Knowledge Based Transactional Behavior

Saddek Bensalem, Marius Bozga, Doron Peled, Jean Quilbeuf

Knowledge Based Transactional Behavior*

Saddek Bensalem[1], Marius Bozga[1], Doron Peled[2], and Jean Quilbeuf[1]

[1] UJF-Grenoble 1 / CNRS, VERIMAG UMR 5104, Grenoble, F-38041, France
[2] Department of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel

Abstract. Component-based systems (including distributed programs and multiagent systems) involve a lot of coordination. This coordination is done in the background, and is transparent to the operation of the system. The reason for this overhead is the interplay between concurrency and non-deterministic choice: processes alternate between progressing independently and coordinating with other processes, where coordination can involve multiple choices of the participating components. This kind of interactions appeared as early as some of the main communication-based programming languages, where overhead effort often causes a restriction on the possible coordination. With the goal of enhancing the efficiency of coordination for component-based systems, we propose here a method for coordination-based on the precalculation of the knowledge of processes and coordination agents. This knowledge can be used to lift part of the communication or synchronization that appears in the background of the execution to support the interaction. Our knowledge-based method is orthogonal to the actual algorithms or primitives that are used to guarantee the synchronization: it only removes messages conveying information that knowledge can infer.

1 Introduction

Component-based systems are a generalization of distributed systems. In concurrent languages like CSP and ADA processes allow binary interactions between processes, often with the choice between outgoing communication restricted to be deterministic. Modern distributed systems may involve more general multi-party coordination, e.g., robots that need to coordinate temporarily on a certain task. While such a system may reveal a behavioral model that is based on interaction primitives, often in the back, there are algorithms that are based on more basic primitives such as asynchronous message passing or shared variables. Algorithms for obtaining synchronization primitives are complicated and require nontrivial overhead. Theoretical results also show some inherent restrictions: a well known result on the dinning philosophers [13] shows that a completely symmetric nonprobabilistic solution cannot exist.

We present here a method for improving the behavior of synchronous interactions by removing some of the overhead for guaranteeing the correct synchronization of components based on knowledge calculation. The main principle is based on the observation that such algorithms need to allow for a very general interaction, but can provide a much

* The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7] under grant agreements no 248776 (PRO3D) and no 257414 (ASCENS) and from ARTEMIS JU grant agreement 2009-1-100230 (SMECY)

more efficient behavior for more limited cases. Analyzing the system before its execution based on model checking of knowledge properties allows us to utilize the particular behavior that is actually needed for the implementation of the synchronization. Knowledge, basically, refers to the facts that hold in all the global states that are consistent with the current local state of some process. A precalculated knowledge, embedded in the processes, allows exploiting the easier cases of behavior, when relevant.

Our method is general, independent of the actual synchronization algorithm or primitives used to obtain it. However, the actual implementation of the method depends on the specific details of the algorithm. We present its implementation on a well known generic synchronization algorithm called α-core [15].

The paper is organized as follows. Section 2 recalls cellular automata, as the underlying semantic model for synchronizing systems, and the α-core protocol, as one possible solution for distributed implementation of such systems. Section 3 presents the key results on exploiting knowledge to reduce the communication overhead for distributed implementation. We provide techniques for using knowledge independently for components and coordinators as well as for combining them. Section 4 reports experimental results obtained using a prototype implementation realized on top of the BIP framework [3]. Finally, Section 5 provides conclusions and future work directions.

2 Preliminaries

2.1 Cellular Automata

The model of execution that we want to obtain is that of synchronizing systems. To describe such systems we are using cellular automata. This model involves several processes, represented as automata with transitions labeled by action names, where the execution of all the actions that share the same name has to be synchronized by all processes. Formally, the cellular automata model is defined as follows:

Definition 1. *An automaton is a tuple* $\langle S, A, \delta, s_0 \rangle$ *where* S *is the set of* states, A *is the set of* actions, $\delta : S \times A \to S$ *is the transition relation,* $s_0 \in S$ *is the* initial state. *An execution of an automaton is a maximal sequence of states* $s_0 s_1 s_2 \ldots$ *such that for each* $i \geq 0$, *there exists* $a \in A$ *such that* $\delta(s_i, a) = s_{i+1}$.

Definition 2. *A cellular automaton is a set of* n *automata* $\mathcal{A}^i = \langle S^i, A^i, \delta^i, s_0^i \rangle$, $i \in \{1, \ldots, n\}$, *such that the sets of states are mutually disjoint, and the sets of actions may have common occurrences (corresponding to interactions).*

Example 1. Figure 1 shows a cellular automaton made of three automata. Each automaton $\mathcal{A}^i$ represents the ith bit of a binary counter (here modulo 8). The most significant bit is represented by the rightmost automaton. Interactions are named after the higher bit that changes during the interaction (e.g., s_1 corresponds to the setting of bit 1 and synchronizes $\mathcal{A}^1$ and $\mathcal{A}^0$, r_2 corresponds to the reset of bit 2 and synchronizes $\mathcal{A}^2$, $\mathcal{A}^1$ and $\mathcal{A}^0$). Each interaction involves either one (s_0), two (s_1) or three (s_2, r_2) automata.

We denote by $S = S^1 \times \ldots \times S^n$ the set of global states of a cellular automaton. A global state $g \in S$ is defined by the state of each automaton $\mathcal{A}^i$ from the cellular automaton. The state of the automaton $\mathcal{A}^i$ at global state g is denoted $g[i]$.

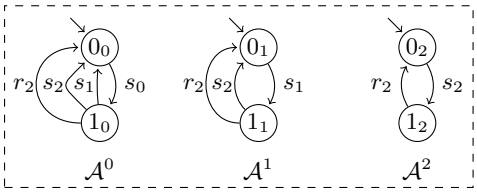


Fig. 1. Example of Cellular Automaton

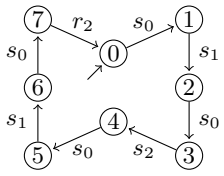


Fig. 2. Global Behavior

Definition 3. *An execution of a cellular automaton is a maximal sequence of global states $g_0 g_1 \ldots$ such that:*

- *g_0 is the tuple made of initial states: $\forall i \in \{1..n\}, g_0[i] = s_0^i$.*
- *For adjacent tuples g_j and g_{j+1} there is an action $a \in \cup_{i\in\{1..n\}} A^i$ such that for each $i \in \{1..n\}$, either $g_{j+1}[i] = \delta^i(g_j[i], a)$ or $a \notin A^i$ and $g_{j+1}[i] = g_j[i]$.*

It is easy to see that the projection of an execution into a single automaton is a prefix of an execution of that automaton.

We denote by $g \xrightarrow{a} g'$ if the action a can be executed from global state g and leads to global state g'. This notation is trivially extended to sequences of interactions, that is for $\sigma = a_1 a_2 \cdots a_k$ we denote by $g \xrightarrow{\sigma} g'$ if there exists global states $g_1, g_2, \cdots g_{k-1}$ such that $g \xrightarrow{a_1} g_1 \xrightarrow{a_2} g_2 \cdots g_{k-1} \xrightarrow{a_k} g'$. We denote by $\sigma|_{A^i}$, the sequence of interactions obtained by removing from σ all occurrences of interactions that are not in A^i.

The set of executions, i.e. global behavior, of the cellular automaton can be represented as a labeled transition system $\mathcal{A} = (S, A, T, g_0)$, where S is the set of global states, $A = \bigcup_{i\in\{1..n\}} A^i$ is the set of actions (or labels), $T \subseteq S \times A \times S$ is the set of valid transitions (as defined by Definition 3) and g_0 is the initial global state.

Example 2. The global behavior of the cellular automaton depicted in Figure 1 is shown in Figure 2. Any global state $g \in \{0, \ldots, 7\}$ denotes the tuple of local states $(g[2]g[1]g[0])$ obtained from the representation of g as a binary number.

Cellular automata are perhaps the simplest model to describe synchronizing systems. Nonetheless, this model is expressive enough to underlie higher-level frameworks with similar synchronization-based communication. In particular, we focus hereafter on the relation between cellular automata and the BIP framework [3], which will be used later in section 4 for concrete experiments. BIP (Behavior-Interaction-Priority) is a component-based framework which allows the construction of hierarchically structured component-based systems. In BIP, atomic components are characterized by their interface, that is, a set of ports (similar to action names) and their behavior, that is, an automaton with transitions labeled by ports. Components are composed by layered application of interactions and priorities. Interactions express synchronization constraints between ports of the composed components. An interaction is a set of ports, every one belonging to a different component, that has to be jointly executed. BIP provides (hierarchical) connectors as a mean to structure and express sets of interactions in a compact

manner. Finally, priorities are used in BIP to filter amongst the set of enabled interactions. Priorities provide an additional coordination mechanism to control the system evolution. A significant part of BIP systems can be structurally represented as cellular automata. That is, any BIP system without priorities can be equally represented as a cellular automaton by mapping BIP interactions into cellular automata interactions. Since a port may be involved in several interactions, BIP atomic components can be transformed into automata by duplicating transitions labeled by a port into a set of transitions labeled by the corresponding interactions.

2.2 The α-core protocol

The α-core protocol [15] was developed to schedule multiprocess interaction. It generalizes protocols for handshake communication between pairs of processes. For each multiprocess interaction, there is a dedicated coordinator on a separate process. To appreciate the difficulty of designing such a protocol, recall for example the fact that the language CSP of Hoare [9] included initially an asymmetric construct for synchronous communication; a process could choose between various incoming messages, but had to commit on a particular send. This constraint was useful for achieving a simple implementation. Otherwise, one needs to consider the situation in which a communication is possible between processes, but one of them may have performed an alternative choice. Later Hoare removed this constraint from CSP. The same constraint appears in the asymmetric communication construct of the programming language ADA. The Buckley and Silberschatz protocol [6] solves this problem for the case of synchronous communication between pairs of processes, where both sends and receives may have choices. Their protocol uses asynchronous message passing between the processes to implement the synchronous message passing construct. The α-core protocol solves the more general problem of synchronizing any number of processes, using only asynchronous message passing. Alternative solutions for this problem have been proposed, using managers [7, 1], a circulating token [12], or a randomized algorithm without managers [10]. Contrarily to other manager-based solutions, α-core does not need unbounded counters. The version presented below includes corrections from [11].

In α-core, the following messages are sent from a participant to a coordinator:

PARTICIPATE A participant is interested in a single particular interaction (hence it can commit on it), and notifies the related coordinator.

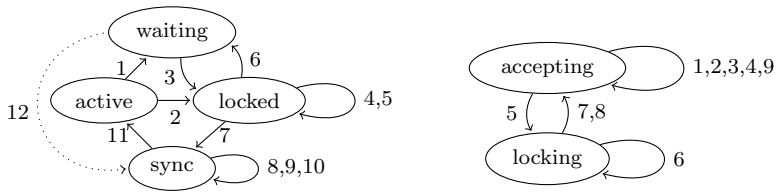
OFFER A participant is interested in one out of several potentially available interactions (a non-deterministic choice).

OK Sent as a response to a **LOCK** message from a coordinator (described below) to notify that the participant is willing to commit on the interaction.

REFUSE Notify the coordinator that the previous **OFFER** is not valid anymore. This message can respond to a **LOCK** message from the coordinator.

Messages from coordinators are as follows:

LOCK A message sent from a coordinator to a participant that has sent an **OFFER**, requesting the participant to commit on the interaction.



(a) Participant (b) Coordinator

Fig. 3. State machines

UNLOCK A message sent from a coordinator to a locked participant, indicating that the current interaction is canceled.
START Notifying a participant that it can start the interaction.
ACKREF Acknowledging a participant about the receipt of a **REFUSE** message.

Fig. 3(a) describes the extended state machine of a participant. Each participant process keeps some local variables and constants:

IS: a set of coordinators for the interactions the participant is interested in.
locks: a set of coordinators that have sent a pending **LOCK** message.
unlocks: a set of coordinators from which a pending **UNLOCK** message was received.
locker: the coordinator that is currently considered.
n: the number of **ACKREF** messages required to be received from coordinators until a new coordination can start.
α: the coordinators that asked for interactions and subsequently refused.

The actions according to the transitions are written as a pair $en \rightarrow action$, where *en* is the condition to execute the transition, which may include a test of the local variables, a message that arrives, or both of them (then the test should hold *and* the message must arrive). We denote the reception of a message **MSG** from process *p* by *p*?**MSG**. The action is a sequence of statements, executed when the condition holds. The statement *p*!**MSG** means "send message **MSG** to process *p*". In addition, each transition is enabled from some state, and upon execution changes the state according to the related extended finite state machine. The participant's transitions, according to the numbering of Fig. 3(a) are:

1. $|IS > 1| \rightarrow$ { foreach $p \in IS$ do p!**OFFER** }
2. $|IS = 1| \rightarrow$ { $locker$:=p, where $IS = \{p\}$; $locker$!**PARTICIPATE**; $locks$, $unlocks$:=$\emptyset$ }
3. p?**LOCK** $\rightarrow$ {$locker$:=p; $locks$, $unlocks$:=$\emptyset$; p!**OK** }
4. p?**LOCK** $\rightarrow$ {$locks$:=$locks \cup \{p\}$}
5. $locks \neq \emptyset \wedge p$?**UNLOCK** $\rightarrow$ {$locker$:=q for some $q \in locks$; q!**OK**; $locks$:=$locks \setminus \{q\}$; $unlocks$:=$unlocks \cup \{p\}$}
6. $locks = \emptyset \wedge p$?**UNLOCK** $\rightarrow$ { foreach $q \in unlocks \cup \{p\}$ do q!**OFFER**}
7. p?**START** $\rightarrow$ {α:=$IS \setminus (unlocks \cup \{locker\})$; foreach $q \in \alpha$ do q!**REFUSE**; $n := |\alpha|$; start participating in the joint action managed by $locker$}
8. p?**LOCK** $\rightarrow$ {}
9. p?**UNLOCK** $\rightarrow$ {}
10. p?**ACKREF** $\rightarrow$ {n:=$n-1$}
11. $n = 0 \rightarrow$ { Let *IS* be the new set of interactions required from the current state. }

For a coordinator, whose extended finite state machine appears in Fig. 3(b), we have the variables *waiting*, *locked*, *shared* and α, holding each a set of processes, and n is a counter for the number of processes that indicated their wish to participate in the interaction. The constant C holds the number of processes that need to participate in the interaction (called, the *cardinality* of the interaction), and the variable *current* is the participant the coordinator is trying to lock. The transitions, according to their numbering from Fig. 3(b) are as follows:

1. $n < C \wedge p?\textbf{OFFER} \rightarrow \{n{:=}n+1;\ shared{:=}\ shared \cup \{p\}\ \}$
2. $n < C \wedge p?\textbf{PARTICIPATE} \rightarrow \{\ n{:=}n+1;\ locked{:=}\ locked \cup \{p\}\ \}$
3. $p?\textbf{REFUSE} \rightarrow \{$ if $p \in shared$ then $n{:=}n-1$; $p!\textbf{ACKREF}$; $shared{:=}shared \setminus \{p\}\}$
4. $n = C \wedge shared = \emptyset \rightarrow \{$ foreach $q \in locked$ do $q!\textbf{START}$; $locked, shared{:=}\emptyset$; $n{:=}0\}$
5. $n = C \wedge shared \neq \emptyset \rightarrow \{current{:=}\ \min(shared);\ waiting{:=}shared \setminus \{current\}$; $current!\textbf{LOCK}\}$
6. $waiting \neq \emptyset \wedge p?\textbf{OK} \rightarrow \{locked{:=}locked \cup \{current\};\ current{:=}\min(waiting)$; $waiting{:=}waiting \setminus \{current\ \}$; $current!\textbf{LOCK}\}$
7. $waiting = \emptyset \wedge p?\textbf{OK} \rightarrow \{locked{:=}locked \cup \{current\}$; foreach q in $locked$ do $q!\textbf{START}$; $locked, waiting, shared{:=}\emptyset$; $n{:=}0\}$
8. $p?\textbf{REFUSE} \rightarrow \{\alpha{:=}(locked \cap shared) \cup \{current, p\}$; foreach $q \in \alpha \setminus \{p\}$ do $q!\textbf{UNLOCK}$; $p!\textbf{ACKREF}$; $shared{:=}shared \setminus \alpha$; $locked{:=}locked \setminus \alpha$; $n{:=}n-|\alpha|\}$
9. $p?\textbf{OK} \rightarrow \{\}$

We propose to characterize the correctness of the implementation by using execution trace equivalence. We assume that the network is reliable and that there is no message loss. We say that the interaction a occurs in a distributed execution of α-core whenever the transition 7 in the coordinator for a is executed. The correctness of α-core guarantees that the executions of the original cellular automaton and the executions of its implementation are the same.

3 Knowledge-Based Optimization

Synchronization algorithms such as α-core impose a lot of overhead in order to guarantee correct interaction. We want to utilize knowledge in order to reduce the overhead in coordination messages. Knowledge appears naturally in distributed systems, as it represents what a process knows from its obseravtions. Halpern and Moses [8] defined a logic to reason about knowledge. Van der Meyden [14] introduced knowledge with perfect recall. Knowledge has been applied to control distributed discrete event systems [16] and to implement priorities between multiparty interactions [2, 5]. However, the previous works assume a conflict resolution mechanism. We propose here a knowledge-based optimization of such a mechanism, which has not been done, at least to our knowledge. Based on [2], we construct a support automaton, which is a controller that either supports or blocks actions, based on precalculated knowledge. There are two kinds of controllers here. The first type is for each process of the system, and the second is per each α-core synchronizing process. The support automaton for a system automaton can reduce overhead by calculating when a component can actually commit to an interaction (offer a **PARTICIPATE** call to α-core), which requires less confirmation messages than simply declaring its participation (by the alternative **OFFER** call). In this case, the knowledge gathered in the precalculated stage can distinguish between the cases when

one has an alternative possibility of coordination or does not. While we could benefit from syntactically distinguishing between these cases based on the code of the system, the use of knowledge, and in particular, knowledge of perfect recall [14], can distinguish the cases where syntactically there can be alternative collaborations, but at this stage of the executions, the alternatives are not available.

Let $\langle \mathcal{A}^1, \mathcal{A}^2, ..., \mathcal{A}^n \rangle$ be a cellular automaton and $\mathcal{A} = (S, A, T, g_0)$ its associated global behavior as defined in section 2.1.

3.1 Knowledge for participants

Let $\mathcal{A}^i = \langle S^i, A^i, \delta^i, s_0^i \rangle$ be a participant. As in [14, 2], we define the *knowledge with perfect recall* of this participant as the facts it can infer based on its local history. Recall that we denote by $\sigma|_{A^i}$ the sequence of interactions obtained by removing from the sequence σ all occurrences of interactions that are not in A^i.

Definition 4 (Indistinguishability of execution sequences for $\mathcal{A}^i$). *Two sequences of interactions σ and σ' are indistinguishable by $\mathcal{A}^i$, denoted $\sigma \equiv_i \sigma'$, iff $\sigma|_{A^i} = \sigma'|_{A^i}$.*

Definition 5 (Knowledge with perfect recall). *Let σ be a sequence of interactions , $\mathcal{A}^i$ be a participant and φ be a predicate. After executing σ, $\mathcal{A}^i$ knows φ if φ holds after any execution σ' that is indistinguishable by $\mathcal{A}^i$ from σ. Formally, $\mathcal{A}^i$ knows φ after executing σ if $\varphi(g)$ holds for every state g in $\{g \in S | \exists \sigma',\ \sigma' \equiv_i \sigma \wedge g_0 \xrightarrow{\sigma'} g\}$.*

In order to compute the knowledge with perfect recall of the participant $\mathcal{A}^i$, we build its support automaton $\mathcal{K}_i$ as in [2]. The support automaton $\mathcal{K}_i$ will follow the execution of observable interactions for $\mathcal{A}^i$, that is, all interactions in A^i. The remaining interactions in $U^i = A \setminus A^i$ are not observable by $\mathcal{K}_i$. Informally, the state reached in $\mathcal{K}_i$ after any sequence $\sigma \in A^*$ summarizes all the global states that can be reached in $\mathcal{A}$ after any sequence $\sigma' \in A^*$ such that σ and σ' are indistinguishable by $\mathcal{A}^i$. Formally, $\mathcal{K}_i$ is defined as the deterministic automaton $\langle S_i, A^i, \delta_i, s_{0i} \rangle$ where:

- The set of states $S_i = 2^S$ correspond to subsets of the global states S.
- The transition function δ_i is defined as $\delta_i(s,a) = \{g' \mid \exists g \in s,\ \exists \sigma \in A^*,\ g \xrightarrow{\sigma} g' \text{ and } \sigma_{|A^i} = a\}$. Informally, for any state s, its successor s' through interaction a contains the set of global states g' that are reached in $\mathcal{A}$ from global states g in s by executing any sequence of unobservable interactions and exactly one a.
- The initial state $s_{0i} = \{g \in S | \exists \sigma \in (U^i)^*,\ g_0 \xrightarrow{\sigma} g\}$. Informally, s_{0i} contains all global states reachable in $\mathcal{A}$ by executing any sequence of unobservable interactions starting from the initial global state g_0.

Example 3. We illustrate the construction above on each automaton of the binary counter example from Figure 1. For $\mathcal{A}^0$, we have $\mathcal{K}_0 = \mathcal{A}$, since $\mathcal{A}^0$ observes all interactions. The support automata obtained for $\mathcal{A}^1$ and $\mathcal{A}^2$ are depicted in Figure 4. Even if by construction, the state $\emptyset$ might be reachable, we do not consider it. Note that $\mathcal{K}_2$ is the same as $\mathcal{A}^2$ up to the name of the states.

The support automaton is used to reduce coordination overhead in α-core as follows. For every $\mathcal{A}^i$, the support automaton $\mathcal{K}_i = \langle S_i, A^i, \delta_i, s_{0i} \rangle$ is *embedded* in the corresponding participant behavior. For our application, there is no need to explicitly keep track of the set of global states corresponding to the states of $\mathcal{K}_i$. Therefore, once the automaton $\mathcal{K}_i$ is constructed, states in S_i can be replaced by elements of any arbitrary finite domain. The participant uses one extra local variable *s* to record the state of the support automaton. This variable is initialized as s_{0i}. Then, this variable is updated when the participant executes an interaction (transition 7) and is used to filter the set *IS* before entering the *active* state (transition 11). The original transitions 7 and 11 are therefore modified into transition $7'$ and $11'$ as follows:

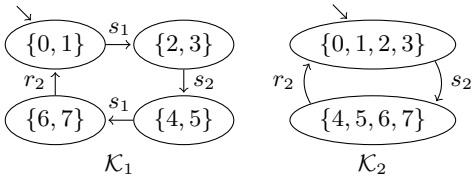


Fig. 4. Support automata for participants $\mathcal{A}^1$ and $\mathcal{A}^2$ of Figure 1.

$7'$. *p*?**START** $\rightarrow$ {α:=*IS* \ *unlocks* \ {*locker*}; foreach $q \in \alpha$ do *q*!**REFUSE**; $n := |\alpha|$; start participating in the joint action *a* managed by *locker*; $\boxed{s := \delta_i(s,a)}$ }

$11'$. $n = 0 \rightarrow$ { Let *IS* be the required interactions; $\boxed{IS := IS \cap \{a \in A | \delta_i(s,a) \neq \emptyset\}}$ }

That is, the optimization restricts the sending of offer messages for interactions that are enabled according to the support automaton. Clearly, this restricts the number of exchanged messages. Moreover, in cases where no conflict exists in the filtered behavior (such as in the binary counter example, the size of the *IS* set is always reduced to 1), **OFFER** messages are replaced by **PARTICIPATE** messages, thus removing the need for further locking by coordinators.

Example 4. As an example, from the state 1 in $\mathcal{A}^1$, two interactions (s_2 and r_2) are possible. In $\mathcal{K}_1$ this state is split in two states that separate the case where s_2 is possible from the case where r_2 is possible.

Proposition 1. *The executions of the distributed implementation with knowledge-optimized participants are the same as the executions of the original automaton.* [1]

3.2 Knowledge for coordinators

Coordinators of the α-core can also gain information about the global context by recording the offers received from different components. That is, new offers are issued by participants only at the initial state and after every successful participation in an interaction. Therefore, offer reception provides to coordinators some (indirect and definitely incomplete) information about the evolution of the system. Nonetheless, this information can be exploited in order to avoid some useless coordination of the α-core protocol. For example, a coordinator may detect that some offers are obsolete (their locking will

[1] For space reasons, proofs are not provided here but in the technical report [4].

always be refused) or stable (on the contrary, their locking will always be accepted by the corresponding participant).

The construction of the support automata for coordinators is a bit more intricate than for participants. We want to benefit from the same approach by constructing a controller that is based on the precalculation of knowledge. However, such calculation can be quite intricate if it takes into account the structure of the α-core algorithm. The complication is due to the above mentioned difference in observation, that is, offers *vs.* interactions. The starting point of the construction is the global behavior $\mathcal{A}$. Clearly, $\mathcal{A}$ does not mention explicitly the sending/reception of offers by participants/coordinators. But, communication of offers can still be inferred from $\mathcal{A}$ knowing the behavior of the α-core protocol. We present hereafter a systematic construction that allows to progressively *refine* $\mathcal{A}$ such that to make visible (relevant) offers communication for any selected coordinator. The construction involves (1) offer generation, as response to execution of (conflicting) interaction, (2) asynchronous offer reception by the coordinator, (3) determinization into a support automaton to be used by the coordinator.

Let a be a fixed interaction. We construct the support automaton $\mathcal{K}_a$ by applying a sequence of transformations on the global behavior $\mathcal{A} = (S, A, T, g_0)$ as follows:

Offer generation: We construct the labelled transition system $\mathcal{A}' = (S, 2^{\{a,1..n\}}, T', g_0)$ by replacing labels of each transition, so that they contain information about the offers concerning a. For an interaction a' and a global state g', we denote by $I(a', g') = \{i \mid a, a' \in A^i \wedge \exists s'' \in S^i,\ \delta^i(g'[i], a) = s''\}$ the set of indices of automata that can participate in a after executing a'. Intuitively, this corresponds to set of offers that the coordinator for a will receive after execution of a'. We relabel the transition $g \xrightarrow{a'} g'$ by $g \xrightarrow{\{a\}\cup I(a,g')} g'$ if $a' = a$ and by $g \xrightarrow{I(a',g')} g'$ otherwise. It might be the case that some transitions have $\emptyset$ as label after this step, which means that they have no observable effect on the a-coordinator and thus are unobservable.

Asynchronous offer reception: We construct the labelled transition system $\mathcal{A}'' = (S'', \{a, \emptyset, 1, ..., n\}, T'', g_0'')$ obtained by breaking transitions in $\mathcal{A}'$ such that there is at most one action (either a or offer reception i) per transition. Formally, we take $S'' = S \times \{0,1,2\}^n$, that is, a state of $\mathcal{A}''$ is defined by a global state g of the cellular automaton and a vector v of n integers in $\{0,1,2\}$. For any participant i, the value v_i gives the number of *pending* offers, that is, potentially sent by i and not yet received by the coordinator. Given the specific behavior of the α-core, the number of pending offers is always between 0 and 2. For a given set of indices $I \subseteq \{1, ..., n\}$, we denote by $\mathbb{1}_I$ the characteristic vector of I that is 1 if $i \in I$ and 0 otherwise. We define the initial state $g_0'' = (g_0, \mathbb{1}_{I(a,g_0)})$. Transitions in T'' are constructed from the following rules, where I denotes an arbitrary index set:

$$\frac{g \xrightarrow{\{a\}\cup I} g' \in T'}{(g,0) \xrightarrow{a} (g', \mathbb{1}_I)} \qquad \frac{g \xrightarrow{I} g' \in T' \quad \forall i \in I,\ v_i \leq 1}{(g,v) \xrightarrow{\emptyset} (g', v + \mathbb{1}_I)} \qquad \frac{v_i > 0}{(g,v) \xrightarrow{i} (g, v - \mathbb{1}_{\{i\}})}$$

Projection and determinization: Finally, we construct the support automaton $\mathcal{K}_a = (S_a, \{a, 1, ..., n\}, \delta_a, g_{a0})$ as the deterministic automaton constructed from $\mathcal{A}''$ by eliminating $\emptyset$ actions which are unobservable. The construction is essentially the same as the one introduced in section 3.1 for participants and is not repeated here.

The previous construction guarantees that whenever an offer from participant $\mathcal{A}^i$ is received by the coordinator, the action i is possible from the current state of the support automata. This is stated in Lemma 1.

Lemma 1. *For any distributed execution σ, its restriction $\sigma|_{\{a,1,..,n\}}$ to actions observable by $\mathcal{K}_a$ is the trace of an execution of $\mathcal{K}_a$.*

Example 5. In Figure 5, we present the different steps leading to the construction of $\mathcal{K}_{s_1}$. To obtain the automaton $\mathcal{A}'$, we relabel the transitions in $\mathcal{A}$. For instance, the transition $0 \xrightarrow{s_0} 1$ in $\mathcal{A}$ brings $\mathcal{A}^0$ in a state where it can take part in s_1. From the s_1-coordinator point of view, this corresponds to receiving an offer from $\mathcal{A}^0$. Thus, the transition is relabelled by $\{0\}$ in $\mathcal{A}'$. In the non-deterministic automaton $\mathcal{A}''$, each state is labelled by a couple (g,v), where g is a global state from $\mathcal{A}$, and $v = v_0v_1$ is a vector where v_i is the number of offers to receive from $\mathcal{A}^i$. The dotted transitions correspond to unobservable actions. Note that we depicted only the half of $\mathcal{A}''$, the other half (corresponding to states $3,4,5,6$) shows the same pattern between states $(3,10)$ to $(6,00)$ as between states $(7,10)$ and $(2,00)$. Finally, the determinized and minimized version of $\mathcal{A}''$ is the automaton $\mathcal{K}_{s_1}$. It states that between two executions of s_1, two offers from $\mathcal{A}^0$ and one offer from $\mathcal{A}^1$ are to be received, in any order.

The coordinator for interaction a observes the offers sent from all participants in a and computes the set of known stable and obsolete components (or offers). We say that a component (offer) $\mathcal{A}^i$ is *stable* at state s in $\mathcal{K}_a$ iff for all paths starting at s, a transition labelled by i cannot be reached without going through a transition labelled by a. Whenever an offer from $\mathcal{A}^i$ is stable, the coordinator *knows* that $\mathcal{A}^i$ can not send a new offer until the interaction a takes place. More precisely, $\mathcal{A}^i$ can only participate in a and the information received from $\mathcal{A}^i$ is up to date. If stable, $\mathcal{A}^i$ can be considered as if it were locked. In a dual manner, we say that a component (offer) $\mathcal{A}^i$ is *obsolete* at state s in $\mathcal{K}_a$ iff for all paths starting at s, a transition labelled by a cannot by reached without going through a transition labelled by i. In this case, the coordinator *knows* that it has to receive a new offer from $\mathcal{A}^i$ before starting the interaction. This information can be used to avoid tentative executions based on obsolete offers.

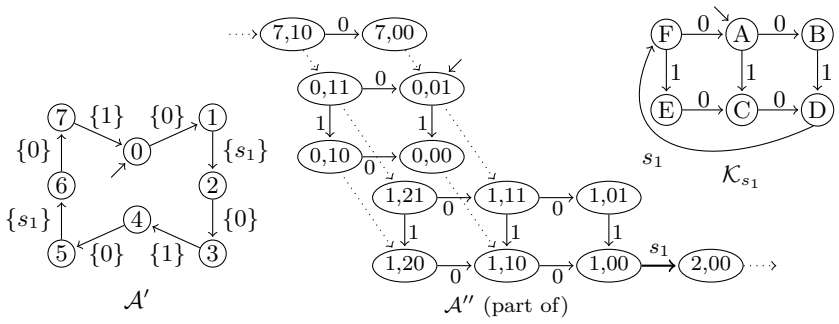


Fig. 5. Construction of the support automaton for the coordinator of s_1.

Example 6. Let us consider the support automaton for the coordinator of s_1 in the binary counter. At state *C*, the coordinator may have received two offers from $\mathcal{A}^0$ and $\mathcal{A}^1$ and the default behavior is to attempt execution for interaction s_1. However, the offer from $\mathcal{A}^0$ is obsolete. Using the support automaton, the coordinator can therefore detect this situation and silently remove that offer, which avoids the execution attempt. At state *D*, both $\mathcal{A}^0$ and $\mathcal{A}^1$ are stable and there is no need to lock them before executing s_1.

The above optimizations are implemented as follows. The coordinator for a follows the automaton $\mathcal{K}_a$ when receiving offers[2] (transitions 1 and 2) and executing a (transitions 4 or 7, from Figure 3(b)). Formally, the coordinator uses an extra variable s tracking the state of the support automaton. The transitions 1,2,4 and 7 of the coordinator are modified into transitions 1',2',4' and 7' as follows:

1′. $n < C \wedge p?$**OFFER** $\rightarrow$ { n:=$n+1$; *shared*:= *shared* $\cup \{p\}$; $\boxed{s := \delta_a(s,p);\ update()}$ }

2′. $n < C \wedge p?$**PARTICIPATE** $\rightarrow$ { n:=$n+1$; *locked*:= *locked* $\cup \{p\}$; $\boxed{s := \delta_a(s,p);\ update()}$ }

4′. $n = C \wedge shared = \emptyset \rightarrow$ { foreach $q \in$ *locked* do q!**START**; *locked*, *shared*:=$\emptyset$; n:=0; $\boxed{s := \delta_a(s,a)}$ }

7′. *waiting* $= \emptyset \wedge p?$**OK** $\rightarrow$ {*locked*:=*locked* $\cup$ {*current*}; foreach q in *locked* do q!**START**; *locked*, *waiting*, *shared*:=$\emptyset$; n:=0; $\boxed{s := \delta_a(s,a);}$ }

The *update* function above is used to modify the *shared* and *locked* sets, given the current support automaton state s as follows:

foreach $p \in$ *shared*

if $p \in stable_a(s)$ { *shared* := *shared* $\setminus$ { p }; *locked* := *locked* $\cup$ { p } }

if $p \in obsolete_a(s)$ { *shared* := *shared* $\setminus$ {p}; n = n -1; p!**LOCK**; p!**UNLOCK** }

Since a component can now be considered as locked even if it sent an **OFFER** message, it may receive a **START** message while waiting to be locked. Therefore, we add a transition 12 from the waiting to the sync state, as depicted in Figure 3(a). We also modify transition 7 into transition 7′ as follows:

7′. $p?$**START** $\rightarrow$ {α:=*IS* $\setminus$ *unlocks* $\setminus$ {$\boxed{p}$}; foreach $q \in \alpha$ do q!**REFUSE**; $n := |\alpha|$; start participating in the joint action managed by $\boxed{\text{p}}$}

12. $p?$**START** $\rightarrow$ {α:=*IS* $\setminus$ *unlocks* $\setminus$ {p}; foreach $q \in \alpha$ do q!**REFUSE**; $n := |\alpha|$; start participating in the joint action managed by p}

Proposition 2. *The executions of the distributed implementation with knowledge-optimized coordinators are included in the executions of the original automaton.*

3.3 Combining knowledge for participants and coordinators

Optimization for participants and coordinator can be combined. In this case, the construction of the support automata for coordinators has to be done on the system obtained using the support automata for participants. In particular, the relabelling step depends on the actual offers sent by participants and thus on their support automata.

[2] Here we consider only *new* offers that we need to distinguish from offers sent when participant executes transition 6. This can be done by using a new message name for offers that are re-sent.

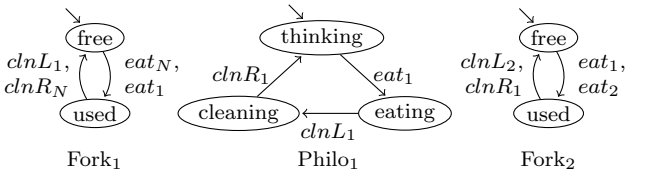


Fig. 6. A fragment of the dining philosophers example

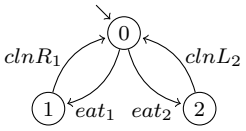


Fig. 7. Support automaton for participant Fork$_2$

4 Experimental Results

We present experimental results for computing and using the support automata for participants as presented in Section 3.1.

Examples The first example presented in Figure 6 is a variation of the classical dining philosophers problem. Each philosopher Philo$_i$ may eat during the interaction eat_i involving its two neighbor forks. Then Philo$_i$ clean first its left fork, then its right fork through interactions $clnL_i$ and $clnR_i$ respectively. We denote philo$N$ an instance with N philosophers.

The second example is called Master/Slave. We assume a set of N masters and M slaves. Each master wants to perform a task for which it needs two slaves that it can chose amongst a pool of size K. We denote msNMK such an instance. If the slave j is in the pool of the master i, then the interaction acq^i_j allows master i to acquire slave j, which brings the slave in state i so that it remembers that i acquired it. On completion on the task, the master i releases simultaneously the two acquired slaves j_1 and j_2 through the $rel^i_{j_1,j_2}$ interaction. Figure 8 shows respectively, the behavior of masters and slaves.

The third example models a transmission protocol that propagates values amongst a chain of memories. At every time, each memory node stores a single value. A fragment of this example is shown in Figure 9. The rule is to propagate (copy) the new value (from the left) only if the memory on the right has already copied the local value. Propagation steps are implemented as ternary interactions denoted by mv_{i,v_1,v_2}, which correspond to the case where memory i changes its value from v_1 to v_2. As an example, the interaction $mv_{i,1,0}$ in the Figure 9 changes the value in Node$_i$ from 1 to 0 if Node$_{i+1}$ already changed its value to 1 and the next value (in Node$_{i-1}$) is 0. For our experiment, the memories form a ring, thus the sequence of values seen by each memory depends only on the initial state of the system. Note that propagation is enabled at places where

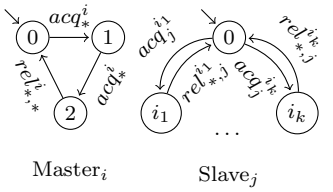


Fig. 8. Master Slave example

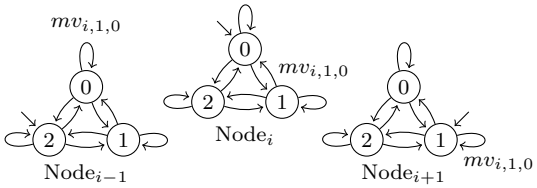


Fig. 9. Three consecutive nodes of the transmission protocol.

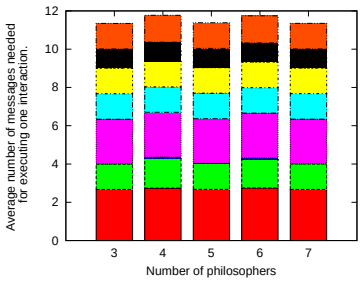


Fig. 10. Dining philosophers: messages per interaction, standard version.

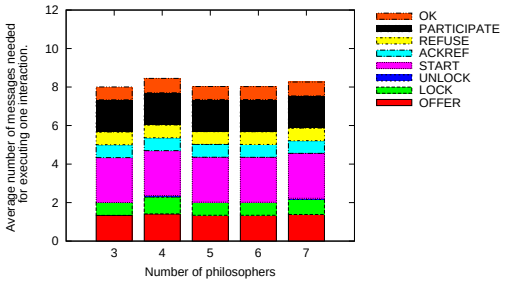


Fig. 11. Dining philosophers: messages per interaction, optimized version.

the ring contains two consecutive nodes holding the same value. We denote by tp$N$ (resp. tpN') an example with N nodes and one (resp. two) enabled propagations.

Building support automata for participants. We implemented the support automaton computation for each participant by using analysis tools of the BIP framework. In Table 4, we present the results of this analysis by giving the average number of states in the original automata and in the support automata. This gives an indication on the size needed to store the knowledge, and the memory needed for execution of the support automata. For the philo$N$ instances, the support automaton of philosophers is the same as the original automaton. For the forks, there is only one additional state, as shown in Figure 7. The added state allows to distinguish who acquired the fork (left or right) and to send only one offer accordingly, thus avoiding unneeded conflict resolution.

		Average number of states		Number of interactions during 60s	
Name	Components	in $\mathcal{A}^I$	in $\mathcal{K}_i$	Standard	Optimized
philo3	6	2.5	3	1129	2251
philo4	8	2.5	3	1811	2499
philo5	10	2.5	3	2261	4448
philo6	12	2.5	3	2624	4542
philo7	14	2.5	3	3093	4603
ms232	5	2.6	3	1491	1504
ms233	5	3	4.6	1128	1129
ms342	7	2.7	3.1	642	1885
ms343	7	3.1	4.9	1278	1265
ms344	7	3.6	7	1256	1251
tp3	3	3	6	750	1499
tp6	6	3	15	750	1500
tp6’	6	3	16	1498	1557
tp9	9	3	24	750	1509
tp9’	9	3	28	1497	3725
tp12	12	3	33	749	1513

Table 1. Results: average size of original and support automaton and performance of the obtained implementation, for each test instance.

In the Master/Slave example, the automaton describing a master is very generic. The corresponding support automaton contains all the possible sequences for acquiring two slaves and then releasing them. In particular, after having acquired two slaves, there is only one possible release interaction, thus only one offer is sent. Finally, in the transmission protocol example, the size of each support automaton is much larger since it depends on the number of nodes in the chain, that is on the sequence of values seen by each node. If two propagations are possible, then the size of the support automaton is slightly increased, since the two propagations may conflict.

Performance of distributed implementation. Using the BIP component framework, we built a transformation that replaces multiparty interactions by the α-core protocol. We obtain a distributed BIP model representing participants and coordinators communicating through asynchronous message passing. From this model, we generate a set of C++ programs communicating through Unix sockets. We ran the obtained code for both standard α-core and knowledge-optimized α-core on a UltraSparcT1 allowing parallel execution of 24 processes. In Table 4, we provide the number of interactions executed during 60 seconds of execution (not including initialization) for both standard and optimized version of each test instance. On the dining philosopher instances, the optimized version is up to twice faster than the standard version. On the Master/Slave instances, except for one, the performance is the same for both versions. On the transmission protocol instances, we have a speedup of at least two, except for the tp6′ example.

In order to evaluate the distributed execution of standard *vs.* optimized versions, we compare the average number of messages needed to perform an interaction for the three examples. For the dining philosophers, these average numbers are shown in Figures 10 and 11. We can observe a reduction of approximatively 25%, mainly because some **OFFER** messages from the fork participants are transformed in **PARTICIPATE** messages. In turn, this reduces the number of participants to lock, and thus the number of messages. For the Master/Slave, the average number of messages needed to complete one interaction for standard and optimized α-core are shown in Figures 12 and 13. Here the number of conflicts depends on the size of the pool of slaves assigned to each master. Since there are many conflicts, the number of offers sent to execute an interaction is quite big. Recall that on this example, performance of both versions is comparable. However, the number of exchanged message is smaller in the optimized version, because less offers are sent. For the transmission protocol, the average number of messages exchanged to execute one interaction for standard and optimized executions is shown in Figures 14 and 15. For the non-primed versions, since there is no dynamic conflict, each participant sends only **PARTICIPATE** messages and each coordinator can directly answer a **START** message. This reduces drastically the number of exchanged messages (6 per interaction, since they are ternary interactions). For the primed version, in some cases a node may participate in two interactions and thus send two **OFFER** messages, which is still much less than in the original version.

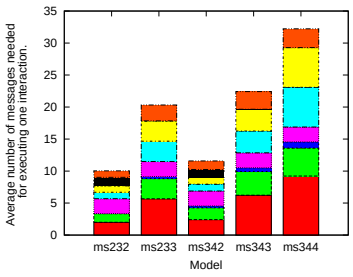


Fig. 12. Master/Slave: messages per interaction, standard version.

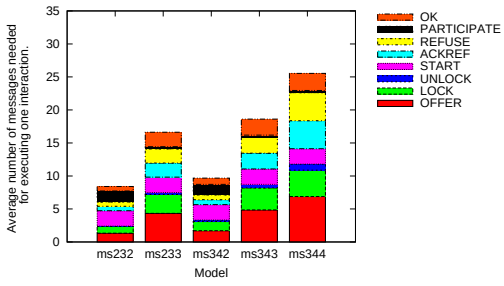


Fig. 13. Master/Slave: messages per interaction, optimized version.

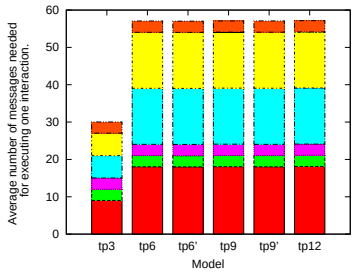


Fig. 14. Transmission protocol: messages per interaction, standard version.

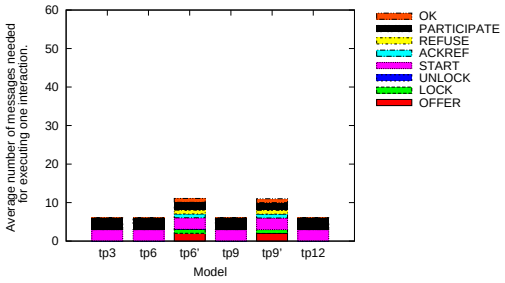


Fig. 15. Transmission protocol: messages per interaction, optimized version.

5 Discussion

An architecture for component-based system can provide a very powerful tool for distributed software development. It assumes some underlying mechanism that provides support for the components to interact and to choose from several alternative actions. It is highly beneficial to develop code at this level, rather than to consider the lower level architecture that uses message passing, or shared variables. On the other hand, obtaining this level of abstraction is expensive: the overhead needed to allow both multiparty interaction and non-deterministic choice requires some nontrivial amount of lower level message exchange.

In this paper we looked at a technique to reduce the overhead needed for supporting high level architecture for component-based systems, such as the BIP systems. Observing a popular algorithm for interaction coordination, the α-core protocol, we remarked that additional information about the amount of overhead makes a lot of difference. The coordination protocol distinguishes the case where there is no non-deterministic choice; then, there are fewer messages sent, as an intent to participate in an interaction is a committed intention. It is often not known in advance how many conflicting choices there are: syntactically, there can be several, but at runtime, there are quite fewer cases available (enabled) at each particular instance. Our method is based on performing a preliminary model checking analysis of the system for detecting such situations. When we find that the local situation admits no non-deterministic choice at any possible global situation, we can employ the more efficient case of committing to an interaction.

This analysis is based on the knowledge of a process, regarding all the possible global states consistent with its local situation. We apply this optimization in two cases: locally at the process level, where the knowledge of the process may be used to transfer a seamingly non-deterministic case into a committing case, and at the level of a process of the coordination algorithm. The latter case is very powerful, as a coordinator process has, to some extent, a more global view, having received requests from different processes. Experiments show that rather than using simple memoryless knowledge, we are required to use history-based knowledge. The reason is that it is the cases where different instances of non-deterministic choice during runtime, rather than a history independent case, are the interesting ones. This can be explained intuitively by the fact

that the history independent case actually hides a coding error, where not committing to an interaction although there are no alternatives should have been replaced by a commitment to the single possible interaction.

We performed experiments on three different examples. Our experiments show a considerable improvement in the number of messages needed to be exchanged. It is important to note that due to the use of history-based knowledge, additional memory is needed to encode the possible histories. In the worst case, the amount of added memory is quite nontrivial, exponential in the size of the system, for each process. However, our experiments show a much better and balanced memory consumption. We intend to conduct further experiments and to apply the knowledge-based technique for reducing message passing in a more aggressive way.

References

1. R. Bagrodia. Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Transactions on Software Engineering (TSE)*, 15(9):1053–1065, 1989.
2. A. Basu, S. Bensalem, D. Peled, and J. Sifakis. Priority scheduling of distributed systems based on model checking. *Formal Methods in System Design*, 39:229–245, 2011.
3. A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *Software Engineering and Formal Methods (SEFM)*, pages 3–12, 2006.
4. S. Bensalem, M. Bozga, J. Quilbeuf, and D. Peled. Knowledge based transactional behavior. Technical Report TR-2012-20, VERIMAG, 2012.
5. S. Bensalem, M. Bozga, J. Quilbeuf, and J. Sifakis. Knowledge-based distributed conflict resolution for multiparty interactions and priorities. In *FMOODS/FORTE*, pages 118–134, 2012.
6. G. N. Buckley and Abraham Silberschatz. An effective implementation for the generalized input-output construct of CSP. *ACM Trans. Program. Lang. Syst.*, 5:223–235, 1983.
7. K. M. Chandy and J. Misra. *Parallel program design: a foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.
8. J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *J. ACM*, 37:549–587, 1990.
9. C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21:666–677, 1978.
10. Y.-J. Joung and S. A. Smolka. Strong interaction fairness via randomization. *IEEE Trans. Parallel Distrib. Syst.*, 9(2):137–149, 1998.
11. G. Katz and D. Peled. Code mutation in verification and automatic code correction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 6015 of *LNCS*, pages 435–450. 2010.
12. D. Kumar. An implementation of n-party synchronization using tokens. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 320–327. IEEE, 1990.
13. D. Lehmann and M. O. Rabin. On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem. In *Principles of Programming Languages (POPL)*, 1981.
14. R. V. Der Meyden. Common knowledge and update in finite environments. *Information and Computation*, 140:115–157, 1997.
15. J. A. Pérez, R. Corchuelo, and M. Toro. An order-based algorithm for implementing multiparty synchronization. *Concurrency and Computation: Practice and Experience*, 16(12):1173–1206, 2004.
16. S.L. Ricker and K. Rudie. Know means no: Incorporating knowledge into discrete-event control systems. *IEEE Trans. on Automatic Control*, 45(9):1656–1668, 2000.