Optimized Distributed Implementation of Multiparty Interactions with Observation

Saddek Bensalem, Marius Bozga, Jean Quilbeuf, Joseph Sifakis

HAL Id: hal-00878242

https://hal.science/hal-00878242

Submitted on 29 Oct 2013

Optimized Distributed Implementation of Multiparty Interactions with Observation [*]

Saddek Bensalem[1]   Marius Bozga[1]   Jean Quilbeuf[1]   Joseph Sifakis[1,2]

[1] UJF-Grenoble 1 / CNRS VERIMAG UMR 5104, Grenoble, F-38041, France
[2] RISD Laboratory, EPFL, Lausanne, CH-1015, Switzerland
{bensalem,bozga,quilbeuf,sifakis}@imag.fr

Abstract

Using high level coordination primitives allows enhanced expressiveness of component-based frameworks to cope with the inherent complexity of present-day systems designs. Nonetheless, their distributed implementation raises multiple issues, regarding both the correctness and the runtime performance of the final implementation. We propose a novel approach for distributed implementation of multiparty interactions subject to scheduling constraints expressed by priorities. We rely on new composition operators and semantics that combine multiparty interactions with observation. We show that this model provides a natural encoding for priorities and moreover, can be used as an intermediate step towards provably correct and optimized distributed implementations.

Categories and Subject Descriptors   F.1.1 [*Theory of Computation*]: COMPUTATION BY ABSTRACT DEVICES; C.5 [*Computer Systems Organization*]: COMPUTER SYSTEM IMPLEMENTATION;   C.2.4 [*Coputer Systems Organization*]: COMPUTER-COMMUNICATION NETWORKS

Keywords   multiparty interaction, priority, observation, conflict resolution, distributed systems

1. Introduction

Correct design and implementation of computing systems has been an active research topic over the past three decades. This problem is significantly more challenging in the context of distributed systems due to a number of factors such as non-determinism, asynchronous communication, race conditions, fault occurrences, etc. Model-based development of such applications aims to ensure correctness through the usage of explicit model transformations.

In this paper, we focus on distributed implementation for models defined using the BIP framework [3]. BIP (Behavior, Interaction, Priority) is based on a semantic model encompassing composition of heterogeneous components. The *behavior* of components is described as an automaton extended by arbitrary data and associated functions written in C. BIP uses an expressive set of composition operators for obtaining composite components from a set of components. The operators are parameterized by a set of *multiparty interactions* between the composed components and by *priorities*, used to specify different scheduling mechanisms between interactions[1].

Transforming a BIP model into a distributed implementation consists in addressing three fundamental issues:

1. *Enabling concurrency.*   Components and interactions should be able to run concurrently while respecting the semantics of the high-level model.

2. *Conflict resolution.*   Interactions that share a common component can potentially conflict with each other.

3. *Enforcing priorities.*   When two interactions can execute simultaneously, the one with higher priority must be executed.

We developed a general method based on source-to-source transformations of BIP models with multiparty interactions leading to distributed models that can be directly implemented [8, 9]. This method has been later extended to handle priorities [10] and optimized by exploiting knowledge [6]. The target model consists of components representing processes and Send/Receive interactions representing asynchronous message passing. Correct coordination is

---

[1] Although our focus is on BIP, all results in this paper can be applied to any model that is specified in terms of a set of components synchronized by interactions with priorities.

2013/10/29

achieved through additional components implementing conflict resolution and enforcing priorities between interactions.

In particular, the conflict resolution issue has been addressed by incorporating solutions to the *committee coordination problem* [12] for implementing multiparty interactions. Bagrodia [1] proposes solutions to this problem with different degrees of parallelism. The most distributed solution is based on the drinking philosophers problem [11], and has inspired the approaches of Pérez et al. [18] and Parrow et al. [17]. In the context of BIP, a transformation addressing all the three challenges through employing *centralized scheduler* is proposed in [2]. Moreover, in [8], we propose transformations that address both the concurrency issue by breaking the atomicity of interactions and the conflict resolution issue by embedding a solution to the committee coordination problem in a distributed fashion.

Distributed implementation of priorities is usually considered as a separate issue, and solved using completely different approaches. For example, in [10], priorities are eliminated by adding explicit scheduler components and more multiparty interactions. This transformation leads to potentially more complex models, having definitely more interactions and conflicts than the original one. In [4, 5, 7], the focus is on reducing the overhead for implementing priorities by exploiting knowledge. Yet, these approaches make the implicit assumption that multiparty interactions are executed atomically and do not consider conflict resolution. In a similar line of work, [6] aims at detecting false conflicts, that is, statically detected but never occurring during execution. However, this method still relies on conflict resolution protocols, at least for states where no false conflicts exist.

In this paper, we propose a combined implementation of the two coordination mechanisms, that is, multiparty interactions and priorities. We propose an appropriate intermediate model and transformations towards fully distributed models dealing adequately with both of them. The contribution is twofold:

1. First, we introduce an alternative observation-based semantic model for BIP. We show that this model is general enough to encompass priorities and multiparty interactions and, moreover, to capture knowledge-based optimization as in [6]. Observation-based semantics reveals two types of conflicts occuring between interactions, that can be handled using different conflict resolution mechanisms (see below).

2. Second, this model is used in an intermediate step of a transformation leading to a distributed implementation. We show that *observation conflicts*, that usually follow from encoding of priorities, can be dealt more efficiently than *structural conflicts*, due to sharing of components between multiparty interactions. We extend the counter-based conflict resolution protocols of Bagrodia in order to handle these types of conflicts. These extensions have

been fully implemented. We report some preliminary results on benchmarks.

The paper is organized as follows. Section 2 introduces the main concepts of the BIP framework together with the alternative observation-based composition semantics. Section 3 recalls the principles for distributed implementation of BIP models, focusing on conflict resolution by using counter-based protocols. Section 4 defines the method for distributed implementation of BIP models with observation and in particular, the necessary adaptation of the conflict resolution protocols. Experiments are reported in Section 5. Section 6 provides conclusions and perspectives for future work.

2. Semantic Models of BIP

In this section, we present BIP[3], a component framework for building systems from a set of atomic components by using two types of composition operators: Interaction and Priority. We then present an alternative model based on Observation that can express Priority. Finally we present a transformation from a component with Observation into a equivalent component with only Interaction.

Atomic Components. An *atomic component* B is a labelled transition system represented by a tuple (Q, P, T) where Q is a set of *control locations* or *states*, P is a set of *communication ports* and $T \subseteq Q \times P \times Q$ is a set of *transitions*.

Interactions. In order to compose a set of n atomic components $\{B_i = (Q_i, P_i, T_i)\}_{i=1,n}$, we assume that their respective sets of control locations and ports are pairwise disjoint; i.e., for any two $i \neq j$ in $\{1..n\}$, we require that $Q_i \cap Q_j = \emptyset$ and $P_i \cap P_j = \emptyset$. We define the global set $P \stackrel{def}{=} \bigcup_{i=1}^{n} P_i$ of ports. An *interaction* a is a set of ports such that a contains at most one port from each atomic component. We take $a = \{p_i\}_{i \in I}$ with $I \subseteq \{1..n\}$ and $p_i \in P_i$. If a is an interaction, we denote by $support(a)$ the set of atomic components that participate in a. This notation is extended to sets of interactions γ, that is, $support(\gamma) \stackrel{def}{=} \bigcup_{a \in \gamma} support(a)$.

Priorities. Given a set γ of interactions, we define a priority as a strict partial order $\pi \subseteq \gamma \times \gamma$. We write $a\pi b$ for $(a, b) \in \pi$ to express that a has lower priority than b.

Composite Components. A *composite component* $\pi\gamma(B_1, \ldots, B_n)$ (or simply *component*) is defined by a set of atomic components $\{B_i = (Q_i, P_i, T_i)\}_{i=1,n}$ composed by a set of interactions γ and a priority $\pi \subseteq \gamma \times \gamma$. If π is the empty relation, then we omit π and simply write $\gamma(B_1, \ldots, B_n)$. A global state q of $\pi\gamma(B_1, \ldots, B_n)$ is defined by a tuple of control locations $q = (q_1, \ldots, q_n)$. The behavior of $\pi\gamma(B_1, \ldots, B_n)$ is a labelled transition system $(Q, \gamma, \rightarrow_{\pi\gamma})$, where $Q = \bigotimes_{i=1}^{n} Q_i$ and $\rightarrow_\gamma, \rightarrow_{\pi\gamma}$ are the least sets of tran-
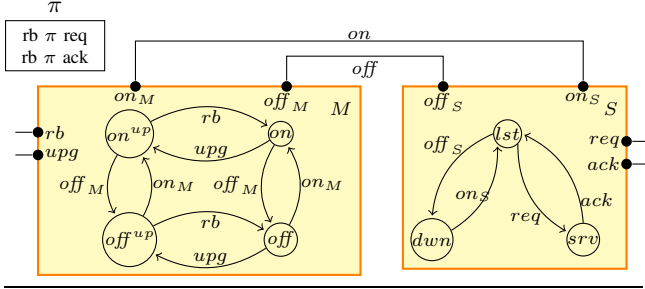
Figure 1. BIP component. Initial state is (off, dwn).

sitions satisfying the rules:

$$a = \{p_i\}_{i \in I} \in \gamma$$
$$\frac{\forall i \in I.\, (q_i, p_i, q'_i) \in T_i \qquad \forall i \notin I.\, q_i = q'_i}{(q_1, \ldots, q_n) \xrightarrow{a}_\gamma (q'_1, \ldots, q'_n)} \text{ [INTER]}$$

$$\frac{q \xrightarrow{a}_\gamma q' \qquad \forall a' \in \gamma.\, a\pi a' \implies q \xrightarrow{a'}_\gamma}{q \xrightarrow{a}_{\pi\gamma} q'} \text{ [PRIO]}$$

Intuitively, transitions $\rightarrow_\gamma$ defined by rule [INTER] specify the behavior of the component without considering priorities. A component can execute an interaction $a \in \gamma$ iff for each port $p_i \in a$, the corresponding atomic component B_i can execute a transition labelled by p_i. If this happens, a is said to be *enabled*. Execution of a modifies atomically the state of all interacting atomic components whereas all others stay unchanged. The behavior of the component is defined by transitions $\rightarrow_{\pi\gamma}$ defined by rule [PRIO]. This rule restricts execution to interactions which are maximal with respect to the priority order. An enabled interaction a can execute only if no other interaction a' with higher priority is enabled.

Example 1. A BIP component is depicted in Figure 1 using a graphical notation. It consists of two atomic components named M and S. Component S is a server, that may receive requests (req) and acknowledge them (ack). Component M is a manager that may perform upgrades (upg) and needs to reboot (rb) the server for the upgrade to be done. Interactions are represented using connectors between the interacting ports. There are 4 unary interactions and 2 binary interactions. The component goes up and down through the binary interactions on and off respectively. Priority rb π req, rb π ack is used to prevent a reboot whenever a request or an acknowledgement are possible.

2.1 Replacing Priority by Observation

According to BIP semantics, a low priority interaction is executed only if all higher priority interactions are not enabled. In general, detecting such situations requires information about components that are not involved in the low priority interaction. We propose here an alternative semantics of BIP parameterized by *Observation*. This semantics makes explicit the sets of components to be observed and the

global state condition to be met for authorizing execution of each interaction.

Observation. Given a BIP component $\gamma(B_1, \ldots, B_n)$, we define an observation as a pair of functions $\mathcal{O} = (obs, pred)$, that are both defined over γ. Let $a \in \gamma$ be an interaction; $obs(a)$ is a subset of $\{B_1, \ldots, B_n\}$ including the set of components *observed* by the interaction a. We require that $obs(a) \cap support(a) = \emptyset$. The observed components and the support of a are the components visible to a, that is $V_a = support(a) \cup obs(a)$. For $a \in \gamma$, $pred(a)$ is a predicate defined on the states of components in V_a.

Composite Component with Observation. A composite component with observation $\mathcal{O}\gamma(B_1, \ldots, B_n)$ is defined by a component $\gamma(B_1, \ldots, B_n)$ and an observation $\mathcal{O}$ over this component. The behavior of $\mathcal{O}\gamma(B_1, \ldots, B_n)$ is the labeled transition system $(Q, \gamma, \longrightarrow_{\mathcal{O}\gamma})$, where $Q = \bigotimes_{i=1}^n Q_i$ is the set of global states, and $\longrightarrow_{\mathcal{O}\gamma}$ is the least set of transitions satisfying the rule:

$$\frac{(q_1, \ldots, q_n) \xrightarrow{a}_\gamma (q'_1, \ldots, q'_n)}{pred(a)\left((q_i)_{B_i \in V_a}\right)}{(q_1, \ldots, q_n) \xrightarrow{a}_{\mathcal{O}\gamma} (q'_1, \ldots, q'_n)} \text{ [OBS]}$$

The rule [OBS] states that a transition a can take place in the component with observation if it is already a valid transition in the component $\gamma(B_1, \ldots, B_n)$ and if the predicate $pred(a)$ holds for the current state of components in V_a. The predicate $pred(a)$, is a boolean expression involving atomic predicates $at(q)$ for each state $q \in \bigcup_{i=1}^n Q_i$. The atomic predicate $at(q)$ evaluates to true whenever the corresponding atomic component is at state q and to false otherwise. The rule [OBS] requires that $pred(a)$ depends only on states of components that are visible to a, that is $pred(a)$ is a boolean expression on $at(q)$ predicates for $q \in \bigcup_{B_i \in V_a} Q_i$.

Example 2. Figure 2 depicts a composite component with observation. Each interaction is labeled by the set of observed components and the corresponding predicate. Here, the only interaction with additional observation is rb, with $obs(rb) = \{S\}$. The predicate for executing rb is written between square brackets.

Observation-based semantics violates the component encapsulation principle as it needs access to inner states of components. We use components with observation as an intermediate model towards a distributed implementation where we exploit the locality of observation: observing only the components visible to an interaction is sufficient to decide whether the interaction can take place.

Priority vs. observation. In Figure 2, we presented an example of composite components with observation. Note that the predicate associated to rb actually encodes the priority rule of Figure 1, since it guarantees that nor req neither ack are enabled when executing rb. We show that given a priority π one can obtain an observation $\mathcal{O}_\pi$ such that the behaviors
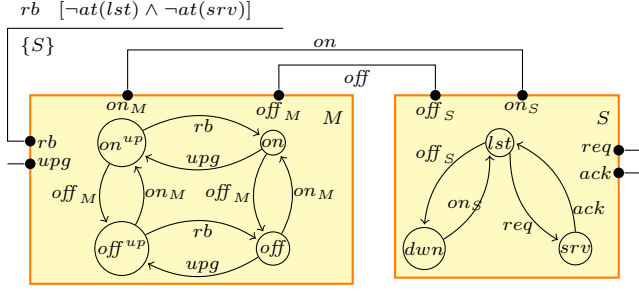
Figure 2. Example of a component with observation.

of the components with priority and observation are identical.

Using $at(q)$ predicates, we define the predicate EN_a stating whether the interaction a is enabled. First, we define the predicate $EN_{p_i}^i$ characterizing enabledness of port p_i in a component $B_i = (Q_i, P_i, T_i)$, that is $EN_{p_i}^i = \bigvee_{(q_i, p_i, -) \in T_i} at(q_i)$. Then, the predicate EN_a can be defined by: $EN_a = \bigwedge_{p_i \in a} EN_{p_i}^i$. Note that this predicate depends only of components in $support(a)$.

Definition 1 (Priority Observation). Given a prioritized BIP component $\pi\gamma(B_1, \ldots, B_n)$, we define the *priority observation* $\mathcal{O}_\pi = (obs, pred)$ for the component $\gamma(B_1, \ldots, B_n)$, for each interaction $a \in \gamma$:

- $obs(a)$ contains all components involved in an higher priority interaction b that do not participate in a. Formally: $obs(a) = (\bigcup_{a \pi b} support(b)) \setminus support(a)$.
- $pred(a)$ ensures that each higher priority interaction b is not enabled. Formally, $pred(a) = \bigwedge_{a \pi b} \neg EN_b$. Obviously, this predicate depends only on components in $support(a) \cup obs(a)$.

For the example in Figure 1, the only low-priority interaction is rb. For all other interactions, $obs(a)$ and $pred(a)$ are respectively $\emptyset$ and $True$. The component with observation obtained from the component with priority is exactly the one depicted in Figure 2. Indeed, rb observes the component S and the predicate on this interaction is $\neg at(lst) \wedge \neg at(srv) = \neg EN_{req} \wedge \neg EN_{ack}$.

Proposition 1. *Given a component with priority $\pi\gamma(B_1, \ldots, B_n)$ and the component with observation $\mathcal{O}_\pi\gamma(B_1, \ldots, B_n)$, where $\mathcal{O}_\pi$ is constructed from π as specified in Definition 1, we have $\longrightarrow_{\pi\gamma} = \longrightarrow_{\mathcal{O}_\pi\gamma}$.*

Proof. For each interaction a, the predicate $pred(a) = \bigwedge_{a \pi b} \neg EN_b$ is equivalent to $\forall b \in \gamma \; a \pi b \implies q \not\xrightarrow{b}_\gamma$. Thus the rules [PRIO] and [OBS] define exactly the same set of transitions. □

In [6], we provided a heuristic to reduce the scope of observation while preserving behavior equivalence. More precisely, this heuristic takes an observation $\mathcal{O}_\pi = (obs, pred)$ and returns another observation $\mathcal{O}' = (obs', pred')$, such that

- $\forall a \in \gamma \; |obs'(a)| \leq |obs(a)|$, the scope of the observation is reduced, and
- $\longrightarrow_{\mathcal{O}'\gamma} \subseteq \longrightarrow_{\mathcal{O}_\pi\gamma}$ the obtained behavior using observation $\mathcal{O}'$ is correct with respect to the original one.

Furthermore, the heuristics ensures that if the inclusion is strict, no deadlocks are introduced. Otherwise, the obtained component has precisely the same behavior as the original one.

2.2 Implementing Observation with Interactions

We start from a component with observation $\mathcal{O}\gamma(B_1, \ldots, B_n)$ and translate it into an equivalent observable BIP component $\gamma'(B_1', \ldots, B_n')$. In order to implement observation, each atomic component has to make explicit its current state, both for interactions where it is involved and for interactions where it is observed. Observation is therefore encoded by extending interactions to observed components.

Transforming Atomic Components. Given an atomic component $B = (Q, P, T)$, we define the corresponding atomic observable component as a labeled transition system $B' = (Q', P', T')$, where:

- $Q = Q'$ the states are the same than in the original component.
- $P' = (P \cup \{obs\}) \times Q$: we add a new port denoted obs, that will be used for observation. All ports contain the information of the current state. We denote by $p(q)$ the port $(p, q) \in P'$.
- For each transition $(q, p, q') \in T$, T' contains the transition $(q, p(q), q')$ where the current state of the component is explicit in the offered port. For $q \in Q$, T' contains the loop transition $(q, obs(q), q)$ that is used when the component is observed.

Transforming Interactions. Given a set γ of interactions and an observation $\mathcal{O} = (obs, pred)$, we define the new set of interactions γ' as follows. For each interaction $a \in \gamma$, where $a = \{p_i\}_{i \in I}$, we extend its support to the components $support(a) \cup obs(a) = \{B_{j_1}', \ldots, B_{j_k}'\}$, and we denote by J the set of indices $\{j_1, \ldots, j_k\}$. For each state of this set of components $(q_{j_1}, \ldots, q_{j_k})$ such that $pred(a)(q_{j_1}, \ldots, q_{j_k})$ holds, γ' contains the interaction $a(q_{j_1}, \ldots, q_{j_k}) = \{p_j'(q_j)\}_{j \in J}$, where $p_j' = obs_j$ if $B_j \in obs(a)$, that is B_j is observed by a, and $p_j' = p_j$ otherwise. This transformation associates to any interaction a of $\mathcal{O}\gamma(B_1, \ldots, B_n)$ a set of interactions $a(q_{j_1}, \ldots q_{j_k})$ of $\gamma'(B_1', .., B_n')$, each interaction of γ' being enabled by states $(q_{j_1}, \ldots, q_{j_k})$ satisfying $pred(a)$.

Proposition 2. *We have $\longrightarrow_{\gamma'} = \longrightarrow_{\mathcal{O}\gamma}$ by mapping the interactions $a(q_{j_1}, \ldots, q_{j_k})$ of γ' to a.*

Proof. The states of $\mathcal{O}\gamma(B_1, \ldots, B_n)$ and $\gamma'(B_1', \ldots, B_n')$ are the same. The transition $q \xrightarrow{a}_{\mathcal{O}\gamma} q'$ can be fired if and only if the components visible to a, namely $\{B_j\}_{j \in J}$,

are in a state $(q_{j_1}, \ldots, q_{j_k})$ satisfying the predicate $pred(a)$. In that case γ' contains an interaction $a(q_{j_1}, \ldots, q_{j_k})$. This interaction only changes the state of participants in a, thus we have $q \xrightarrow{a}_{\gamma'} q'$. $\qquad\square$

Note that the duplication of interactions can be avoided by using models extended with variables and guards on interactions, In that case, instead of creating a new port $p(q)$ for any pair in $P \times Q$, each port exports a state variable q. Then $pred(a)$ is the guard associated with the interaction a, and depends only on variables exported by the ports involved in a.

3. Decentralized Implementation of BIP

We provide here the principle of the method for distributed implementation of BIP presented in [8, 9]. This method relies on a systematic transformation from arbitrary BIP components[2] into distributed BIP components with Send/Receive interactions. These are binary point-to-point and directed interactions from one sender component (port), to one receiver component (port) implementing message passing, from the sender to the receiver. The transformation guarantees that the receive port is always enabled when the corresponding send port becomes enabled, and therefore Send/Receive interactions can be safely implemented using any asynchronous message passing primitives (e.g., MPI send/receive communication, TCP/IP network communication, etc...).

In a distributed setting, each atomic component executes independently and thus has to communicate with other atomic components in order to ensure correct execution with respect to the original semantics. Thus, a reasonable assumption is that each component will publish its offer, that is the list of its enabled ports, and then wait for a notification indicating which interaction has been chosen for execution. This is achieved by splitting each transition in atomic components: one part sends the offer, the other part is triggered by the notification and executes the chosen interaction.

The main difficulty when transforming a BIP component into a distributed Send/Receive BIP component is to resolve conflicts between simultaneously enabled interactions. In a centralized execution, only one entity is responsible for executing interactions, and has exclusive access to all components. In contrast, in a distributed setting, several entities may be responsible for executing interactions. A conflict occurs if two different entities try to execute two interactions involving a common component. If both entities send a notification to this component, then the original semantics is jeopardized, since a component cannot participate in two concurrently enabled interactions. For conflict resolution, a protocol must be used in order to ensure that conflicting interactions are not executed concurrently. This protocol takes into account the offers from components and sends back no-

tifications so that the distributed execution is correct with respect to the original semantics.

Distributed conflict resolution boils down to solving the *committee coordination problem* [12], where a set of professors organize themselves in different committees, a meeting requires the presence of all professors to take place and two committees that have a professor in common cannot meet simultaneously. Different solutions have been provided, using managers [1, 12, 17, 18], circulating tokens [15], or randomized algorithms without managers [14] to implement the conflict resolution.

We first describe how atomic components are modified to send offers and receive notifications. Then, we focus on the Bagrodia's solutions from [1], that use managers and counters to implement conflict resolution. Finally, we recall how these protocols are used for building a 3-layer distributed component.

3.1 Distributed Atomic Components

The transformation of atomic components consists in splitting each transition into two consecutive transitions: (i) an *offer* that publishes the current state of the component, and (ii) a *notification* that triggers the transition corresponding to the chosen interaction. The offer transition publishes its enabled ports through a set of special ports, labeled $o(Off)$ where Off is the subset of enabled ports.

Definition 2 (Distributed atomic components). Let $B = (Q, P, T)$ be an atomic component. The corresponding transformed atomic component is $B^\perp = (Q^\perp, P^\perp, T^\perp)$, such that:

- $Q^\perp = Q \cup \{\perp_q \,|q \in Q\}$ is the union of *stable* states Q and *busy* states $\{\perp_q \,|q \in Q\}$.
- $P^\perp = P \cup \{o(Off)|Off \subseteq P\}$, where $o(Off)$ is a port indicating that ports in $Off \subseteq P$ are enabled.
- the set of transitions $T^\perp$ include, for every transition $\tau = (q, p, q') \in T$:
  1. an *offer* transition $\left(\perp_q, o(\{p|q \xrightarrow{p}\}), q\right)$ that goes from a busy to a stable state and publishes the offer.
  2. a *notification* transition $q \xrightarrow{p} \perp_{q'}$ that goes from a stable to a busy state and executes the transition from the original component.

Notice that we introduced a new port for each possible offer. This allows us to using the same model as for non-distributed atomic components. However, as the notation suggests, we can use a single port o with exported variables as described in [9].

3.2 Bagrodia's Counter-based Conflict Resolution

In Bagrodia's solutions, the protocol is made of one or several managers that receive offers from the atomic components and reply with notifications.

Centralized (Single) Manager. The first solution consists of a single manager. In order to ensure mutual exclusion of conflicting interactions, the protocol maintains two counters for each atomic component B_i:

- The *offer-count* n_i which counts the number of offers sent by the component. This counter is initially set to 0 and is incremented each time an offer from B_i is received.
- The *participation-count* N_i which counts the number of times the component participated in an interaction. This counter is initially set to 0 and is incremented each time the manager selects an interaction involving B_i for execution.

Intuitively, the offer-count n_i associated to an offer from a component B_i correspond to a time stamp. The manager maintains the last used time stamp (N_i) for each component. If the time stamp (n_i) of an offer is greater than the last used time stamp (N_i), then the offer from B_i has not been consumed yet. Otherwise, some interaction has taken place and the manager has to wait for a new offer from this component.

Furthermore, the manager recalls the last offer sent by each component. Thus in order to schedule an interaction, it must check that (1) the interaction is enabled according to the last offers received and (2) these offers are still valid according to the n_i and N_i counters. We define formally the behavior of the centralized protocol as a composition operator over distributed atomic components.

Definition 3 (Centralized Counter-based Implementation). Given a BIP component $\gamma(B_1, \ldots, B_n)$ we define the behavior of the counter-based centralized implementation as an infinite state LTS $(Q^\perp, \gamma^\perp, T^\perp)$ where:

- The set of states $Q^\perp$ is the product of the states of the atomic components with the state of the protocol:

$$Q^\perp = \bigotimes_{i=1}^{n} Q_i^\perp \times \bigotimes_{i=1}^{n} \left(\mathbf{N} \times \mathbf{N} \times 2^{P_i} \right)$$

The state of the manager is defined by n triplets $m_i = (n_i, N_i, \mathit{Off}_i)$, one for each component B_i, where n_i and N_i are the values of the corresponding counters and Off_i is the last offer from B_i. We denote by (q, m) a state of $Q^\perp$, $q[i]$ and $m[i]$ represent the ith component of the tuples q and m.

- The interactions $\gamma^\perp$ consists of interactions of the original component and the offers:

$$\gamma^\perp = \gamma \cup \bigcup_{i=1}^{n} \bigcup_{\mathit{Off} \in 2^{P_i}} o_i(\mathit{Off}_i)$$

- There are two types of transitions in $T^\perp$:
  (1) *offer transitions:* From state $(q, m) \in Q^\perp$, there is an offer transition in $T^\perp$ if for some component B_i an offer is enabled: $(q[i], o_i(\mathit{Off}), q_i') \in T_i^\perp$. Then $T^\perp$ contains the transition $(q, m) \xrightarrow{o_i(\mathit{Off})} (q', m')$, where :

- $q'[i] = q_i'$,
- $m'[i] = (n_i + 1, N_i, \mathit{Off})$, with $m[i] = (n_i, N_i, \mathit{Off}_i)$,
- for all $j \neq i$, $q'[j] = q[j]$ and $m'[j] = m[j]$.

(2) *execute transitions:* From state $(q, m) \in Q^\perp$, there is an execute transition in $T^\perp$ if for some interaction $a = \{p_i\}_{i \in I}$, we have, for all $i \in I$ (with $m[i] = (n_i, N_i, \mathit{Off}_i)$):

- $p_i \in \mathit{Off}_i$: the interaction is enabled according to the last offers,
- $n_i > N_i$: the last offers are still valid.

Then, the transition $(q, m) \xrightarrow{a} (q', m')$ is in $T^\perp$, with:

- $\forall i \in I$, $q'[i]$ is the state such that $(q[i], p_i, q'[i]) \in T_i^\perp$,
- $\forall i \in I$, $m'[i] = (n_i, N_i + 1, \mathit{Off}_i)$: counters of participants are incremented.
- $\forall j \notin I$, $q'[j] = q[j] \wedge m'[j] = m[j]$

We show that the component $\gamma(B_1, \ldots, B_n)$ and the corresponding counter-based implementation are observationally equivalent in the sense of Milner [16]. We first prove the following lemma.

Lemma 1. *If $n_i > N_i$, then the component $B_i^\perp$ is in a stable state q_i and $\mathit{Off}_i = \{p | q_i \xrightarrow{p}_i\}$.*

Proof. The construction of $B_i^\perp$ implies that it alternates offer and execute transitions. Initially, $n_i = N_i$ and $B_i^\perp$ is in a busy state. The only possible transition is an offer, which brings the system to a state where $n_i = N_i + 1 > N_i$ is true and the offer transition ensures the property to prove. Next possible step in $B_i^\perp$ is an execute action, after which again $n_i = N_i$ and $B_i^\perp$ is a busy state. This behavior repeats forever. $\square$

In order to show observational equivalence, we have to define the observable actions of both systems. For the component $\gamma(B_1, \ldots, B_n)$ the observable actions are the interactions γ. For the counter-based implementation, the visible actions are the execute actions γ. We denote by β the offer actions.

We define a relation between states Q of the centralized component and states $Q^\perp$ of its distributed implementation. To each state $q^\perp \in Q^\perp$ of the distributed implementation, we associate a state $e(q^\perp) \in Q$ of the original component. For each component $B_i^\perp$, $q^\perp[i]$ is either a stable state q_i or a busy state $\perp_{q_i}$. In both cases, we take $e(q^\perp)[i] = q_i$. We say that a state $q \in Q$ and $q^\perp \in Q^\perp$ are equivalent, denoted by $q^\perp \sim q$, if $q = e(q^\perp)$.

Proposition 3 (Correctness of Centralized Counter-based Implementation). *Given a component $\gamma(B_1, \ldots, B_n)$, the labeled transitions systems (Q, γ, T) and $(Q^\perp, \gamma^\perp, T^\perp)$ of its distributed implementation are observationally equivalent.*

Proof. We have to prove that:

1. If $q^\perp \xrightarrow{\beta} r^\perp$, then $\forall q \sim q^\perp$, $r \sim q^\perp$.
2. If $q^\perp \xrightarrow{a} r^\perp$, then $\forall q \sim q^\perp$, $\exists r \in Q$ $q \xrightarrow{a}_\gamma r \wedge r \sim r^\perp$.
3. If $q \xrightarrow{a} r$, then $\forall q^\perp \sim q$, $\exists r^\perp \in Q^\perp$ $q^\perp \xrightarrow{\beta^* a} r^\perp \wedge r \sim r^\perp$.

1. This is a consequence of the definition of $\sim$.

2. The transition $(q^\perp, a, r^\perp)$ is possible at state $q^\perp \in Q^\perp$ if for each participant B_i in the interaction, the counters verify $n_i > N_i$, and for each port $p_i \in a$, we have $p_i \in Off_i$. The Lemma 1 ensures that in the equivalent state $q \in Q$, we have as well $q \xrightarrow{a} r$. The construction of distributed atomic components ensures that $r \sim r^\perp$.

3. If $q \xrightarrow{a} r$, then for each state $q^\perp \sim q$, each participant B_i in a is either in a busy or in a stable state. In the first case, it can perform an offer transition, labeled β, and reach a stable state. By point 1., the stable state $q'^\perp$ such that $q^\perp \xrightarrow{\beta^*} q'^\perp$ is also equivalent to q. At state $q'^\perp$, all offers transitions for a have been executed and we have $q^\perp \xrightarrow{\beta^*} q'^\perp \xrightarrow{a} r^\perp$, with $r^\perp \sim r$. $\qquad\square$

In Definition 3, the enabling of offer transitions depends exclusively on the state of the component sending the offer. Similarly, the enabling of execute transitions is decided by the manager alone. Thus we can assume an asynchronous execution where an offer transition is executed first by the atomic component, by sending a message and then by the manager when receiving the message. Similarly, the execute transitions are performed after the manager sends messages to components involved in the interaction.

Decentralized (Multiple) Manager(s). In [1], Bagrodia decentralizes the manager into a set of distributed managers, also relying on counters to ensure correct execution of the interactions. The correctness is guaranteed as long as each manager can check and modify atomically all the N_i counters corresponding to an interaction. Bagrodia proposes two protocols guaranteeing this atomicity:

- The token ring protocol, where a token circulates through all managers. This token stores the N_i counters for the whole system, which guarantees atomic access for each manager.
- The dining philosophers protocol, where two interactions that involve a common component share a fork with a copy of the N_i counter on it. In order to execute an interaction, the manager needs to acquire all forks and can then check and update if necessary all N_i values simultaneously.

It can be shown that these protocols are trace equivalent with the centralized implementation [9]. However, they are not observationally equivalent with the centralized implementation, since the position of the token or of the forks may prevent some choices to be made (see [9] for details).

3.3   3-layer Distributed Architecture

The obtained distributed components must meet the following three properties: (1) preserve the behavior of each atomic component, (2) preserve the behavior of interactions, and (3) resolve conflicts in a distributed manner. To ensure these properties, we structure distributed components according to a hierarchical architecture with three layers. The lower layer includes the transformed atomic components. The second layer deals with distributed interaction execution by implementing interaction protocols (IP). The third layer deals with conflict resolution. Since several distributed algorithms exist for conflict resolution, this layer is generic with appropriate interfaces. An example of 3-layer architecture obtained from the component presented in Figure 1 is depicted in Figure 3.
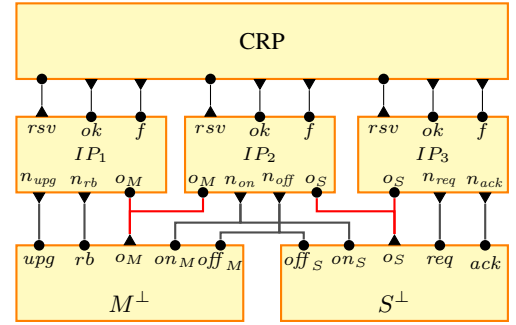

Figure 3. 3-layer distributed implementation of component from Figure 1.

Components Layer. This layer contains the distributed version of the atomic components, as described in section 3.1. In Figure 3, it corresponds to components $M^\perp$ and $S^\perp$.

Interaction Protocol. This layer consists of a set of interaction protocols each hosting a set of interactions from the original BIP component. Conflicts between interactions included in the same interaction protocol are resolved by that component locally. On Figure 3, IP_1 handles interaction upg and rb, IP_2 handles on and off, and IP_3 handles req and ack.

The interaction protocol evaluates the guard of each interaction and executes the code associated with an interaction that is selected locally or by the upper layer. The interface between this layer and the component layer provides ports for receiving offers from each component (through ports such as o_M) and notifying the components on permitted port for execution (through ports such as n_{on}). Sender ports are denoted by triangles and receiver ports by bullets. Interactions with one sender and multiple receivers means that the sender sequentially sends a message to each receiver.

Conflict Resolution Protocol. This algorithm embeds one of the Bagrodia's counter-based protocols as presented in the previous section. The protocols have been slightly modified since managers do not receive offers one by one from components but instead receive the set of offers corresponding to

an interaction sent by one of the interaction protocols. The protocol can either be centralized, or distributed e.g. token ring or dining philosophers. The interface between this layer and the Interaction Protocol involves ports for receiving requests to *reserve* an interaction (labelled rsv) and responding by either success (labelled ok) or failure (labelled f).

4. Distributed Implementation of Observational Semantics

Applying the transformation presented in Subsection 2.2 followed by the distribution method presented in 3 allows to obtain a distributed model from a component with observation. This method leads to a *multiparty-based* implementation. We show here that a multiparty-based implementation is costly, as it treats all observation conflicts as structural conflicts. We propose an optimized version of Bagrodia's counter-based protocol presented in the previous section, that allows us to build an *observation-aware* implementation.

4.1 Observation Conflicts

Using the transformation presented in 2.2, we can transform a component with observation into a observable component. This transformation implements observation of components through new ports denoted obs. However, it introduces new structural conflicts between interactions on the observation ports obs.
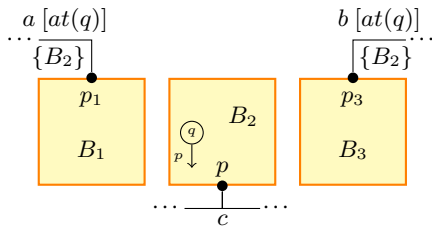

Figure 4. Model with observation.

As an example, consider the model depicted in Figure 4. It contains three atomic components and three fragments of interaction. Interactions a and b observe the atomic component B_2. Execution of a or b will not change the state of B_2 since none of its transitions is involved. Intuitively, a and b can be executed in parallel, they do not really conflict. However, execution of c changes the state of the atomic component B_2 and may disable the predicate associated to a or b. Thus a and c cannot be executed simultaneously. They are conflicting.

This type of conflicts also appears in transactional memories [13]. In this context, different transactions (interactions) can simultaneously read (observe) a variable (an atomic component), but writing on a variable (executing a transition) requires exclusive access to the variable.

When we transform such a model with observation into a observable model, as described in subsection 2.2, we obtain
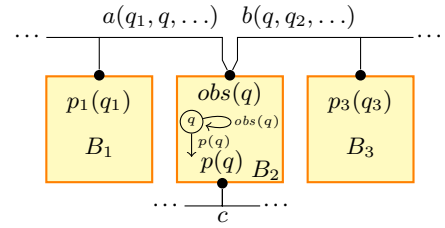

Figure 5. Observable model obtained from the model with observation in Figure 4.

the model depicted in the Figure 5. The observation is implemented by adding a new port $obs(q)$ and extending interactions a and b to that new port. In this model, B_2 becomes a participant in the interactions a and b by executing a loop transition. This results in a structural conflict between a and b.

The 3-layer distributed implementation generated from a component obtained with the transformation presented in Subsection 2.2 involves an unnecessarily high number of exchanged messages. Consider the model presented in Figure 5. Execution of interaction a followed by interaction b requires at least 4 messages between the component B_2 and the protocol. Indeed, each interaction requires at least one offer and one notification. These four messages could be replaced by a single one, indicating that B_2 is at state q to the protocol, since the component B_2 does not need to be notified when it is observed.

4.2 Counter-based Conflict Resolution for Observation

The transformation from a component with observation to an observable component adds new conflicts and results in a message-inefficient distributed implementation. In order to avoid this, we modify the conflict resolution protocol to take observation into account. The particularity of observation is checking that a component is at a particular state, without state change. This differs from multiparty interactions, where observation is combined with state change.

The proposed adaptation of the counter-based protocol presented in Definition 3 can be reused in the 3-layer BIP model to encompass observation and thus priority.

This adaptation relies on the following key facts:

- *Observation of a component does not imply state change*. Freshness of the offer from a component (the observation) is still validated by checking $n_i > N_i$. However, upon execution of an interaction, the N_i counters corresponding to the observed components are not incremented. Thus $n_i > N_i$ still holds and another interaction observing the same component can still take place.

- *The state predicates need to be checked*. This assumes that every component sends its local state with its offer and that the manager knows the state predicate for each interaction.

Definition 4. Given a BIP component with observation $\mathcal{O}\gamma(B_1, \ldots, B_n)$ we define the behavior of the adapted counter-based centralized implementation as an infinite state LTS $(Q^\perp, \gamma^\perp, T^\perp)$ where:

- The set of states $Q^\perp$ is the product of the states of the atomic components with the state of the protocol:

$$Q^\perp = \bigotimes_{i=1}^{n} Q_i^\perp \times \bigotimes_{i=1}^{n} \left(\mathbf{N} \times \mathbf{N} \times 2^{P_i} \times Q_i\right)$$

  The state of the manager is defined by n quadruples $m_i = (n_i, N_i, \mathit{Off}_i, q_i)$, one for each component B_i, where n_i and N_i are the values of the corresponding counters, Off_i is the last offer from B_i and q_i is the last known state from B_i. We denote by (q, m) a state of $Q^\perp$, $q[i]$ and $m[i]$ represent the ith element of the tuples q and m.

- The interactions of $\gamma^\perp$ include the interactions from the original component and the offers:

$$\gamma^\perp = \gamma \cup \bigcup_{i=1}^{n} \bigcup_{\mathit{Off} \in 2^{P_i}} o_i(\mathit{Off}_i)$$

- There are two types of transitions in $T^\perp$:
  (1) offer transitions: From state $(q, m) \in Q^\perp$, there is an offer transition in $T^\perp$ if for some component B_i an offer is enabled: $(q[i], o_i(\mathit{Off}), q_i') \in T_i^\perp$. That is, $T^\perp$ contains the transition $(q, m) \xrightarrow{o_i(\mathit{Off})} (q', m')$, where:
    - $q'[i] = q_i'$,
    - $m'[i] = (n_i + 1, N_i, \mathit{Off}, q[i])$ (with $m[i] = (n_i, N_i, \mathit{Off}_i, q_i)$,
    - for all $j \neq i$, $q'[j] = q[j]$ and $m'[j] = m[j]$.
  (2) execute transitions: From state $(q, m) \in Q^\perp$, there is an execute transition in $T^\perp$ if for some interaction $a = \{p_i\}_{i \in I}$, we have, for all $i \in I$ (with $m[i] = (n_i, N_i, \mathit{Off}_i, q_i)$):
    - $p_i \in \mathit{Off}_i$: the interaction is enabled according to the last offers,
    - $n_i > N_i$: the last offers are still valid.
  Furthermore, we require that $pred(a)((q_i)_{B_i \in V_a})$ holds. Then, the transition $(q, m) \xrightarrow{a} (q', m')$ is in $T^\perp$, with:
    - $\forall i \in I$, $q'[i]$ is the state such that $(q[i], p_i, q'[i]) \in T_i^\perp$,
    - $\forall i \in I$, $m'[i] = (n_i, N_i + 1, \mathit{Off}_i, q_i)$: counters of participants are incremented.
    - $\forall j \notin I$, $q'[j] = q[j] \wedge m'[j] = m[j]$

As for the counter-based implementation, we prove the correctness of the adapted version using Milner's observational equivalence.

Proposition 4 (Correctness of adapted Counter-based Implementation). *Given a component $\mathcal{O}\gamma(B_1, \ldots, B_n)$, the labeled transitions systems (Q, γ, T) and $(Q^\perp, \gamma^\perp, T^\perp)$ of its distributed implementation are observationally equivalent.*

The proof has the same structure as for the Proposition 3, and uses the same equivalence relation. The only difference is in points 2. and 3. where we have to take into account the additional enabling condition. More precisely, we have to show that the truth value of the enabling condition is preserved by the equivalence relation restricted to *stable* states. This is obtained by considering the counters of observed components.

The correctness is guaranteed through the fact that checking the freshness of offers sent by visible components and incrementing the counters of participant components is an atomic action. Thus as for Bagrodia's original version, the manager can be distributed provided this atomicity is ensured, either by the token ring or by the dining philosophers solutions.
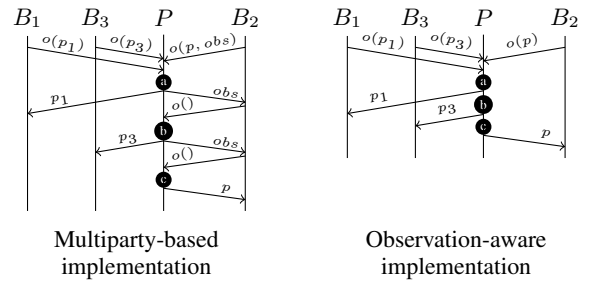

Figure 6. Exchanges of messages to execute the sequence a, b, c in the model of Figure 4, for the two implementations.

Example 3. To illustrate the behavior of this new protocol, consider again the model depicted in Figure 4. We obtain a multiparty-based implementation by transforming it into the model of Figure 5 and then using the original protocol from Bagrodia. The modified protocol presented here allows to obtain an observation-aware implementation directly from the model in Figure 4. In Figure 6, we compare the behavior of the two approaches, when executing the interaction sequence a, b, c. On the left, we show the messages exchanged in the multiparty-based implementation. On the right we show the messages exchanged in the observation-aware implementation. For each process (the distributed components B_i and the protocol P) Figure 6 presents the sequence of messages received and sent. The black circles indicate that an interaction is scheduled by the Protocol. Note that the component B_2 is observed by a and b and is participant in c. With the multiparty-based implementation, the observation is treated as a participation. Both execution of a and b trigger the emission of a notification (obs) to B_2 followed by a new offer ($o()$). With the observation-aware implementation, the first offer sent by B_2 is observed but not consumed by a and b. So, there is no need to send notifications and wait for corresponding offers. Only the execution of c consumes the offer. For this particular configuration, the new protocol spares 4 messages and increases parallelism since b and c
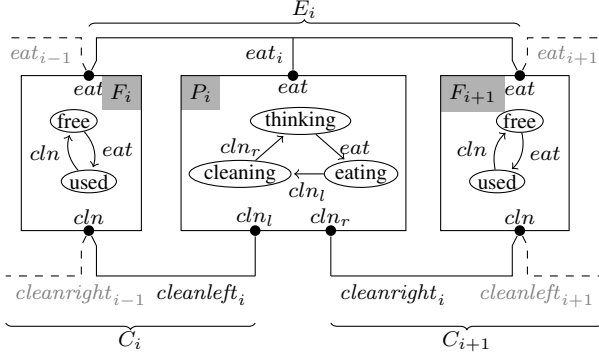
Figure 7. Fragment of the dining philosopher component. Braces indicate how interactions are grouped into interaction protocols.

can be launched directly after a, without waiting for a new offer.

The observation-aware implementation is more message-efficient than the multiparty-based implementation. If there is no observation, both implementations behave exactly the same. If there is an observation, executing the observing interaction results in the emission of a notification to each observed component in the multiparty-based implementation. This notification is not generated in the observation-aware implementation. Moreover, in the observation-aware implementation, an offer may be shared between several interactions observing the same component, reducing further the overall number of messages.

5. Experiments

We compare the execution time and the number of exchanged messages for several distributed implementations of a component with priority. The first step involves transformation of this component into a component with observation. Then we consider the two following sequences of transformations.

- Transform the component with observation into an observable component as explained in Subsection 2.2. Then generate a 3-layer distributed model embedding Bagrodia's conflict resolution protocol described in Subsection 3.2. This method results in a multiparty-based implementation.
- Directly transform the component with observation into a 3-layer distributed model embedding the modified conflict resolution protocol described in Subsection 4.2. This method results in a observation-aware implementation.

For both implementations, we used the centralized version of the conflict resolution protocol.

5.1 Dining Philosophers

We consider a variation of the dining philosophers problem, denoted by Philo$N$ where N is the number of philosophers. A fragment of this composite component is presented in Figure 7. In this component, an "eat" interaction eat_i involves a philosopher and the two adjacent forks. After eating, philosopher P_i cleans the forks one by one ($cleanleft_i$ then $cleanright_i$). We consider that each eat_i interaction has higher priority than any $cleanleft_j$ or $cleanright_j$ interaction.

This example has a particularly strong priority rule. Indeed, executing one "$clean$" interaction requires to check that *all* "eat" interactions are disabled, that is to observe all components. This example allows to compare both implementations under strong priority constraints.

As explained in Section 3.3, the construction of our distributed implementation is structured in 3 layers. The second layer is parameterized by a partition of the interactions. For this example, the partition is built as follows. There is one interaction protocol E_i for every eat_i interaction and one interaction protocol C_i for every pair $cleanright_{i-1}$, $cleanleft_i$. Only the latter deals with low priority interactions that need to observe additional atomic components.

We compare multiparty-based and observation-aware implementations. For both, once we have built the distributed components, we use a code generator that generates a standalone C++ program for each atomic component. These programs communicate by using Unix sockets.
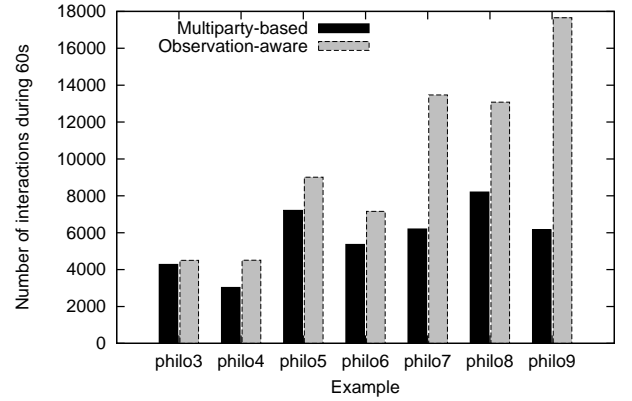

Figure 8. Number of interactions executed in 60s for the dining philosophers example.

The obtained code has been run on a UltraSparc T1 that allows parallel execution of 24 threads. For each run, we count the number of interactions executed and messages exchanged in 60 seconds, not including the initialization phase. For each instance we consider the average values obtained over 10 runs. The number of interactions executed by each implementation is presented in Figure 8. The total number of messages exchanged for the execution of each implementation is presented in Figure 9.
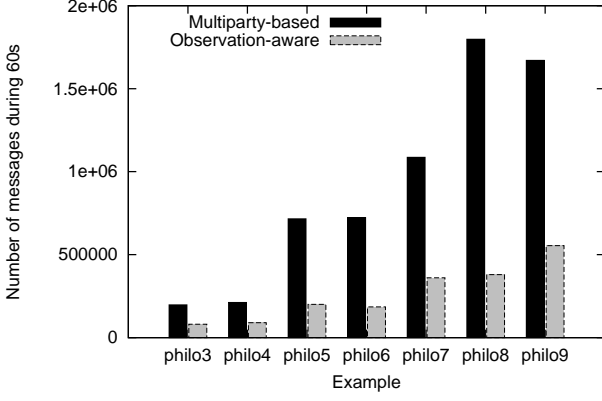
Figure 9. Number of messages exchanged in 60s for the dining philosophers example.

The comparison of the two implementations shows a huge difference both in performance (number of interactions executed) and communications needed (total number of messages exchanged). The observation-aware implementation is fastest and needs fewer messages than multiparty-based implementation. This can be explained as follows. In both cases, eat_i interactions can execute in parallel, provided they do not involve a common fork. However, resolving priority conflicts requires to observe all components for executing a $cleanleft_i$ or a $cleanright_i$ interaction. In the multiparty-based implementation, observed components must synchronize to execute some interaction $cleanleft_i$ or $cleanright_i$. Between two "*clean*" executions, each component has to receive a notification and to send a new offer. This strongly restricts the parallelism. In the observation-aware implementation, a component offer is still valid after execution of an interaction observing that component. For a "*clean*" interaction, only two components will need to send a new offer before another "*clean*" interaction can be executed. This explains the speedup.

5.2 Jukebox

The second example is a jukebox depicted in Figure 10. It represents a system, where a set of readers $R_1 \ldots R_4$ access data located on N disks $D_1 \ldots D_N$. Readers may need to access any disk. We denote by jukebox$N$ the jukebox component with N disks. Access to disks is managed by jukeboxes J_1, J_2 that can load any disk to make it available to the connected readers. The interaction $load_{i,k}$ (respectively $unload_{i,k}$) allows loading (respectively unloading) the disk D_i in the jukebox J_k. Each reader R_j is connected to a jukebox through the $read_j$ interaction. Once a jukebox has loaded a disk, it can either take part in a "read" or "unload" interaction. Each jukebox repeatedly loads all N disks in a random order.

If unload interactions are always chosen immediately after a disk is loaded, then readers may never be able to read data. Therefore, we add the priority $unload_{i,k} \pi read_j$, for all i, j, k. This ensures that "read" interactions will take place before corresponding disks are unloaded. Furthermore, we assume that readers connected to J_1 need more often disk 1 and that readers connected to J_2 need more often disk 2. Therefore, loading these disks in the corresponding jukeboxes is assigned higher priority: $load_{i,1} \pi load_{1,1}$ for $i \in \{2,3\}$ and $load_{i,2} \pi load_{2,2}$ for $i \in \{1,3\}$. Each interaction is handled by a dedicated interaction protocol.
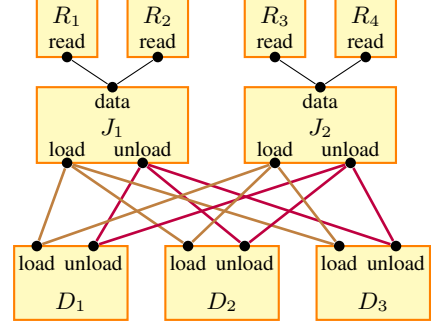

Figure 10. Jukebox component with 3 discs.

Compared to the Dining Philosopher example, this one has more localized priorities, in the sense that they do not require to observe the global state of the system. Here a priority rule is used to express a scheduling policy that aims to improve the efficiency of the system, in terms of "*read*" interactions. Generating the same example without taking priority into account results in an implementation that does less "*read*" interactions.
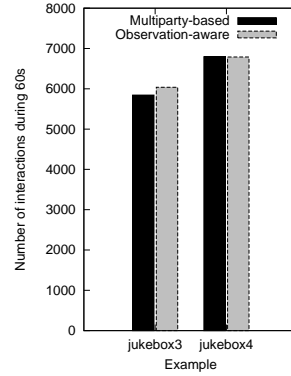

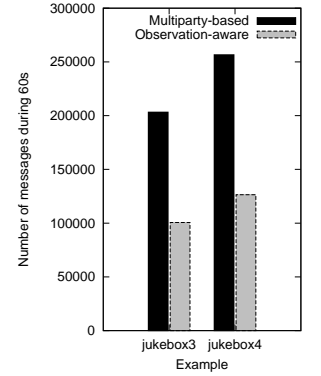**Figure 11.** Number of interaction executed in 60s for the jukebox example.

Figure 12. Number of messages exchanged in 60s for the jukebox example.

We performed the same measurements, in the same conditions as for the previous example. The number of interactions executed in 60s is presented in Figure 11. Here performance of both versions is the same. The main reason is that no or few parallelism is allowed between low priority interactions, i.e. two "*unload*" interactions from the same jukebox cannot be launched sequentially and run in parallel since they involve the same jukebox. However, Figure 12 shows that fewer messages are exchanged, with the

observation-aware implementation. Intuitively, this difference corresponds to the notifications and subsequent offers to and from observed components, that are not necessary with the observation-aware implementation.

6. Conclusion

We proposed different methods of generating a distributed implementation for multiparty interactions with observation. The proposed model ensures enhanced expressiveness as the enabling conditions of an interaction can be strengthened by state predicates of components non participating in that interaction. It directly encompasses modeling of priorities which are essential for modeling scheduling policies. We have proposed a transformation leading from a model with observation into an equivalent model with interactions. The transformation consists in creating events making visible state-dependent conditions.

Expressing observation by interactions allows the application of existing distributed implementation techniques, such as the one presented in [9]. We have proposed an optimization of the conflict resolution algorithm from [1] that takes into account the fact that an observed component does not participate in the observing interaction. Preliminary experiments show significant performance improvement of this optimized implementation method.

Future work directions include the study of knowledge-based techniques [6] for efficient conflict resolution, in particular by minimizing the set of the observed components for each interaction. We also plan to study optimized implementations of systems with multiparty interaction and observation, for other implementations based on other conflict resolution protocols, such as α-core [18].

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, P. Bidinger, M. Bozga, and J. Sifakis. Distributed semantics and implementation for systems with interaction and priority. In *Formal Techniques for Networked and Distributed Systems (FORTE)*, pages 116–133, 2008.

[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] Ananda Basu, Saddek Bensalem, Doron Peled, and Joseph Sifakis. Priority scheduling of distributed systems based on model checking. *Formal Methods in System Design*, 39(3):229–245, 2011.

[5] S. Bensalem, M. Bozga, S. Graf, D. Peled, and S. Quinton. Methods for knowledge based controlling of distributed systems. In *Automated Technology for Verification and Analysis - 8th International Symposium, ATVA 2010, Proceedings*, volume 6252, pages 52–66. Springer, September 2010.

[6] S. Bensalem, M. Bozga, J. Quilbeuf, and J. Sifakis. Knowledge-based distributed conflict resolution for multi-

[7] Saddek Bensalem, Doron Peled, and Joseph Sifakis. Knowledge based scheduling of distributed systems. In *Essays in Memory of Amir Pnueli*, pages 26–41, 2010.

[8] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, and J. Sifakis. From high-level component-based models to distributed implementations. In *EMSOFT*, 2010.

[9] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, and J. Sifakis. A framework for automated distributed implementation of component-based models. *Distributed Computing*, 25(5):383–409, 2012.

[10] B. Bonakdarpour, M. Bozga, and J. Quilbeuf. Automated distributed implementation of component-based models with priorities. In *EMSOFT*, pages 59–68, 2011.

[11] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(4):632–646, 1984.

[12] K. M. Chandy and J. Misra. *Parallel program design: a foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[13] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.

[14] Y.-J. Joung and S. A. Smolka. Strong interaction fairness via randomization. *IEEE Trans. Parallel Distrib. Syst.*, 9(2):137–149, 1998.

[15] D. Kumar. An implementation of n-party synchronization using tokens. In *ICDCS*, pages 320–327, 1990.

[16] R. Milner. *Communication and concurrency*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1995.

[17] J. Parrow and P. Sjödin. Multiway synchronizaton verified with coupled simulation. In *International Conference on Concurrency Theory (CONCUR)*, pages 518–533, 1992.

[18] 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.