators about the local readiness of the interaction.

To check the global readiness of an interaction a, messages of the form $POSSIBLE(a)$ are exchanged (Transition 1 of Figure 2), and peers in which

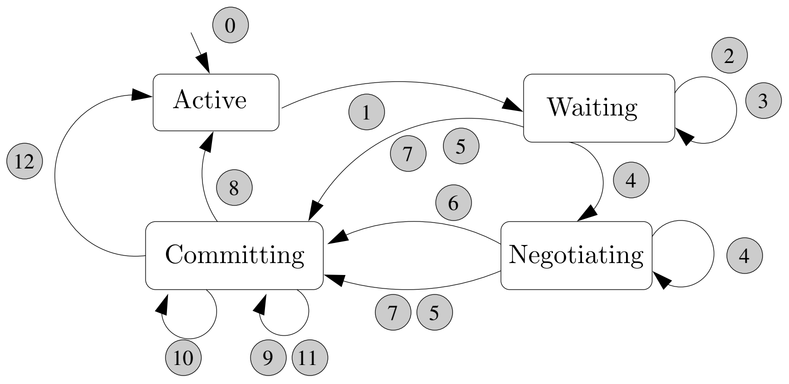

Fig. 2. The Phases of the Local Controller behavior (see Table 3 for the transitions)

a is currently not locally enabled respond with $NOTPOSSIBLE(a)$ after which the requesting controller abandons a.

If LC_K is the negotiator of a ($a \in toNegotiate$) and if it collects $POSSIBLE(a)$ from all the participants of a, then a is detected to be *globally ready*. If, in addition, a is involved in a priority rule, $Main$ creates a new activity $Negotiate(a)$ which checks whether a is enabled and the LC_K goes to phase $Negotiating$ through transition 4.

▪ $Negotiate(a)$ activity checks the enabledness of an interaction a by sending a $READY(b)$ message to all negotiators of interactions b with higher priority than a ($b \in higherPrio(a)$), it checks whether their interactions are globally ready (and thus a cannot be executed now).

In turn the negotiators of b, as soon as they are not executing an interaction and have found out whether b is globally ready, respond positively or negatively as soon as they have the information available. In fact, it is sufficient that $NOTREADY(b)$ messages are sent as a is blocked anyway as long as it does not have a response concerning b.

▪ If an interaction with maximal priority is globally ready, it is immediately known to be enabled and a *Committing* phase is entered through transitions 5 or 6 by killing activities $Main$, $WaitingForCommit$ and $Negotiate$ and creating $TryToStart$ (see further down).

▪ The $Main$ activity, which is running while LC_K is in the phases *Active*, $Negotiating$ and $Waiting$, handles local priorities locally. Whenever an interaction b is known to be globally ready, it kills all activities $Negotiate(a)$ if $a < b$.

▪ Concurrently to $Main$, the activity $WaitingForCommit$ handles incoming *COMMIT* messages. Whenever a $COMMIT(a)$ is received from the negotiator of a, which means that a is checked to be enabled by the negotiator of a. All other negotiation activities are terminated and a $TryToStart$ activity is created (Transition 7). The existence of the activity $WaitingForCommit$ means that no other actions is in its *Committing* phase yet.

■ To avoid multiple commits for different interactions, $COMMIT$ message is only sent by $TryToStart$ activity as it is created when all other activities terminate.
■ If LC_K is the negotiator of a, then $TryToStart(a)$ sends a $COMMIT(a)$ message to all participants of a and waits for a $COMMIT$ response from all of them. Once, all participants send back a $COMMIT$, the negotiator orders the execution of a by sending a $START$ message and it executes a together with the controlled component. Note that if $TryToStart$ fails committing to a because it receives a $REFUSE$ message from at least one of the participants — in that case the peer has committed to a conflicting interaction — the controller starts again by checking the global readiness of its locally ready interactions (transition 8).
■ If LC_K is not the negotiator of a, then $TryToStart(a)$ sends a $COMMIT(a)$ message to the negotiator of a and waits for a $START$ or a $REFUSE$. If it receives $START$, the controller executes a together with the controlled component which corresponds to the transition 12 of Table 3. If a $REFUSE$ message is received, then activity $TryToStart(a)$ terminates and new $Main$ and $WaitingForCommit$ activities are created.
■ Finally, an activity $AnswerNegotiators$ (not represented in Figure 3) is always running in all states of the state diagram of Figure 2, if LC_K is the negotiator for at least one interaction a that dominates some other interaction. This activity receives messages of the form $READY(a)$. It returns $NOTREADY(a)$ if a is in the $notReadySet$, returns $READY(a)$ if a is in the $readySet$, and otherwise defers the answer until the status of a is known.

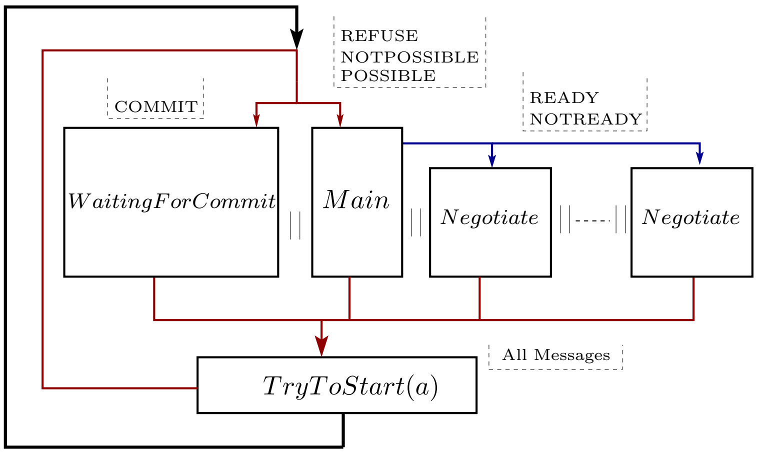

Fig. 3. Structure of the Protocol for a Local Controller

Avoiding Deadlocks Due to Decision Cycles

The protocol as described above may lead to a deadlock or a livelock in a situation where a set of components are all ready to execute at least 2 from a set of conflicting enabled interactions. Such a situation may occur in *Committing*

Table 2. Variables used by the Protocol

Variable	Description
$possibleSet(q)$	The set of interactions locally ready in q.
$readySet$	The set of interactions detected to be globally ready.
$notReadySet$	The set of interactions which are locally ready but detected to be not globally ready.
$toNegotiate$	The set of interactions for which the local controller is the negotiator.
$lessPrio(a)$	Interactions locally ready with less priority than a
$higherPrio(a)$	Interactions locally ready with higher priority than a.
$Peers(a)$	The set of participants in the interaction a.
$Neg(a)$	The negotiator of the interaction a.
$ReadyPeers(a)$	The set of participants in a for which a is locally ready.

phase of the protocol, where a negotiator has sent a $COMMIT$ message to all participants and waits for their response. Similarly one of these participants could be a negotiator of a different interaction and being in a *Committing* phase. This may lead to a deadlock if the set of interactions for which the negotiators are committing forms what we call a *cycle*.

Definition 9. *A cycle is a set of interactions* $A = \{a_i\}_{i=1}^n$ *involving a set of components* $\{K_i\}_{i=1}^n$ *for which the following holds: For all* $i \in [1, n]$, a_i *is an interaction involving the two components* K_i *and* $K_{\{i+1 mod n\}}$ *and there exists at least one global state in which all these interactions are enabled. We denote the fact that A is a cycle by* $Cycle(A)$. *Note that in such a global state each* K_i *has at least two enabled interactions, in the corresponding local state, one interaction with* K_{i-1} *and the other with* K_{i+1}.

To avoid deadlocks due to such cycles, we use a solution that we have already proposed for the binary version of our protocol described in [6]. The idea is to detect statically the set of potential cycles of the system. Then, we define for each cycle statically one of the components involved as a *Cyclebreaker*. Whenever a potential deadlock may be reached (from the point of view of the Cyclebreaker), the *Cyclebreaker* will commit to one of the interactions and refuse the other. This approach avoids defining a total order over all interactions or components, as proposed in [1,17], which may lead to systematically avoiding certain interactions and which may compromise liveness. Our solution is more faithful to the initial description of the system as it does not exclude any interactions unless an actual cycle occurs. Thus to avoid deadlocks due to cycles, if a given controller sends a *COMMIT* message and then it receives another *COMMIT* message for a different interaction, then either there is no cycle involving these two interactions or there exists at least one. In the second case, if the received *COMMIT* concerns the interaction committed by a *Cyclebreaker*, then the controller cannot send back a *REFUSE* and thus if this interaction is not the one committed by the

Cyclebreaker, the controller will send back a *REFUSE* which breaks the cycle. To perform this solution locally, we define pairs of interactions representing the local view of LC_K about a given cycle.

Notation 1. *We denote by $cyclesof(K)$, the set of pairs of interactions of K involved in some cycle. $(a, b) \in cyclesof(K)$ implies that a and b are interactions of K and $\exists$ A such that $Cycle(A) \wedge \{a, b\} \subseteq A$.*
We denote by $Cyclebreaker(A, K)$ the predicate which holds if the component K is the Cyclebreaker *of a cycle A.*
We denote also by $notRefuse(K)$ the set of pairs of interactions of the form (a, b) such that $(a, b) \in notRefuse(K)$ implies:

1. *$(a, b) \in cyclesof(K)$*
2. *$\forall$ cycles A such that $\{a, b\} \subseteq A$, $Cyclebreaker(A, K_a)$ holds, where K_a is a participant in the interaction a. This means that whenever K sends $COMMIT(b)$ message, and then it receives $COMMIT(a)$, it will not send back $REFUSE(a)$.*

Note that the order of interactions of a pair in $notRefuse(K)$ is relevant as the first interaction is the one that cannot be refused by K. Note that a pair of interactions $(a, b) \notin notRefuse(K)$ means that either there is no cycles involving these two interactions ($(a, b) \notin cyclesof(K)$) or that there exist such cycles ($(a, b) \in cyclesof(K)$) but if K commit for b and receives a *COMMIT* message for a then it can send back a $REFUSE(a)$ to its peer K_a because the latter is not the *Cyclebreaker* of these cycles. Theorem 3 proves that this way to deal with cycles allows indeed to avoid deadlocks.

4 Correctness

We now prove that the proposed protocol satisfies the following properties [2]:

- Correctness: Only interactions allowed by $DS_{<}^{\mathcal{K}}$ can be executed:
  1. only locally ready interactions can be executed.
  2. if a component executes an interaction, the remaining components participating in that interaction will execute it (Synchronization).
  3. interactions in conflict (structural or prioritized conflict) cannot be committed simultaneously (Safety).
- Progress: when an interaction is enabled it will eventually be executed or one of its participants executes a conflicting interaction.

The first item of the correctness property is guaranteed by Definition 8, as each local controller and its corresponding controlled component synchronize on the interaction chosen to be executed. Thus only interactions which are locally ready for components can be executed.

Theorem 1 (Synchronization). *Our protocol guarantees that if a participant executes interaction b, then all of the components participating in b will execute it.*

Table 3. Transitions of the Local Controller State Diagram

Tr	*message*	*guard*	*action*
0		possibleSet(q_0)$\neq \emptyset$, q_0: initial state of K	Create($Main$), Create($WaitingForCommit$)
1		possibleSet$\neq \emptyset$	$\forall a$ $\in$(possibleSet$\cap$toNegotiate), send(POSSIBLE(a),Neg(a)) $\forall a$ $\in$(possibleSet$\setminus$toNegotiate), send(POSSIBLE(a), Peers(a))
2	POSSIBLE(a)	(a $\in$possibleSet$\setminus$toNegotiate)	send(POSSIBLE(a),Neg(a))
2	NOTPOSSIBLE(a)	(a $\in$possibleSet$\cap$toNegotiate)	$notReadySet$:= $notReadySet \cup a$
3	POSSIBLE(a, K)	(a $\in$possibleSet$\cap$toNegotiate) $\wedge$(Peers(a)$\neq$readyPeers(a)$\cup K$)	readyPeers(a):= readyPeers(a)$\cup K$
4	POSSIBLE(a, K)	(a $\in$possibleSet$\cap$toNegotiate) $\wedge$(Peers(a)==readyPeers(a)$\cup K$) $\wedge$ $a \notin$ prioFree	create(Negotiate(a)), readySet:=readySet$\cup a$, ($\forall$ b $\in$lessPrio(a),kill(Negotiate(b))
5	POSSIBLE(a, K)	(a $\in$possibleSet$\cap$toNegotiate) $\wedge$(Peers(a)==readyPeers(a)$\cup K$) $\wedge$ $a \in$ prioFree	Kill(WaitingForCommit), Kill(Main), Send(COMMIT(a),Peers(a))
6		$Negotiate$(a)==OK	Kill(WaitingForCommit) Kill(Main), Send(COMMIT(a),Peers(a))
7	COMMIT(a)	(a $\in$possibleSet$\setminus$toNegotiate)	Kill(WaitingForCommit) Kill(Main), Send(COMMIT(a),Neg(a)
8	REFUSE(a)	Committed(a)	Goto(Active),Reset(readySet), Keep(possibleSet), Create($Main$), Create($WaitingForCommit$).
9	COMMIT(b)	Committed(a),($a \neq b$) $(a,b) \notin cyclesof(K)$ or $(a,b) \in notRefuse(K)$	waitingSet:=waitingSet$\cup\{b\}$
10	COMMIT(b)	Committed(a),($a \neq b$)and($(a,b) \in cyclesof(K)$ and $(a,b) \notin notRefuse(K)$)	Send(REFUSE(b), Peers(b))$\wedge$readySet:=readySet$\setminus\{b\}$
11	COMMIT(a, K)	Peers(a)$\neq$ readyToCommit(a)$\cup$K, $a \in$ possibleSet$\cap$toNegotiate	readyToCommit(a):= readyToCommit(a)$\cup K$
12	COMMIT(a, K)	Peers(a)==readyToCommit(a)$\cup$K, $a \in$ possibleSet$\cap$toNegotiate	Send(START(a), Peers(a)) and $\forall$ b $\in$possibleSet, Send(REFUSE(b), Peers(b))$\wedge$ Execute(a)
12	START(a)	$a \in$ possibleSet$\setminus$toNegotiate	$\forall b$ $\in$possibleSet,$b \neq a$, Send(REFUSE(b),Peers(b)), Execute(a),Update($possibleSet(q)$), Create($Main$), Create($WaitingForCommit$).

Proof. What we have to prove is that if a participant K_i of the interaction b executes it, then the rest of participants will also do so. If a component K_i which is the negotiator of b, executes b then it must according to transition 12 of Table 3 have received $COMMIT$ messages from all the participants of b. Note that $COMMIT$ is a blocking message, which means that all participants will stay waiting for a response from the negotiator and once they receive the $START$ message they will also execute b. If a component K_j which is not the negotiator of b, executes b then it must according to transition 12 of Table 3 have received a $START$ message from the negotiator K_b of b. This means that similarly, K_b has sent a $START$ message to all participants of b, and as previously detailed all these participants are in a blocking state waiting for the message of the negotiator.

Theorem 2 (Safety). *Let be S a global state, b_1 an interaction and denote A the set $Conflict_S(b_1) \cup PrioConflict_S(b_1)$ of interactions that are in conflict with b_1 in the global state S. Our protocol guarantees that if b_1 is fired in state S, no interaction in A is fired in S.*

Proof. Suppose for $b_2 \in A$ that:
First case: $b_2 \in Conflict_S(b_1)$, that is b_1 and b_2 share a common component K. First of all, only interactions, for which the corresponding negotiator has sent a $START$ message to all participants, are executed. If the common component participating in b_1 and b_2 is the negotiator of both interactions, then only one interaction can be executed as according to transition 12 of Figure 2 a negotiator can send $START$ only for one interaction at a time and the property is satisfied. If b_1 and b_2 have different negotiators. Suppose that both negotiators have sent a $START$ message to execute b_1 and b_2. This means that all participants involved in b_1 and b_2 have sent a $COMMIT$ message to their negotiators (according to transition 12 of Table 3). As K is a common component, then this means that K has sent two $COMMIT$ messages one for the negotiator of b_1 and one for those of b_2 which is impossible as only one $COMMIT$ message can be sent at a time. In fact, a $COMMIT$ message can only be sent by the $TryToStart$ activity which does not have any other concurrent activity (see Figure3).
Second case: $b_2 \in PrioConflict_S(b_1)$, that is b_1 and b_2 are concurrent (and thus belong to different components) and either $b_1 < b_2$ or $b_2 < b_1$. Suppose that K_{b_1} is the negotiator for b_1 and K_{b_2} is the negotiator for b_2.

If $b_2 < b_1$, then b_2 should not be executed before the execution of b_1 — which has started — has been completed and K_{b_1} enters *Active* phase for the successor state of S. We have now to prove that from that moment on K_{b_2} cannot "believe that b_1 is *not ready*" which is the condition for committing to b_2.

Indeed, if K_{b_2} does not yet know about the readiness of b_1, before committing b_2, it will send a $READY(b_1)$ message to K_{b_1}, but as b_1 is already engaged for execution, K_{b_1} will not send any response before the execution of b_1 is terminated the next state reached, and the readiness of b_1 evaluated in the new state; and K_{b_2} remains blocked for b_2 during this time.

Now, we must prove that K_{b_2} cannot have old, depreciated knowledge that b_1 is not ready. This can only be the case, if at some point b_1 was not ready and K_{b_1} has sent $NOTREADY(b_1)$ to K_{b_2}, and then transitions concurrent to b_2 have been executed leading to the current state S in which b_1 is ready and executed, and K_{b_2} may use incorrect knowledge and execute b_2. This corresponds exactly to a situation of confusion, which we have excluded (see Section 2). If $b_1 < b_2$, the situation is almost symmetric.

Lemma 1. *If a negotiator K_1 of an interaction a_0 sends a COMMIT message to a participant K_2, then K_1 will receive a REFUSE(a_0) or a COMMIT(a_0) message from K_2 within a finite delay.*

Proof. We assume that the actual execution of an interaction a_0 as well as all the basic functions used in our protocol terminate and every message reaches its recipient within a finite delay. If K_1 waits for a response, after sending a *COMMIT*(a_0) message to K_2, this means that it exists a global state S of the system in which a_0 is enabled and that K_2 is in the phase *Committing*(a_1) $(a_0 \neq a_1)$ (see the diagram of Figure 2). Indeed K_2 cannot be in any of the rest of the phases $Waiting$, *Active* or $Negotiating$ as the activity $WaitingForCommit$ running in these phases (see Figure 3) will catch this *COMMIT*(a_0) message and will send back a *COMMIT*(a_0) to K_1. Thus, K_2 does not respond because it is trying to commit to another interaction $a_1 \neq a$. Which means that K_2 is in the phase *Committing*(a_1) and that in the same global state S, a_1 is enabled. According to the Table 3 one of the following cases holds:

1- $(a_0, a_1) \in cyclesof(K_2)$ and $(a_0, a_1) \notin notRefuse(K_1)$ (according to the guard of transition 10 of Table 3), in this case K_2 sends back a *REFUSE*(a_0) to K_1 within a finite delay.
2- $(a_0, a_1) \notin cyclesof(K_2) \vee (a_0, a_1) \in notRefuse(K_2)$, in this case K_2 is also waiting for an answer from K_3 about a_1. Similarly, if K_3 does not answer with a *REFUSE*(a_1), then it exists an interaction a_2 enabled in S such that $(a_1, a_2) \notin cyclesof(K_3) \vee (a_1, a_2) \in notRefuse(K_3)$. As there exists a finite number n of components in the system, this means that there exists some cycle of size k in S for which the following holds:

$$\begin{gathered}(a_0, a_1) \notin cyclesof(K_2) \vee (a_0, a_1) \in notRefuse(K_2)\\(a_1, a_2) \notin cyclesof(K_3) \vee (a_1, a_2) \in notRefuse(K_3)\\\dots\\(a_{k-2}, a_{k-1}) \notin cyclesof(K_k) \vee (a_{k-2}, a_{k-1}) \in notRefuse(K_k)\end{gathered}$$

This is a contradiction. Indeed, the first part of each property means that there is no cycle containing these interactions, which is not true as we have a circular sequence which means a cycle. The second part does not hold as we assume that each cycle has exactly one *Cyclebreaker* which may try to commit to one of the interactions only.

Theorem 3 (Progress). *Let b be an* enabled *interaction. Our protocol guarantees that b will eventually be executed or a component participating in it executes another interaction.*

Proof. The enabledness of an interaction is first detected by the negotiator of this interaction. An interaction b is enabled for its negotiator K_b, when COMMIT(b) messages are sent from all the participants of b. When it is detected to be enabled the negotiator of b goes to phase *committing*(b). b becomes disabled when its negotiator leaves this state either by executing b (through transition 12 according to Table 3) or because one of the participants of b executes another interaction (through transition 8 according to Table 3). In other words what we have to prove is that the negotiator K_b of b cannot stay in this state eternally. When K_b is in phase $committing(b)$, then it has send $COMMIT$ messages to all participants of b, and according to Lemma 1, K will receive eventually a $COMMIT$ messages from all participants or at least one $REFUSE$ from one of the participants and thus will leave the phase $committing(b)$ through transition 12 or 8.

5 Experimental Results

In this section, we report the results of experiments undertaken using an implementation of our protocol. Our implementation uses JAVA 1.6 to ensure the different algorithm computations and Message Passing Interfaces (MPIs) [15,19] to ensure the communication layer between components. Two metrics have been used to evaluate the performance of the protocol, namely *response-time* and *message-count*. The metric *message-count* computes the average number of messages needed to schedule one interaction for execution. The *response-time* is measured from the instant at which an interaction becomes locally ready (as viewed by its negotiator) to the instant at which it is selected for execution by the protocol. This metric is defined as the sum of two other metrics: *sync-time* and *selection-time*: *sync-time* measures the (mean) time taken by the algorithm to ensure that a given interaction is globally *ready*, starting from the moment when it is locally ready in its negotiator. *selection-time* measures the (mean) time taken by the algorithm to select an interaction for execution once it has been found globally ready. Note that the enabledness of an interaction is checked during the *selection-time*. *sync-time* is independent of priorities between interactions.

5.1 Sensitivity to the Choice of Negotiators

We illustrate the sensitivity of our protocol to the choice of negotiators by means of the well-known Dining Philosophers problem [12]. As proposed in [17], using multiparty interactions, a simple solution to this problem can be provided as each philosopher could pickup both two forks at a time by means of a three-party interaction (see Figure 5). However, using only binary interactions, the solutions to this problem must rely on some distinction amongst the behaviors of the philosophers, which makes such solutions not scalable nor reusable [16]. This problem models any situation where any entity needs to access a set of resources in mutual exclusion.

Using this example, we study how the choice of negotiators in a system may affect the performance of the protocol. We have carried out a series of experiments for the system of dining philosophers depicted in Figure 5 in the case of 2, 3 and 4 philosophers.

For each case, we have measured the already described metrics (message-count, sync-time and selection-time) to execute one interaction and we have focused on two configurations depending on the choice of the negotiators for interactions. In a first configuration, we have assigned as negotiator of the interaction the component *Philosopher* involved in this interaction. Then, in a second configuration we have assigned the *Fork* component as negotiator for interactions in which it is involved. Figure 4, shows that the *message-count* when the *Philosophers* are the negotiators is higher than the case when *Forks* components are chosen as negotiators.

This is expected as the component *Fork* is involved in more than one interaction and thus it has more knowledge about the state of the participants of these interactions. However, the component *Philosopher* is involved in only one interaction and so, when it has to schedule an interaction for execution, it needs to communicate with other components to get knowledge about their states and thus exchanges more messages. More precisely, in the case of the system with two Philosophers, when the component $Philosopher_i$ is the negotiator of the interaction $getForks_i$, then according to the description of the protocol provided in Section 3, each *Philosopher* tries to commit for its interaction by sending to both *Forks* a *COMMIT* message making a total of 4 *COMMIT* messages. When a component $Fork_1$, for example, is the negotiator of both interactions in the system namely, $getFork_1$ and $getFork_2$, then $Fork_1$ tries to commit to only one interaction which means that only 2 *COMMIT* messages will be sent by the negotiator. Consequently, when assigning negotiators in a system, the designer has to take into account the number of interactions in which negotiators are involved. Thus, the more the interactions in which a negotiator is involved, the less it needs to exchange messages. Similarly, the response-time metric is also affected by the choice of negotiators as it can be observed in the right-hand side of Figure 4.

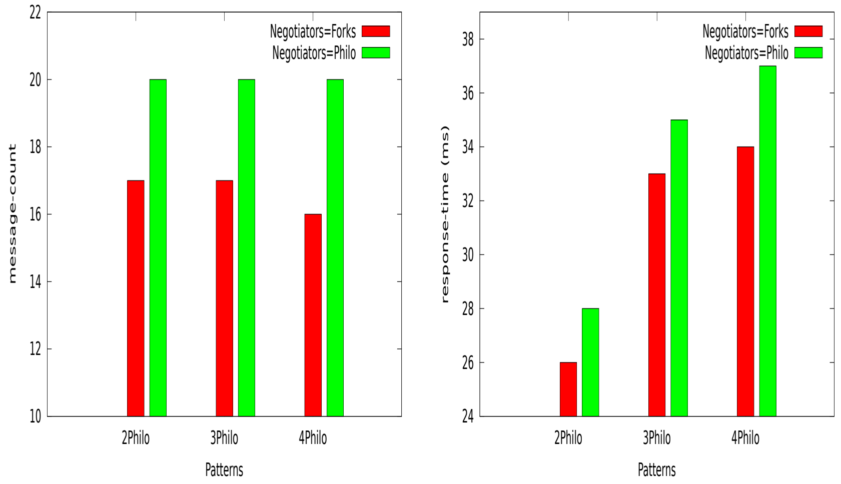

Fig. 4. Sensitivity to the choice of negotiators

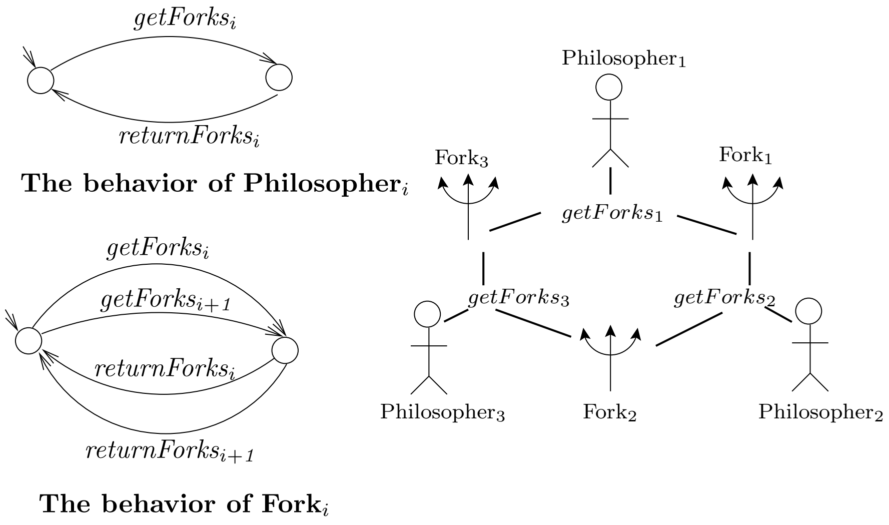

Fig. 5. The Dining Philosophers with multiparty interactions

5.2 Example with Priorities: Jukebox System

We use a Jukebox example to illustrate the use of priorities and to study how our algorithm performs, in particular, when global priorities are defined. The system is defined by a set of readers (R_1, ...,R_4) which need to access data located on Discs (D_1,D_2). Access to discs is handled by Jukebox components (J_1,J_2). Each Jukebox can load any disc to one of the readers it manages. Figure 6, represents the structure of the Jukebox system, where each *load* interaction corresponds to a $load_{J_i}D_iR_i$ interaction with J_i, D_i and R_i are respectively the connected Jukebox, Disc and Reader. The Jukebox J_1 manages the access of the readers R_1 and R_2, and J_2 the access of the readers R_3 and R_4. The behavior of each component is depicted in Figure 7. The interaction $load_{J_i}D_iR_i$ is a three-party interaction between the Jukebox J_i, The Disc D_i and the Reader R_i allowing, J_i to load D_i for the reader R_i. The interaction $unload_{j_i}D_i$ of unloading the disc is a binary interaction as it does not involve the reader. In Figure 7, the jukebox system is modeled without priorities. However, two types of priorities could be defined:

- Priorities to enforce termination: We give priority to *load* interactions over the *unload* ones. Formally, it defines the following priorities: $\{unload_{j_j}D_i < load_{J_j}D_iR_i\}_{j\in\{1,2\}}$. Note that this set of priorities defines *local priorities* as they include interactions of a common component namely the jukebox. Table 4, depicts the different results obtained when running the Jukebox system and measures the time taken and the *message-count* for the execution of two *load* interactions. Note that with priorities, the time taken to satisfy two readers is considerably lower than for the case without priorities. However, as introducing priorities needs more communication a main drawback is the *message-count*.

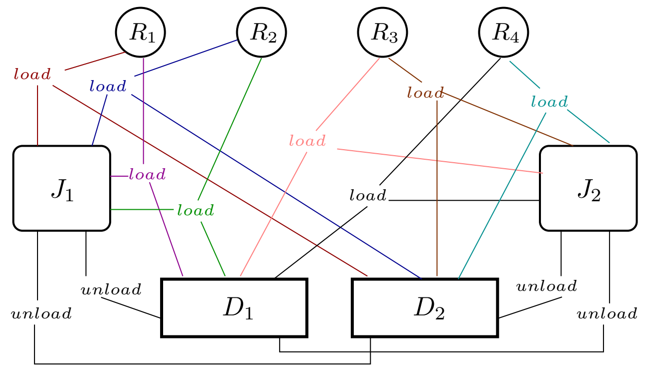

Fig. 6. The Jukebox System

– Priorities to manage resource access: We define priorities between Readers when accessing to a given Disc. We define for example the priority rule: $load_{J_1}D_1R_1 < load_{J_2}D_1R_3$ which means that whenever the Readers R_1 and R_3 want to load the Disc D_1, the priority is given to the Reader R_3. Such a priority rule involves the Disc as a common component which can consequently ensure this rule locally. However, in general, resources are represented by passive components which are not designed to manage or to make decisions. As managing access to resources is handled by the Jukebox component, we assign J_1 and J_2 as negotiators of the *load* interactions. Thus such a priority will be handled as a *global* priority rule, as components J_1 and J_2 have to communicate to decide which interaction to fire. Without this priority rule, for 20 executions of *load* interactions, which may involve any of the 4 Readers, we obtain an average of 915 messages exchanged. However, with $load_{J_1}D_1R_1 < load_{J_2}D_1R_3$, no interaction *load* for Reader 1 takes place and the average of the messages exchanged is about 1050, which is expected as global priorities are handled by the exchange of additional messages, namely $READY$ and $NOTREADY$, between the negotiators.

Table 4. Enforcing Termination using Priorities

	elapsed-time (ms)	message-count
Without priorities	580	65
With priorities	300	75

6 Related Work

This paper handles two important issues in the context of distributed control.

The first one concerns the enforcement of global properties, which are here priorities between interactions, in a distributed setting which is challenging to implement [5,11].

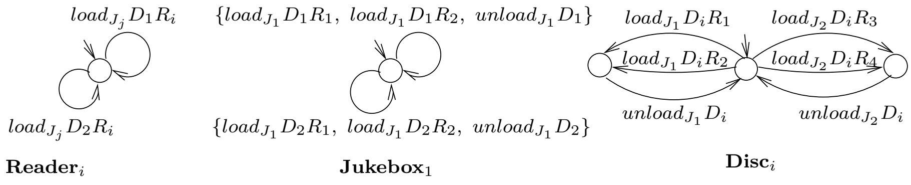

Fig. 7. Components of the Jukebox System

In [14,4,7], model-checking and *knowledge* are used to transform the system with priorities into a new system without priorities by restricting the possible choices in order to impose priorities. This means that they reduce concurrency of the initial system, whereas in our approach we guarantee *maximal progress*, which means that we allow all possible interactions of the initial description.

Similarly, to enforce priorities the approach proposed in [10] codes the priorities in the behavior of the initial system, so as to obtain a new model without priorities, then implements the so-obtained model in a distributed setting. Such approach adds to the system particular components called *managers* associated to each interaction which increase considerably the size of the studied system. Moreover, this makes their implementation less flexible to any change on the set of priorities as it means changing the system structure whenever the set of priorities is changed. Whereas in our approach the implementation of the protocol is still the same to any set of priorities.

Second, we propose a new protocol implementing *multiparty* interactions in a distributed setting. In [2,1,17], similar algorithms have been proposed but no *global* priorities are handled. In [3] *local* priorities have been defined to deal with deadlocks due to decision cycles as each controller explores its *possibleSet* in a decreasing order of priority which means that exactly one interaction, which has the highest priority, will be always executed if it is enabled. Similarly, to deal with such problem, the α-core algorithm proposed in [17] defines a total order between components which also means that some interactions can never be executed if they are in conflict with the interactions of the component with the highest priority, which limits the variability of the executions of the initial system. However, the solution we propose to deal with deadlocks due to decision cycles, uses some static *knowledge* about the structure of the system to define beforehand the set of potential cycles and then defines, only when needed, a priority given to the interaction chosen by the *Cyclebreaker*.

The α-core algorithm defines for each interaction a particular component called *Coordinator* managing the corresponding interaction and collecting information from all its participants to decide about its execution. In our approach the coordinator, i.e. the negotiator is one of the participant which allows to reduce the number of messages exchanged as the negotiator exploits already some local knowledge.

Similarly, in [1], *managers* are associated to interactions. However, a given manager is responsible for managing a subset of interactions and thus managing conflicts between managers is achieved by means of a circulating token allowing the manager having it to execute its corresponding non-conflicting interactions. This solution based on a circulating token may lead to a situation in which a manager can never execute its interactions as it never gets the token at the right moment.

7 Conclusion

In this paper, we have focused on component-based systems with multiparty interactions and priorities. We have provided a protocol to implement such systems in a distributed setting. A variety of protocols for implementing multiparty interactions exist in the literature, but our approach is innovative as it handles in addition global priorities between interactions. An implementation of the proposed protocol is also provided with a set of experimental results allowing to analyze its performance. There are several research directions for future work. First, more experimentation is needed in particular to compare the performance of the protocol to existing approaches. Second, we are interested in combining our approach with *knowledge*-based methods, as it is proposed in [4], in order to optimize the performance of our protocol by taking into account a pre-calculated *knowledge* and thus reducing communication between local controllers.

References

1. Bagrodia, R.: A distributed algorithm to implement n-party rendezvous. In: Nori, K. (ed.) FSTTCS 1987. LNCS, vol. 287, pp. 138–152. Springer, Heidelberg (1987)
2. Bagrodia, R.: Process synchronization: Design and performance evaluation of distributed algorithms. IEEE Trans. Software Eng. 15(9), 1053–1065 (1989)
3. Bagrodia, R.: Synchronization of asynchronous processes in CSP. ACM Trans. Program. Lang. Syst. 11(4), 585–597 (1989)
4. Basu, A., Bensalem, S., Peled, D., Sifakis, J.: Priority Scheduling of Distributed Systems Based on Model Checking. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 79–93. Springer, Heidelberg (2009)
5. Basu, A., Bidinger, P., Bozga, M., Sifakis, J.: Distributed Semantics and Implementation for Systems with Interaction and Priority. In: Suzuki, K., Higashino, T., Yasumoto, K., El-Fakih, K. (eds.) FORTE 2008. LNCS, vol. 5048, pp. 116–133. Springer, Heidelberg (2008)
6. Ben-Hafaiedh, I., Graf, S., Quinton, S.: Building distributed controllers for systems with priorities. Journal of Logic and Algebraic Programming (2010)
7. Bensalem, S., Peled, D., Sifakis, J.: Knowledge based scheduling of distributed systems. In: Manna, Z., Peled, D.A. (eds.) Time for Verification. LNCS, vol. 6200, pp. 26–41. Springer, Heidelberg (2010)
8. Bolton, C.: Adding Conflict and Confusion to CSP. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 205–220. Springer, Heidelberg (2005)
9. Bolton, C.M.: Capturing Conflict and Confusion in CSP. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 413–438. Springer, Heidelberg (2007)

10. Quilbeuf, J., Bonakdarpour, B., Bozga, M.: Automated distributed implementation of component-based models with priorities. Technical Report TR-2011-3, Verimag Research Report
11. Mani Chandy, K., Lamport, L.: Distributed snapshots: determining global states of distributed systems. ACM Trans. Comput. Syst. 3(1), 63–75 (1985)
12. Dijkstra, E.W.: Hierarchical ordering of sequential processes, pp. 198–227. Springer-Verlag New York, Inc., New York (2002)
13. Gößler, G., Sifakis, J.: Priority Systems. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2003. LNCS, vol. 3188, pp. 314–329. Springer, Heidelberg (2004)
14. Graf, S., Peled, D., Quinton, S.: Achieving Distributed Control through Model Checking. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 396–409. Springer, Heidelberg (2010)
15. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: portable parallel programming with the message-passing interface, 2nd edn. MIT Press, Cambridge (1999)
16. Lynch, N.A., Merritt, M., Weihl, W.E., Fekete, A.: Atomic Transactions. Morgan Kaufmann, San Francisco (1993)
17. Pérez, J.A., Corchuelo, R., Toro, M.: An order-based algorithm for implementing multiparty synchronization. Concurrency - Practice and Experience 16, 1173–1206 (2004)
18. Rudie, K., Murray Wonham, W.: Think globally, act locally: decentralized supervisory control. IEEE Transactions on Automatic Control 37(11), 1692–1708 (1992)
19. Snir, M., Otto, S.W., Huss-Lederman, S., Walker, D.W., Dongarra, J.: MPI: The complete reference. MIT Press, Cambridge (1996)