Science of Computer Programming 98 (2015) 293–316

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Optimized distributed implementation of multiparty interactions with Restriction ☆

Saddek Bensalem [a], Marius Bozga [a], Jean Quilbeuf [a,*], Joseph Sifakis [a,b]

[a] UJF-Grenoble 1/CNRS VERIMAG UMR 5104, Grenoble, F-38041, France
[b] RISD Laboratory, EPFL, Lausanne, CH-1015, Switzerland

ARTICLE INFO

Article history:
Received 3 March 2013
Received in revised form 30 January 2014
Accepted 5 February 2014
Available online 20 February 2014

Keywords:
Multiparty interaction
Priority
Observation
Conflict resolution
Distributed systems

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 a new composition operator named Restriction, whose semantics dynamically restricts the set of interactions allowed for execution, depending on the current state. We show that this operator provides a natural encoding for priorities. We provide a knowledge-based optimization that modifies the Restriction operator to avoid superfluous communication in the final implementation. We complete our framework through an enhanced conflict resolution protocol that natively implements Restriction. A prototype implementation allows us to compare performances of different optimizations.

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 from high-level models to code.

In this paper, we focus on distributed implementation for models defined using the BIP framework [6]. 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 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]

☆ This article extends two papers, presented at the AGERE!2012 workshop and at the FMOODS/FORTE 2012 conference. The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007–2013] under grant agreement No. 248776 (PRO3D) and No. 257414 (ASCENS) and from ARTEMIS JU grant agreement ARTEMIS-2009-1-100230 (SMECY).

* Corresponding author.
E-mail addresses: bensalem@imag.fr (S. Bensalem), bozga@imag.fr (M. Bozga), quilbeuf@fortiss.org (J. Quilbeuf), sifakis@imag.fr (J. Sifakis).

[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.

http://dx.doi.org/10.1016/j.scico.2014.02.013
0167-6423/

A multiparty interaction is a high-level construct that expresses a strong synchronization between a fixed set of components. Such an interaction takes place only if all its participant components agree to execute it. If two multiparty interactions involve a common component, they are conflicting because the common component cannot participate in both interactions. 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. Such interactions should be executed in mutual exclusion.
3. *Enforcing priorities*. When two interactions are simultaneously enabled, only the one with higher priority can be chosen for execution. Priorities can be applied indifferently between conflicting or non-conflicting interactions.

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 [16,17]. This method has been later extended to handle priorities [18] and optimized by exploiting knowledge [12]. The target model consists of components representing processes and Send/Receive interactions representing asynchronous message passing. Correct coordination is 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* [20] for implementing multiparty interactions. Intuitively, this problem consists in scheduling several meetings, every one involving a set of professors. A meeting requires the whole attendance. A professor cannot participate in more than one meeting at a time. Bagrodia [3] proposes solutions to this problem with different degrees of parallelism. The most distributed solution is based on the drinking philosophers problem [19], and has inspired the later approaches of Pérez et al. [41] and Parrow and Sjödin [39]. In the context of BIP, a transformation addressing all the three challenges through employing a *centralized scheduler* is proposed in [5]. Moreover, in [16], the transformation is extended to 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. However, such an implementation should simultaneously enforce the priority rules and the mutual exclusion of conflicting interactions. In [18], priorities are eliminated by adding explicit scheduler components. This transformation leads to potentially more complex models, having definitely more interactions and conflicts than the original one. In [7], situations where priorities and multiparty interactions are intermixed, called confusion, are avoided by adding more priorities.

Enforcing priority rules is done when deciding execution of low priority interactions, by checking that no interaction with more priority is ready to execute. This check requires a synchronous view of the components involved in higher priority interactions. The distributed knowledge [22] of a component consists of all the information that it can infer about other components state, based on its current state and the reachable states. In [15,8,4], 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 [13], we introduce a new composition operator called Restriction. This operator associates a state predicate to each multiparty interaction. The semantics of Restriction allows executing an interaction only if the associated predicate evaluates to true in the current state. For instance, an interaction “start” representing a car starting at a crossing might be restricted with the predicate “the traffic light is green”. Note that the predicate possibly depends on components that do not participate in the interaction. In our example, the traffic light component is not necessarily a participant in the “start” interaction.

This paper is an extension of both [12] and [13], and combines the two approaches. This combination yields several methods for obtaining a distributed implementation of multiparty interactions subject to priorities. These methods rely on an appropriate intermediate model and transformations towards fully distributed models. Each transformation, depicted by an arrow in Fig. 1, either refines the composition operator used to glue components of the model or optimizes the model. The contribution is manyfold:

1. First, we introduce an alternative semantics for BIP that relies on the Restriction operator. We show that this operator is general enough to encompass priorities through a simple transformation (Transformation 1 of Fig. 1). The Restriction operator reveals two types of conflicts occurring between interactions, that can be handled using different conflict resolution mechanisms (see below).
2. Second, we show that the knowledge-based optimization originally presented in [12] can be extended to handle the Restriction and reduce the overall coordination of the model. This optimization (Transformation 2 of Fig. 1) modifies only the predicates used by the Restriction operator.
3. Third, a model with Restriction can be used as an intermediate step in the transformations 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*, where two multiparty interactions involve a common component. In particular, we compare two approaches for generating a distributed implementation. The first consists in encoding Restriction with

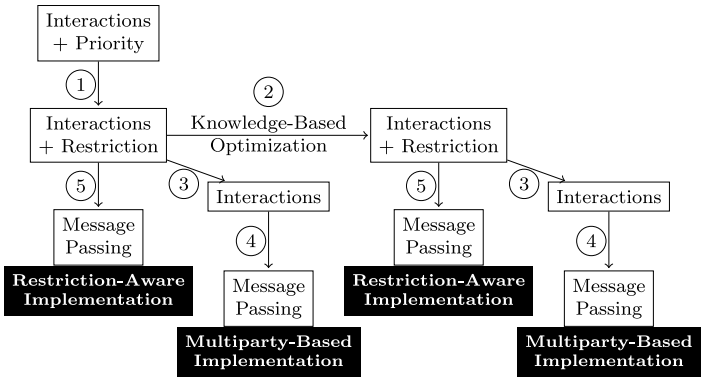

Fig. 1. Possible paths to generate a distributed implementation from a model encompassing multiparty interactions and priorities.

multiparty interactions and then using a conflict resolution protocol (Transformations 3 and 4). The second is new and uses a conflict resolution protocol designed to handle Restriction (Transformation 5).

4. These approaches have been fully implemented and evaluated through benchmarks.

The paper is organized as follows. Section 2 introduces the main concepts of the BIP framework together with the new Restriction composition operator. An adaptation of the knowledge-based optimization from [12] for Restriction is presented in Section 3. Section 4 recalls the principles for distributed implementation of BIP models, focusing on conflict resolution by using counter-based protocols and present a solution for distributed implementation of Restriction. Experiments are reported in Section 5. Section 6 presents the related work for conflict resolution, priorities and knowledge. Section 7 provides conclusions and perspectives for future work.

2. Semantic models of BIP

In this section, we present BIP [6], 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 type of composition, named Restriction, that can express Priority. Finally, we present a transformation from a component with Restriction into an equivalent component with only Interaction.

Atomic components. An *atomic component* B is a labeled transition system represented by a tuple (Q, q^0, P, T) where Q is a set of *control locations* or *states*, q^0 is the initial state, P is a set of *communication ports* and $T \subseteq Q \times P \times Q$ is a set of *transitions*.

We write $q \xrightarrow{p} q'$ if $(q, p, q') \in T$. If the target state is irrelevant, we write $q \xrightarrow{p}$ to express that an outgoing transition labeled by p is possible at state q.

2.1. Interactions & priorities

In order to compose a set of n atomic components $\{B_i = (Q_i, q_i^0, P_i, T_i)\}_{i=1,n}$, we assume that their respective sets of control locations and ports are pairwise disjoint; i.e., for any two $i \neq j$ in $\{1, \ldots, n\}$, 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. In the sequel, we denote by *participants*(a) the components that participate in a, and by $I_a \subseteq \{1, \ldots, n\}$ the corresponding indices. We also denote $a = \{p_i\}_{i \in I_a}$, where $p_i \in P_i$ is the port of B_i participating in 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 a 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 $\{B_i = (Q_i, q_i^0, P_i, T_i)\}_{i=1,n}$ 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 labeled transition system $(Q, q^0, \gamma, \rightarrow_{\pi\gamma})$, where $Q = \bigotimes_{i=1}^{n} Q_i$, $q^0 = (q_1^0, \ldots, q_n^0)$ and $\rightarrow_\gamma, \rightarrow_{\pi\gamma}$ are the least sets of transitions satisfying the rules:

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

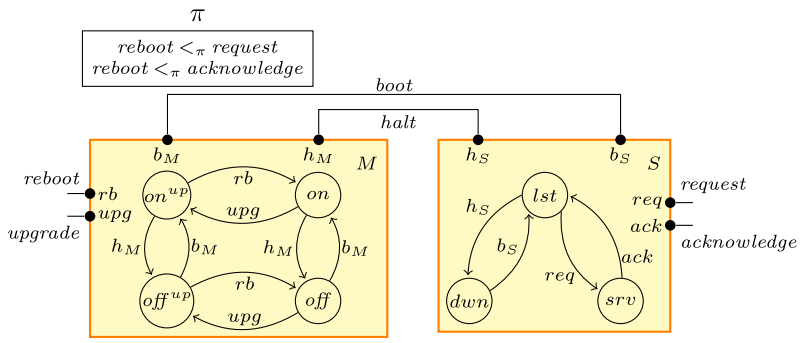

Fig. 2. A BIP component. Initial state is (off, dwn).

$$\frac{q \xrightarrow{a}_{\gamma} q' \qquad \forall a' \in \gamma \,.\, a <_{\pi} a' \Longrightarrow q \not\xrightarrow{a'}_{\gamma}}{q \xrightarrow{a}_{\pi\gamma} q'} \text{[PRIO]}$$

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 labeled 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 specified by transitions $\rightarrow_{\pi\gamma}$ defined by rule [PRIO]. This rule restricts execution to interactions which are maximal with respect to the priority order among the enabled ones. 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 Fig. 2 using a graphical notation. It consists of two atomic components named M and S. The component S is a server, that may receive requests (through the port req) and acknowledges them (through the port ack). Formally, $S = (\{lst, dwn, srv\}, dwn, \{h_S, b_S, req, ack\}, \{(lst, h_S, dwn), (dwn, b_S, lst), (lst, req, srv), (srv, ack, lst)\})$. The component M is a manager that may perform upgrades (upg) and needs to reboot (rb) the server for the upgrade to be effective. Interactions are represented using connectors between the interacting ports. There are 4 unary interactions, that are interactions involving a single port, and 2 binary interactions. Formally $\gamma = \{\{b_M, b_S\}, \{h_M, h_S\}, \{rb\}, \{upg\}, \{req\}, \{ack\}\}$. In the figure, we use names to denote interactions, for instance $\{b_M, b_S\}$ is called *boot*. The depicted system goes up and down through the binary interactions *boot* and *halt* respectively. The priority $\pi = \{(reboot, request), (reboot, acknowledge)\}$ is used to prevent a reboot whenever a request or an acknowledgment are possible.

2.2. Interaction and Restriction

The priority operator is defined over a system made of components composed by interactions. We consider a new operator, also defined over components composed by interactions, called Restriction operator. A Restriction operator assigns a predicate, defined on the global states, to each interaction. This operator inhibits execution of interactions for which the predicate evaluates to false at the current state. We later show that a priority operator can be expressed as a Restriction operator.

Given a BIP component $\gamma(B_1, \ldots, B_n)$, we define Restriction as a function ρ, such that for each interaction $a \in \gamma$, $\rho(a)$ is a boolean formula involving atomic predicates of the form $at(q)$, where $q \in \bigcup_{i=1}^{n} Q_i$ is one of the local states of the atomic components. The predicate $at(q)$ is true at a given global state $(q_1, \ldots, q_n)$, if q is a local state of the atomic component B_i and $q = q_i$. We denote by $\rho_a : Q \rightarrow \{True, False\}$ the predicate obtained by combining the $at(q)$ predicates according to $\rho(a)$, that is the semantics of $\rho(a)$. In the sequel, we often refer directly to the predicate ρ_a, instead of using the formula $\rho(a)$.

A composite component with Restriction $\rho\gamma(B_1, \ldots, B_n)$ is defined from a component $\gamma(B_1, \ldots, B_n)$ and a Restriction operator ρ over this component. The behavior of $\rho\gamma(B_1, \ldots, B_n)$ is the labeled transition system $(Q, q^0, \gamma, \longrightarrow_{\rho\gamma})$, where $Q = \bigotimes_{i=1}^{n} Q_i$ is the set of global states, $q^0 = (q_1^0, \ldots, q_n^0)$ is the initial state, and $\longrightarrow_{\rho\gamma}$ is the least set of transitions satisfying the rule:

$$\frac{q \xrightarrow{a}_{\gamma} q' \quad \rho_a(q)}{q \xrightarrow{a}_{\rho\gamma} q'} \text{[RESTR]}$$

The rule [RESTR] states that a transition a can execute if it is already a valid transition in the component $\gamma(B_1, \ldots, B_n)$ and if the predicate ρ_a holds for the current state of the components.

Although it is defined on the global state, a Restriction predicate ρ_a might depend only on a subset of the components. We use the syntactical definition given for the formula $\rho(a)$ to determine the set of components on which ρ_a depends. More precisely, we denote by $support(\rho(a))$ the set of components $\{B_i \mid \exists q \in Q_i.\ at(q)$ appears in $\rho(a)\}$. The components

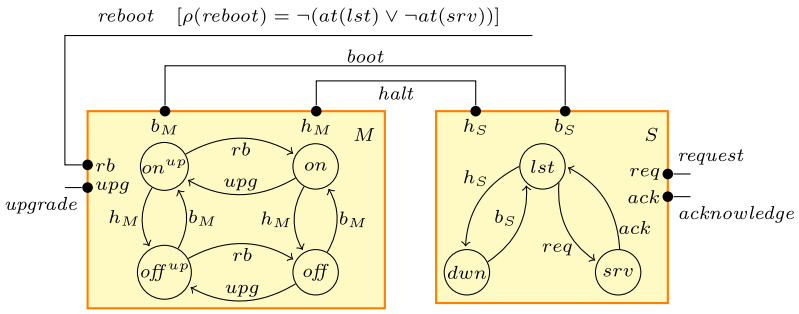

Fig. 3. A component with Restriction.

in $observed_\rho(a) \stackrel{\text{def}}{=} support(\rho(a)) \setminus participants(a)$ play a particular role in the execution of the interaction a: they do not participate in the interaction but their state is observed to decide whether it can execute.

Example 2. Fig. 3 depicts a composite component with Restriction. Each interaction is labeled by the corresponding formula, which is omitted if it is true. The Restriction operator specified in the example implements the priority operator from Example 1. Here, *reboot* is the only interaction whose formula is not true, because it is the only low priority interaction. The formula assigned to *reboot* is $\neg(at(lst) \vee at(srv))$. It evaluates to false whenever interactions with more priority than *reboot* are enabled.

The Restriction operator violates the component encapsulation principle as it depends on the inner states of components and not only on their interfaces (ports). It is however useful because it explicitly defines the components to observe for deciding the execution of an interaction a. Indeed, this decision can be taken solely based on the states of components in $observed_\rho(a)$ and $participants(a)$.

2.3. Implementing priority with Restriction

In Fig. 3, we presented an example of composite component with Restriction. Note that the predicate associated to *reboot* actually encodes the priority rule of Fig. 2, since it guarantees that nor *request* neither *acknowledge* are enabled when executing *reboot*. We show that given a Priority π one can obtain a Restriction ρ^π such that the behaviors of the components with Priority and Restriction are identical. This corresponds to the Transformation 1 of Fig. 1.

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

Definition 1 *(Priority Restriction).* Let $\pi\,\gamma(B_1, \ldots, B_n)$ be a prioritized BIP component. The *priority Restriction* ρ^π associates to each interaction $a \in \gamma$ the predicate $\rho^\pi_a = \bigwedge_{a <_\pi b} \neg EN_b$.

For the example in Fig. 2, the only low-priority interaction is *reboot*. For every other interaction a, the predicate ρ^π_a is *True*. The component with Restriction obtained from the component with Priority is exactly the one depicted in Fig. 3. Indeed, the predicate ρ_{reboot} is $\neg at(lst) \wedge \neg at(srv)$ which is equivalent to $\neg EN_{request} \wedge \neg EN_{acknowledge}$.

Proposition 1. *Given a component with Priority $\pi\,\gamma(B_1, \ldots, B_n)$ and the component with Restriction $\rho^\pi\,\gamma(B_1, \ldots, B_n)$, where ρ^π is constructed from π as specified in Definition 1, we have $\longrightarrow_{\pi\gamma} = \longrightarrow_{\rho^\pi\gamma}$.*

Proof. For each interaction a, the predicate $\rho^\pi_a = \bigwedge_{a <_\pi b} \neg EN_b$ is equivalent to $\forall b \in \gamma.(a <_\pi b \Longrightarrow q \stackrel{b}{\not\rightarrow}_\gamma)$. Thus the rules [PRIO] and [RESTR] define exactly the same set of transitions. □

2.4. Implementing Restriction with interactions

We start from a component with Restriction $\rho\gamma(B_1, \ldots, B_n)$ and translate it into an observably equivalent BIP component $\gamma'(B'_1, \ldots, B'_n)$ by using the Transformation 3 of Fig. 1. In order to implement Restriction, each atomic component has to make explicit its current state, both for interactions where it participates and for interactions where it is observed. Then each interaction is extended so that components that are observed by an interaction a because of the Restriction ρ become participants in the corresponding interaction in γ'.

2.4.1. Transforming atomic components

Given an atomic component $B = (Q, q^0, P, T)$, we define the corresponding observable atomic component as a labeled transition system $B' = (Q', q'^0 P', T')$, where:

- $Q = Q'$ and $q^0 = q'^0$ the states are the same as in the original component as well as the initial state.
- $P' = (P \cup \{obs\}) \times Q$: 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.

The ports of the transformed atomic component are actually couples made of one port and one state of the original component. These explicit the current state of the component to the interactions, in order to evaluate Restriction predicates.

2.4.2. Transforming interactions

Given a set γ of interactions and a Restriction ρ, we define the set of interactions γ'. If $a \in \gamma$ is an interaction, it is transformed into a set of interactions that are included in γ'. By definition, given a global state $q = (q_1, \ldots, q_n)$, the value of $\rho_a(q)$ depends only on components in $participants(a) \cup observed_\rho(a)$. We denote by I_a (resp. J_a) the indices of components in $participants(a)$ (resp. $observed_\rho(a)$), and we denote by $\{i_1, \ldots, i_k\}$ the indices in I_a and J_a. For each interaction $a = \{p_i\}_{i \in I_a}$ and each partial state $(q_{i_1}, \ldots, q_{i_k})$ that satisfies ρ_a, γ' contains the interaction $a(q_{i_1}, \ldots, q_{i_k}) = \{p_i(q_i)\}_{i \in I_a} \cup \{obs_j(q_j)\}_{j \in J_a}$. This construction extends each original interaction $a = \{p_i\}_{i \in I_a}$ to a set of interactions defined over the transformed ports $p_i(.)$ and the additional ports $obs_j(.)$ of the observed components. The interaction between ports $p_i(q_i)$ and $obs_j(q_j)$ is added only if $\rho_a(q_1, \ldots, q_n)$ evaluates to true.

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

Proof. The state spaces of $\rho\gamma(B_1, \ldots, B_n)$ and $\gamma'(B'_1, \ldots, B'_n)$ are the same. The transition $q \xrightarrow{a}_{\rho\gamma} q'$ can be fired if and only if the components visible to a, namely $participants(a) \cup observed_\rho(a)$, denoted $\{B_{i_1}, \ldots, B_{i_k}\}$, are in a state $(q_{i_1}, \ldots, q_{i_k})$ satisfying the predicate ρ_a. In that case γ' contains an interaction $a(q_{i_1}, \ldots, q_{i_k})$. This interaction only changes the state of participants in a, thus we have $q \xrightarrow{a}_{\gamma'} q'$.

Conversely, if we have $q \xrightarrow{a}_{\gamma'} q'$, with $q = (q_1, \ldots, q_n)$, there is an interaction $a(q_{i_1}, \ldots, q_{i_k})$ in γ'. By definition of γ' we have $\rho_a(q)$. If $a(q_{i_1}, \ldots, q_{i_k})$ is enabled at state q, each port $p_i(q_i)$ for $i \in I_a$ is enabled, thus a is enabled. By combining that a is enabled at q and $\rho_a(q)$ holds, we have $q \xrightarrow{a}_{\rho\gamma} q''$. If the transition $q_i \xrightarrow{p_i(q_i)} q'_i$ is in the component B'_i, then the transition $q_i \xrightarrow{p_i} q'_i$ is in the component B_i, therefore $q'' = q'$. □

Note that the duplication of interactions can be avoided by using models extended with variables and guards on interactions, such as the one presented in [17]. 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 ρ_a is the guard associated with the interaction a, and depends only on variables exported by the ports involved in a.

3. Knowledge-based reduction of observation

Before executing an interaction a, one must check that the state of atomic components in $participants(a)$ and in $observed_\rho(a)$ satisfies ρ_a. In a distributed context, consistently checking this condition requires a synchronized observation of the corresponding components. Therefore, our first optimization, corresponding to Transformation 2 of Fig. 1, tries to minimize the number of components to synchronize in order to check ρ_a. Since executing an interaction a involves synchronizing its participants, the only set that could be reduced is $observed_\rho(a)$.

In this section, we use distributed knowledge to replace a given Restriction ρ by a new one ρ'. The new Restriction reduces the set of observed components $observed_{\rho'}(a)$ for each interaction. The distributed knowledge ρ'_a is a safe under-approximation of ρ_a, depending only on $participants(a)$ and $observed_{\rho'}(a)$. Distributed knowledge assumes that every component knows the set of reachable states, or at least one over-approximation of it.

There is a tradeoff between minimizing the number of observed components in ρ' and the faithfulness of the under-approximation. When reducing the set of observed components $observed_{\rho'}(a)$, one also restricts the set of global states where ρ'_a detects a valid interaction, that is, evaluates to true. To characterize how much of originally valid transitions are detected, we define *detection levels*.

We first explain in Section 3.1 how the set of reachable states is approximated using invariants. We then present distributed knowledge in Section 3.2. In Section 3.3, we show how knowledge can be used to compute a correct Restriction, provided that the set of components to observe is given. We also define two detection levels that characterize faithfulness of

the obtained component with respect to the original component. In Section 3.4, we provide heuristics that try to minimize the number of observed components, while ensuring a given detection level.

3.1. Invariants and reachable states

Let $B = \rho\gamma(B_1, \ldots, B_n)$ be a component with Restriction. We say that the state q is reachable (from the initial state q^0) if there exist a sequence of interactions $a_1, \ldots, a_k \in \gamma$ and states $q^1, \ldots, q^k$ such that $q^0 \xrightarrow{a_1}_{\rho\gamma} q^1 \xrightarrow{a_2}_{\rho\gamma} \cdots \xrightarrow{a_k}_{\rho\gamma} q^k = q$. We denote by $\mathcal{R}$ the set of reachable states of B.

An invariant of B is a predicate $\mathfrak{I} : Q \to \{True, False\}$ satisfied by all reachable states of B. The characteristic set $\tilde{\mathcal{R}}$ of $\mathfrak{I}$ provides an over-approximation of the reachable states ($\mathcal{R} \subseteq \tilde{\mathcal{R}}$). We are interested in two types of inductive invariants that can be generated automatically [14], respectively:

- *boolean invariants*, that is, conjunctions of boolean constraints of the form $\bigvee_{j\in J} at(q_j)$. For the example of Fig. 2, $at(on^{up}) \vee at(on) \vee at(dwn)$ is a boolean invariant. Indeed, by executing any transition from a state where the invariant holds, we reach a state where the invariant still holds. For instance, executing the interaction *halt* disables the predicates $at(on)$ or $at(on^{up})$ but enables the predicate $at(dwn)$. It characterizes a set of control locations such that at each global state, at least one location of the set is active. Such constraints are obtained using methods described in [14].
- *linear invariants*, that is, conjunctions of linear constraints of the form $\sum_{j\in J} k_j at(q_j) = k_0$, where all k_j and k_0 are integer constants, and $at(q_i)$ is equal to 1 if q_i is active and 0 otherwise. For the example of Fig. 2, $at(on^{up}) + at(on) + at(dwn) = 1$ is a linear constraint. Again, by executing any transition from a state where the invariant holds, we reach a state where the invariant still holds. A linear constraint corresponds to a set of places such that the weighted sum of tokens in the places remains constant throughout the execution. Linear invariants are obtained using algebraic techniques as described in [34].

The two above categories of invariants are particularly useful for several reasons. First, they provide good approximations for the enabling/disabling conditions of interactions. This has been empirically demonstrated by the successful application of such invariants for checking deadlock-freedom of component-based systems in BIP [14,10]. Second, the methods for computing these invariants are tractable and scale for large systems. Their computation is based on the (interaction) structure of the system and can be done incrementally [9]. In particular, it does not involve fixpoints computation and avoids state space exploration.

3.2. Knowledge and indistinguishability

Given a model $\gamma(B_1, \ldots, B_n)$, the knowledge of a set of atomic components $\mathcal{L} \subseteq \{B_1, \ldots, B_n\}$ is the information about the current global states through a synchronized observation of the components in $\mathcal{L}$. Intuitively, the synchronized observation of the components gives a partial state of the system. The corresponding global state is (1) reachable and (2) coherent with the partial state observed on $\mathcal{L}$. The information obtained through an observation of $\mathcal{L}$ correspond to properties that are true in all global states verifying conditions (1) and (2). Any subset $\mathcal{L}$ induces an equivalence relation on the global states, defined as follows.

Definition 2 *(Indistinguishability equivalence for $\mathcal{L}$).* Given $\mathcal{L}$, we define the indistinguishability equivalence $\sim_{\mathcal{L}}$ on global states $q \in Q$ as $q \sim_{\mathcal{L}} q'$ iff $\forall B_j \in \mathcal{L}\ q_j = q'_j$.

Intuitively, two states are indistinguishable for $\mathcal{L}$ if their restrictions to the states of atomic components in $\mathcal{L}$ are identical. An equivalence class of this relation is a set of global states that are coherent with an observation of the local state of atomic components in $\mathcal{L}$. Given an over-approximation $\tilde{\mathcal{R}}$ of the reachable states and an arbitrary state predicate ϕ, we define the predicate "$\mathcal{L}$ knows ϕ" as $K_{\mathcal{L}}^{\tilde{\mathcal{R}}}\phi(q) = \forall q' \in \tilde{\mathcal{R}}\ q' \sim_{\mathcal{L}} q \Longrightarrow \phi(q')$. This predicate is defined on $\tilde{\mathcal{R}}$. Intuitively, a set of components $\mathcal{L}$ knows a predicate ϕ at a global state q if ϕ holds in any reachable state q' that $\mathcal{L}$ cannot distinguish from q.

Fig. 4 illustrates $K_{\mathcal{L}}$ with respect to ϕ and $\tilde{\mathcal{R}}$. Each global state within $\tilde{\mathcal{R}}$ is a point characterized by two coordinates: the projections of this state on the states of $\mathcal{L}$ and the states of its complement $\overline{\mathcal{L}} = \{B_1, \ldots, B_n\} \setminus \mathcal{L}$. On the left, the gray region represents the characteristic set of ϕ. In the middle, the gray region represents the characteristic set of "$\mathcal{L}$ knows ϕ" that is the set of the global states where observation of $\mathcal{L}$ suffices to assert "ϕ is true". On the right, the gray region represents the set of the states where "$\mathcal{L}$ knows not ϕ" that is the set of the global states where observation of $\mathcal{L}$ suffices to assert "ϕ is false".

3.3. Building a Restriction with reduced observation

This subsection defines a new Restriction ρ' built as a knowledge approximation of ρ. This new Restriction is parameterized by assigning to each interaction $a \in \gamma$ a set of components $obs(a)$ that defines observation. We require that $obs(a) \cap participants(a) = \emptyset$.

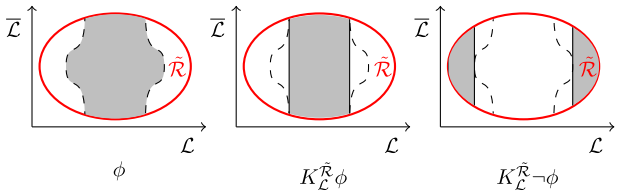

Fig. 4. Knowledge-based approximation of ϕ for observation $\mathcal{L}$, within reachable states $\tilde{\mathcal{R}}$.

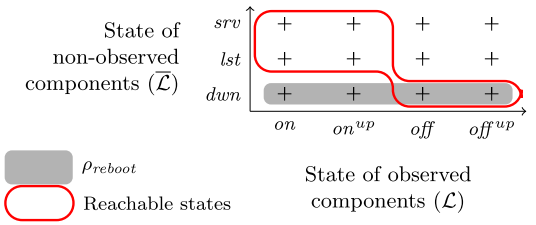

Fig. 5. Global states of the example from Fig. 3, assuming that the component M is observed.

Definition 3. Given a component $B = \rho\gamma(B_1, \ldots, B_n)$, an over-approximation $\tilde{\mathcal{R}}$ of its reachable states and the sets of observed components $\{obs(a)\}_{a\in\gamma}$, the Restriction ρ' with reduced observation associates to each interaction $a \in \gamma$ the predicate $\rho'_a = K^{\tilde{\mathcal{R}}}_{\mathcal{L}_a}\rho_a$ where $\mathcal{L}_a = participants(a) \cup obs(a)$.

By definition of knowledge, we have $\rho'_a \Rightarrow \rho_a$. Furthermore, ρ'_a depends only on the state of components in $\mathcal{L}_a$. In other words, the actual set of observed components for each interaction a is exactly $observed_{\rho'}(a) = \mathcal{L}_a \setminus participants(a)$, that is the parameter $obs(a)$. Therefore, in the sequel, we directly denote by $observed_{\rho'}(a)$ the parameter $obs(a)$.

Example 3. We detail the reduction of the Restriction predicate for the interaction *reboot* of the example in Fig. 3. The predicate $\rho_{reboot} = \neg at(srv) \wedge \neg at(lst)$ depends on the state of the component S, and thus we have originally $observed_{\rho}(reboot) = \{S\}$. In that very simple example, the only possible reduction is to take $observed_{\rho'}(reboot) = \emptyset$. It means that we rely only on the state of M, that is the unique participant in *reboot*, to determine whether ρ_{reboot} holds.

In Fig. 5, we present all the global states of the system, and indicate whether they are reachable or not. This figure is the instantiation of Fig. 4 when considering the example in Fig. 3 and observing only M. On this example, both boolean and linear invariants characterize the exact set of reachable states. For instance, the state (off, lst) is not reachable as it does not satisfy the boolean constraint $at(on^{up}) \vee at(on) \vee at(dwn)$. We obtain the same conclusion by using the linear constraint $at(on^{up}) + at(on) + at(dwn) = 1$.

In the figure, each local observation (state of M) corresponds to a column representing the global states that are indistinguishable from that local observation. The interaction *reboot* is enabled at local state on^{up} and off^{up}. When the local state is on^{up}, the system is either at state (on^{up}, lst) or (on^{up}, srv). In both cases, ρ_{reboot} does not hold, thus "M knows $\neg\rho_{reboot}$" at that state. Conversely, at local state off^{up}, the global state is (off^{up}, dwn). Therefore, M knows that ρ_{reboot} holds if its current state is off^{up}. Finally, the new Restriction predicate for *reboot* is $\rho'_{reboot} = at(off^{up})$, which depends only on the state of M.

By reducing too much the set $observed_{\rho'}(a)$, one takes the risk of always obtaining $\rho'_a = false$, as the observation might not be sufficient to ensure that ρ_a holds. We define two criteria characterizing different detection levels, namely basic and complete. Intuitively, the condition for basic detection ensures that the sets of states enabling at least one interaction from γ is the same for ρ and ρ'. In contrast, the condition for complete detection ensures that every interaction in γ is enabled in the same states for ρ and ρ'.

Definition 4 *(Detection levels).* Let $\rho\gamma(B_1, \ldots, B_n)$ be a component and ρ' be a Restriction obtained by reducing observation in ρ using the over-approximation $\tilde{\mathcal{R}}$ of the reachable states. We say ρ' is:

- *Basic* iff $\forall q \in \tilde{\mathcal{R}}.(\bigvee_{a\in\gamma}(\rho_a(q) \wedge EN_a(q)) = \bigvee_{a\in\gamma}(\rho'_a(q) \wedge EN_a(q)))$.
- *Complete* iff for each interaction $a \in \gamma$: $\forall q \in \tilde{\mathcal{R}}.(EN_a(q) \wedge \rho'_a(q) = EN_a(q) \wedge \rho_a(q))$.

Theorem 1 relates the detection levels and corresponding guarantees on the component equipped with the computed Restriction ρ'. Baseness ensures that ρ' does not introduce deadlocks. Completeness ensures that the global behaviors of the component equipped with ρ and the component equipped with ρ' are identical.

Algorithm 1 Pseudo-code of simulated annealing

Input: An initial solution *init*, a `cost` function, an `alter` function, temperature bounds Θ_{max} and Θ_{min}.
Output: A solution with a minimized cost.

```
1: sol := init
2: Θ := Θ_max
3: while Θ > Θ_min do
4:    sol' := alter(sol)
5:    Δ := cost(sol') − cost(sol)
6:    if Δ < 0 or random() < e^(−Δ/Θ) then
7:       sol := sol'
8:    end if
9:    Θ := 0.99 × Θ
10: end while
11: return sol
```

Theorem 1. *Let $\rho\gamma(B_1, \ldots, B_n)$ be a component and ρ' be the Restriction obtained by reducing observation in ρ. Then, $\longrightarrow_{\rho'\gamma} \subseteq \longrightarrow_{\rho\gamma}$ and:*

1. *If ρ' is basic, then $q \in \tilde{R}$ is a deadlock for $\longrightarrow_{\rho'\gamma}$ only if q is a deadlock for $\longrightarrow_{\rho\gamma}$.*
2. *If ρ' is complete, then $\longrightarrow_{\rho'\gamma} = \longrightarrow_{\rho\gamma}$.*

Proof. Since for each $a \in \gamma$, $\rho'_a \Longrightarrow \rho_a$, we have $\longrightarrow_{\rho'\gamma} \subseteq \longrightarrow_{\rho\gamma}$.

1. By contraposition, let $q \in \tilde{R}$ be a deadlock-free state for $\longrightarrow_{\rho\gamma}$, i.e. such that $\exists a \in \gamma \ (EN_a(q) \wedge \rho_a(q))$. Baseness ensures that $\bigvee_{a\in\gamma}(EN_a \wedge \rho'_a)$ holds and thus $\exists b \in \gamma$ such that $EN_b(q) \wedge \rho'_b(q)$. Thus $q \xrightarrow{b}_{\rho'\gamma}$ and q is a deadlock-free state for $\longrightarrow_{\rho'\gamma}$.

2. Assume that $q \xrightarrow{a}_{\rho\gamma} q'$. Then $EN_a \wedge \rho_a(q)$ holds. Completeness ensures that $EN_a \wedge \rho'_a(q)$ also holds. Thus $q \xrightarrow{a}_{\rho'\gamma} q'$. □

These results characterize to what extent the original Restriction can be captured through partial observation. The approach suggested by this subsection is to come up with some sets of components to observe, then compute the corresponding Restriction and see whether it fits a given detection level. In the next subsection, we propose the reverse approach, that is heuristics that minimize the number of observed atomic components, yet ensuring the required detection level.

3.4. Heuristics to minimize observed components

In this subsection, we propose for each detection level from Definition 4 a heuristic that takes as input a component with Restriction and outputs minimized sets of observed components $\{observed_{\rho'}(a)\}_{a\in\gamma}$. In general, finding the optimal solution, that is which minimizes the number of observed components while ensuring baseness or completeness is hard. This problem is actually similar to the identification of the minimal number of attributes required to distinguish a set of objects in the rough set theory [40] which is known to be NP-hard. Each of the proposed heuristics guarantees that the reduced Restriction ρ' built using the returned sets of observed components meets the corresponding detection level. The results depend upon the approximation $\tilde{\mathcal{R}}$ of the reachable states used for computing the knowledge. We assume throughout this subsection a fixed over-approximation $\tilde{\mathcal{R}}$ of the reachable states. In practice, $\tilde{\mathcal{R}}$ is provided by linear or boolean invariants.

We propose a solution to the minimizing observation problem based on the simulated annealing meta-heuristic [33]. A pseudo-code for the simulated annealing is shown in Algorithm 1. This heuristic allows searching for optimal solutions to arbitrary cost optimization problems. The search through the solution space is controlled by a *temperature* parameter Θ that ranges from Θ_{max} to Θ_{min} during the execution. At every iteration of the simulated annealing, temperature decreases slowly (line 9) and the current solution moves into a new, nearby solution *still ensuring either baseness or completeness* (line 4). If the new solution is better (i.e. observes fewer components), then it becomes the current solution. Otherwise, it may be accepted with a probability that decreases when (1) the temperature decreases or (2) the extra cost of the new solution increases (line 6). The idea is to temporarily allow a bad (but correct) solution whose neighbors may be better than the current one. By the end of the process, the temperature is low, which prevents bad solutions from being accepted. The choice of the parameters Θ_{min} and Θ_{max} depends on the possible cost values that depend on the model, and also define the execution time of the simulated annealing. These values should be chosen so that a very bad solution can be accepted when temperature $\Theta = \Theta_{max}$ and that the probability to accept a solution worse than the current one is almost 0 when $\Theta = \Theta_{min}$. In the sequel, we only provide initial solutions *init* as well as `alter` and `cost` functions that are used to ensure either completeness or baseness.

Algorithm 2 Function `alter` for ensuring completeness

Input: A component $\rho\gamma(B_1,\ldots,B_n)$, an interaction a and a solution $\mathcal{L}_a$.
Output: A solution $\mathcal{L}'_a$ that is a neighbor of $\mathcal{L}_a$.

```
1:  L'_a := L_a
2:  choose B_i in L'_a \ participants(a)
3:  L'_a := L'_a \ {B_i}                                   //perturbation
4:  while EN_a ∧ K^R̃_{L'_a} ρ_a ≠ EN_a ∧ ρ_a do
5:      choose B_i in {B_1,...,B_n} \ L'_a
6:      L'_a := L'_a ∪ {B_i}                               //completion
7:  end while
8:  choose B'_i in L'_a \ participants(a)
9:  while EN_a ∧ K^R̃_{L'_a} ρ_a = EN_a ∧ ρ_a do
10:     L'_a := L'_a \ {B'_i}                              //reduction
11:     if L'_a = participants(a) then
12:         EXIT participants(a)                           //cannot observe less components
13:     end if
14:     choose B'_i in L'_a \ participants(a)
15: end while
16: return L'_a
```

3.4.1. Ensuring completeness

According to Definition 4, checking for completeness is performed interaction by interaction. Therefore, minimizing observation can be carried out independently for each interaction. Given an interaction a we are seeking for a minimal set of atomic components $\mathcal{L}_a$ such that $K^{\tilde{\mathcal{R}}}_{\mathcal{L}_a}\rho_a = \rho_a$. Note that finding such a set $\mathcal{L}_a$ yields the corresponding set of components to observe by taking $observed_{\rho'}(a) = \mathcal{L}_a \setminus participants(a)$.

The initial solution is obtained by taking the set of atomic components that are needed to decide ρ_a, that is $init_a = participants(a) \cup observed_\rho(a)$. At each iteration of the simulated annealing, a new solution is computed using the `alter` function shown in Algorithm 2. First, one atomic component is removed from the solution (perturbation, line 3), possibly breaking completeness. Then, new atomic components are added randomly until the solution ensures complete detection again (completion, line 6). Note that loop terminates because in the worst case observing all components is sufficient to decide whether ρ_a holds. Finally, atomic components are removed randomly, provided they do not contribute to completeness (reduction, line 10). This loop stops when removing a component breaks completeness or when there is no more component to remove. The latter case arises when observing the participants is enough to evaluate the Restriction predicate. In that case (line 12), the simulated annealing can be stopped as $participant(a)$ is an optimal solution.

After completion and during reduction steps, the completeness condition is checked (line 9). On termination, this ensures that the solution returned by the heuristic is complete.

The cost of a solution is obtained by counting the number of atomic components in $observed_{\rho'}(a) = \mathcal{L}_a \setminus participants(a)$. The cost function is thus $\texttt{cost}(\mathcal{L}_a) = |\mathcal{L}_a \setminus participants(a)|$.

3.4.2. Ensuring baseness

Baseness is achieved if for every state where an interaction is allowed by ρ, at least one interaction is also allowed in ρ'. Baseness is a global property for the set of all interactions. Indeed, when reducing the set of components observed by an interaction, one restricts the set of states where this interaction can be executed. Whether this restriction breaks baseness depends on whether there exists another interaction which can execute at the removed states. Therefore, a solution $\{\mathcal{L}_a\}_{a\in\gamma}$ to the minimizing observation ensuring baseness cannot be built independently for each interaction.

The initial solution assumes that each interaction a observes all atomic components that are needed to decide ρ_a, that is $init_a$. Thus the initial solution is $init = \{init_a\}_{a\in\gamma}$. As for completeness, the `alter` function for baseness presented in Algorithm 3 computes a new solution based on the same three steps (perturbation, completion, reduction) being performed on a family of sets of observed atomic components, instead of a single set. As in the previous case, the completion terminates by observing all components in the worst case. Conversely, the reduction terminates either when the baseness property is broken or when no more component can be removed. Again, in the latter case the whole algorithm can be stopped as an optimal solution has been found.

After completion and during reduction steps, the baseness condition is checked (line 9). This guarantees that the returned solution is basic. Here the cost of the solution is the sum of the number of atomic components observed by each interaction. Thus, we define the `cost` function as $\texttt{cost}(\{\mathcal{L}_a\}_{a\in\gamma}) = \sum_{a\in\gamma} |\mathcal{L}_a \setminus participants(a)|$.

4. Decentralized implementation of BIP

We provide here the principle of the method for distributed implementation of BIP as presented in [17,16]. This method relies on a systematic transformation from arbitrary BIP components into distributed BIP components. A distributed BIP component relies only on message passing interactions. A message passing interaction is defined by a send port and a receive port and models the passing of a message between the corresponding components. The transformation guarantees that

Algorithm 3 Function `alter` for ensuring basic detection

Input: A component $\rho\gamma(B_1, \ldots, B_n)$, a solution $\{\mathcal{L}_a\}_{a\in\gamma}$,
Output: A solution $\{\mathcal{L}'_a\}_{a\in\gamma}$ that is a neighbor of $\{\mathcal{L}_a\}_{a\in\gamma}$.

$\{\mathcal{L}'_a\}_{a\in\gamma} := \{\mathcal{L}_a\}_{a\in\gamma}$
choose b **in** γ **and** B_i **in** $\mathcal{L}_b \setminus participants(b)$
$\mathcal{L}'_b := \mathcal{L}'_b \setminus \{B_i\}$ *//perturbation*
while $\bigvee_{a\in\gamma}(EN_a \wedge K^{\tilde{B}}_{\mathcal{L}'_a}\rho_a) \neq \bigvee_{a\in\gamma}(EN_a \wedge \rho_a)$ **do**
  choose b **in** γ **and** B_i **in** $\{B_1, \ldots, B_n\} \setminus \mathcal{L}_b$
  $\mathcal{L}'_b := \mathcal{L}'_b \cup \{B_i\}$ *//completion*
end while
choose b **in** γ **and** B_i **in** $\mathcal{L}_b \setminus participants(b)$
while $\bigvee_{a\in\gamma}(EN_a \wedge K^{\tilde{B}}_{\mathcal{L}'_a}\rho_a) = \bigvee_{a\in\gamma}(EN_a \wedge \rho_a)$ **do**
  $\mathcal{L}'_b := \mathcal{L}'_b \setminus \{B_i\}$ *//reduction*
  if $\forall a \in \gamma, \mathcal{L}'_a = participants(a)$ **then**
    EXIT $\{participants(a)\}_{a\in\gamma}$ *//cannot observe less components*
  end if
  choose b **in** γ **and** B_i **in** $\mathcal{L}_b \setminus participants(b)$
end while
return $\{\mathcal{L}'_a\}_{a\in\gamma}$

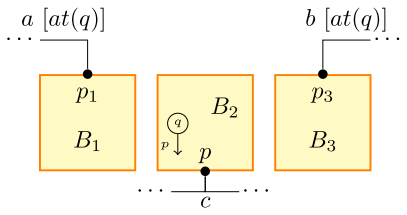

Fig. 6. A composite component with Restriction.

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 behavior is obtained by splitting each transition: one part sends the offer, the other part is triggered by the notification and executes the chosen interaction.

A set of distributed atomic components that send offers and wait for notifications requires a mechanism that receives offers and sends the notifications, accordingly to the offers received and the semantics of the original model. In our implementation, this mechanism consists of one or several additional components. Respecting the semantics of the original model can be described as two tasks:

1. Detect enabled interactions whose Restriction predicate evaluates to true,
2. Resolve conflicts between interactions that involve a common component.

Our solutions rely on Bagrodia's algorithms from [3]. These algorithms use counters to implement mutual exclusion of conflicting interactions. We propose an extended version that encompasses the Restriction operator.

In Section 4.1, we present the different kinds of conflicts that arise between interactions subject to Restriction. We then describe in Section 4.2 how atomic components are modified to send offers and receive notifications. In Section 4.3, we formally describe a solution relying on a single manager and prove its correctness in Section 4.4. Finally, we informally describe how this manager can be decentralized to obtain a 3-layer distributed implementation.

4.1. Conflicts and observation

A conflict appears when two entities compete for a single resource. In our case, the potentially competing entities are the interactions and the resource is the participation of a component. Intuitively, two interactions are conflicting if they involve a common component, which is a participant for at least one of them.

As an example, consider the composite component depicted in Fig. 6. 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.

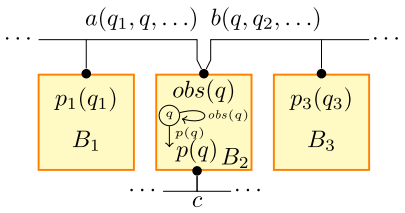

Fig. 7. Observable model obtained from the composite component with Restriction in Fig. 6.

These types of conflicts also appear in transactional memories [31]. 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.

The transformation of the model from Fig. 6 into an observable model, as described in Section 2.4, yields the model depicted in Fig. 7. The Restriction is implemented by adding a new port $obs(q)$ to B_2 and extending interactions a and b to that new port. In this model, B_2 becomes a participant, through the port $obs(q)$ in the interactions a and b. This results in a structural conflict between a and b.

In the context of multiparty interactions, a component either participates in or does not interfere with an interaction. In particular, models where interactions can observe components without modifying them, such as the one in Fig. 6, cannot be directly implemented. The transformation presented in Section 2.4 allows to transform a model with Restriction into a model containing only multiparty interactions. Nevertheless, the obtained distributed implementation involves an unnecessarily high number of exchanged messages: Consider the model presented in Fig. 7. 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 (on the port $obs(q)$). 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. 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 the list of enabled ports through a port, labeled o.

Definition 5 *(Distributed atomic components).* Let $B = (Q, q^0, P, T)$ be an atomic component. The corresponding transformed atomic component is $B^{\perp} = (Q^{\perp}, q^{0^{\perp}}, P^{\perp}, T^{\perp})$, such that:

- $Q^{\perp} = Q \cup \{\perp_q \mid q \in Q\}$ is the union of *stable* states Q and *busy* states $\{\perp_q \mid q \in Q\}$.
- $q^{0^{\perp}} = \perp_{q^0}$ the initial state is the busy state associated to the initial state of the original component.
- $P^{\perp} = P \cup \{o\}$, where o is a new port which publishes the currently enabled ports and the current state of the component.
- The set of transitions $T^{\perp}$ includes, for every transition $\tau = (q, p, q') \in T$:
  1. An *offer* transition $(\perp_q, o, q)$ that goes from a busy to a stable state and publishes the port enabled from this stable state.
  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.

We introduced a new port that publishes offers. The actual offer sent depends on the current state of the component. Given a stable state $q \in Q$, we denote by $\texttt{offer}(q) = \{p \in P \mid \exists q' \in Q, q \xrightarrow{p} q'\}$ the set of ports enabled at state q. In a more concrete implementation, the distributed component exports the set $\texttt{offer}(q)$ and the current state q through the offer port [17].

4.3. Counter-based conflict resolution

In Bagrodia's solutions, the protocol is implemented through one or several managers that receive offers from the atomic components and reply with notifications. We extend these solutions to encompass Restriction predicates.

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 so far. 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 the component B_i.

Furthermore, the manager knows the set of enabled ports and the current state through the offers sent by each component. Thus in order to schedule an interaction, it must check that

1. all ports involved in the interaction are enabled according to the last offers received,
2. the Restriction predicate associated to the interaction evaluates to true according to the last offers, and
3. these offers are still valid according to the n_i and N_i counters.

If these three conditions hold, the interaction can be executed. Upon execution the participation-counts (N_i) associated to the participants are updated to the values of the offer-counts (n_i), so that participants cannot interact again until a new offer is sent and $n_i > N_i$ holds again. In particular, any other interaction that involves (as participants or for observation) one of the participants of the previously executed interaction is blocked until new offers from the conflicting components are sent. Hence, mutual exclusion of conflicting interactions is ensured.

The participation-counts of the observed components are not updated, even if they are checked to ensure that offers satisfying the Restriction predicate are still valid. Thus a component can be "observed" many times, as long as it does not participate in an interaction. We define formally the behavior of the composition of the centralized protocol with the distributed atomic components.

Definition 6. Given a BIP component with Restriction $\rho\gamma(B_1, \ldots, B_n)$ we define the behavior of the adapted counter-based centralized implementation as an infinite state LTS $(Q^\perp, q^0, \gamma^\perp, T^\perp)$ 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 $m_i = (n_i, N_i, \texttt{offer}_i, q_i)$, one for each component B_i, where n_i and N_i are the values of the corresponding counters, $\texttt{offer}_i$ 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.
- $q^0 = ((\perp_{q_1^0}, \ldots, \perp_{q_n^0}), ((0, 0, \emptyset, q_1^0), \ldots, (0, 0, \emptyset, q_n^0)))$. The initial state of system is obtained by taking the initial states of the distributed atomic components, and assigning 0 to each counter in the manager.
- The interactions of $\gamma^\perp$ include the interactions from the original component and the offers:

$$\gamma^\perp = \gamma \cup \bigcup_{i=1}^{n} \{o_i\}$$

- 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 $B_i^\perp$ an offer is enabled: $(q[i], o_i, q_i') \in T_i^\perp$. In that case, $T^\perp$ contains the transition $(q, m) \xrightarrow{o_i} (q', m')$, where:
  – $q'[i] = q_i'$,
  – $m'[i] = (n_i + 1, N_i, \texttt{offer}(q_i'), q_i')$, with $m[i] = (n_i, N_i, \_, \_)$,
  – 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 there is an interaction $a = \{p_i\}_{i \in I_a}$, such that (we denote $m[i] = (n_i, N_i, \texttt{offer}_i, q_i)$):
  – $\forall B_i \in \mathit{participants}(a)$, $p_i \in \texttt{offer}_i$: the interaction is enabled according to the last offers,
  – $\rho_a((q_1, \ldots, q_n))$ evaluates to true according to the values stored in the manager state m,
  – $\forall B_i \in \mathit{participants}(a) \cup \mathit{observed}_\rho(a), n_i > N_i$: the last offers of participants and observed components are still valid.
  Then, the transition $(q, m) \xrightarrow{a} (q', m')$ is in $T^\perp$ with (q', m') defined by:
  – $\forall i \in \mathit{participants}(a)$, $q'[i]$ is the state such that $(q[i], p_i, q'[i]) \in T_i^\perp$,
  – $\forall i \in \mathit{participants}(a)$, $m'[i] = (n_i, N_i + 1, \texttt{offer}_i, q_i)$: counters of participants are incremented.
  – $\forall j \notin \mathit{participants}(a)$, $q'[j] = q[j] \wedge m'[j] = m[j]$.

A global state (q, m) of this protocol clearly separates the state of the components q and the state of the manager m. The enabling of offer transitions depends exclusively on the state of the component sending the offer. Similarly, the enablement

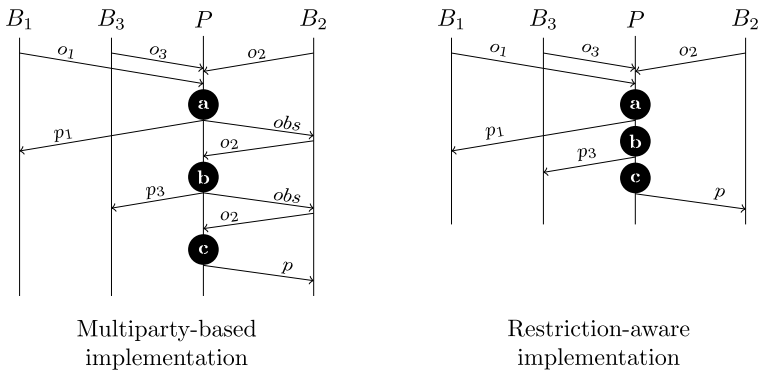

Fig. 8. Exchanges of messages to execute the sequence a, b, c in the model of Fig. 6, for the two implementations.

of execute transitions depends only on the state of the manager. Thus we can assume an asynchronous execution where an offer or an execute transition correspond to a message passing. In the case of the offer transition, the message is sent from the atomic component to the manager. For the execute transition, messages are sent from the manager to each participant in the interaction.

With components where all Restriction predicates are always true, the above construction falls back to the original solution from Bagrodia. We can compare a multiparty-based implementation, obtained by encoding Restriction predicates into multiparty interactions, with a Restriction-aware implementation that directly encompass Restriction predicates.

Example 4. Consider again the model depicted in Fig. 6. We obtain a multiparty-based implementation by transforming it into the model of Fig. 7 and building a distributed implementation from that model. The protocol presented here is able to build a Restriction-aware implementation directly from the model in Fig. 6. In Fig. 8, 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 Restriction-aware implementation. For each process (the distributed components B_i and the protocol P) Fig. 8 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 a 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_2). With the Restriction-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 execution sequence, the Restriction-aware implementation spares 4 messages and increases parallelism since b and c can be launched directly after a, without waiting for a new offer.

4.4. Correctness

We show that the component $\rho\gamma(B_1, \ldots, B_n)$ and the corresponding counter-based implementation are observationally equivalent in the sense of Milner [38]. We first prove the following lemma on the reachable states of the distributed implementation.

Lemma 1. *Let $B_i^\perp$ be a distributed atomic component. The component $B_i^\perp$ is in a stable state q_i iff $n_i > N_i$. Furthermore, if $B_i^\perp$ is in a stable state we have $m_i = (n_i, N_i, \texttt{offer}(q_i), q_i)$.*

Proof. The construction of $B_i^\perp$ implies that it alternates offer and execute transitions. In the initial configuration, $n_i = N_i = 0$ and $B_i^\perp$ is in a busy state. Therefore the equivalence holds.

The only possible transition from this configuration is an offer, which brings the system to a state where $n_i = N_i + 1 > N_i$ and the $B_i^\perp$ is in a stable state. In this configuration, the equivalence holds as well. Furthermore, the offer transition ensures that the offer and state in m_i correspond to those of $B_i^\perp$.

In the configuration where $B_i^\perp$ is in a stable state and $n_i = N_i + 1 > N_i$, the only next possible step in $B_i^\perp$ is an execute action. By executing this step we reach again a configuration where $n_i = N_i$ and $B_i^\perp$ is a busy state. □

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. These interactions correspond to the execute interactions of the distributed implementation, that are also γ. We denote by β the offer interactions.

We define a relation between states Q of the centralized component and states $Q^\perp$ of its distributed implementation. To each state $(q^\perp, m) \in Q^\perp$ of the distributed implementation, we associate a state $e((q^\perp, m)) \in Q$ 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 $e((q^{\perp},m))[i]=q_i$. We say that a state $q\in Q$ and $(q^{\perp},m)\in Q^{\perp}$ are equivalent, denoted by $(q^{\perp},m)\sim q$, if $q=e((q^{\perp},m))$.

Proposition 3 *(Correctness of centralized counter-based implementation). Given a component $\rho\gamma(B_1,\ldots,B_n)$, the labeled transitions systems $(Q,q^0,\gamma,\rightarrow_{\rho\gamma})$ and $(Q^{\perp},q^{0\perp},\gamma^{\perp},\rightarrow_{\perp})$ of its distributed implementation are observationally equivalent.*

Proof. We have to prove that:

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

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

2. The transition $((q^{\perp},m),a,(r^{\perp},m))$ is possible at state $(q^{\perp},m)\in Q^{\perp}$ if for each participant and observed component B_i in the interaction, the counters verify $n_i>N_i$, for each port $p_i\in a$, we have $p_i\in\texttt{offer}_i$, and ρ_a evaluates to true according to the state m of the manager. Lemma 1 ensures that in the equivalent state $q\in Q$, we have for each component B_i participant in a $p_i\in\texttt{offer}(q_i)$. Furthermore, the lemma ensures that for all participant and observed component, the state stored in m is the actual state in $q^{\perp}$, which is equal to q by definition of $\sim$. Therefore $\rho_a(q)$ evaluates to true and $q\xrightarrow{a}_{\rho\gamma} r$. The construction of distributed atomic components ensures that $r\sim(r^{\perp},m)$.

3. If $q\xrightarrow{a}_{\rho\gamma} r$, then for each state $(q^{\perp},m)\sim q$, 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. Let $(s^{\perp},m'')$ be a state such that $(q^{\perp},m)\xrightarrow{\beta^*}_{\perp}(s^{\perp},m'')\not\xrightarrow{\beta}_{\perp}$. Such state is attained when all components have performed their offer transitions. Since offers transitions do not modify any common part of the state, executing them in any order yields the same final state and therefore there is a unique state $(s^{\perp},m'')$ as above. By Lemma 1, since at $(s^{\perp},m'')$ all distributed components are in a stable state, we have for $1\leqslant i\leqslant n$:

- $n_i>N_i$,
- $m_i''=(n_i,N_i,\texttt{offer}(q_i'),q_i')$ where $q_i'=q_i$ by definition of $\sim$.

According to m'', $\rho_a((q_1',\ldots,q_n'))=\rho_a(q)$ evaluates to true and for each port $p_i\in a$, we have $p_i\in\texttt{offer}(q_i)$. Thus $(s^{\perp},m'')\xrightarrow{a}_{\perp}(r^{\perp},m')$. By construction $(r^{\perp},m')\sim r$. □

4.5. 3-layer distributed architecture

With a single manager we obtain a 2-layer implementation, namely the components layer and the manager. In [17], we further decompose the manager in two layers, respectively an interaction layer, and a conflict resolution layer.

The *interaction layer* receives the offers from the components, compute the enabled interactions and evaluates the Restriction predicates. Whenever an interaction is possible, the interaction layer sends a reservation request rsv to the conflict resolution layer for executing the interaction. The request contains the offer-count (n_i) of each component participant in or observed by the interaction. The interaction layer then waits for an answer from the conflict resolution layer. If the answer is positive (ok), the interaction is executed and the interaction layer sends a notification to each participant in the interaction. Otherwise the answer is f (fail), and the interaction protocol waits for a new offer before trying a new reservation.

The *conflict resolution layer* receives the requests along with the offer-count. It is also responsible for maintaining the participation-count N_i. Upon reception of a request, it compares the received n_i with the maintained N_i. If for all involved components $n_i>N_i$, then all for all participants the value of N_i is updated to n_i and a positive answer ok is sent back. Otherwise a negative answer f (fail) is sent back.

As shown in Fig. 9, the interaction layer can be separated in several processes (here IP_1,IP_2,IP_3), in order to allow concurrent execution of interactions. The transformation from a BIP model into a 3-layer implementation is actually parameterized by a partition of the interaction. Each class of the partition yields a distinct process in the final implementation. For our example, the partition is $\{\{upgrade,reboot\},\{boot,halt\},\{request,acknowledge\}\}$. Thus in our case, the process IP_1 is responsible for detecting whenever the interaction *reboot* or the interaction *upgrade* is enabled.

If there are several processes handling interactions, each component maintain its n_i counter and send it with the offer. The components send offer only to processes handling interactions in which they are observed or participate. If some interaction conflicts only with interactions that are handled in the same process, the conflict can be resolved locally. Thus,

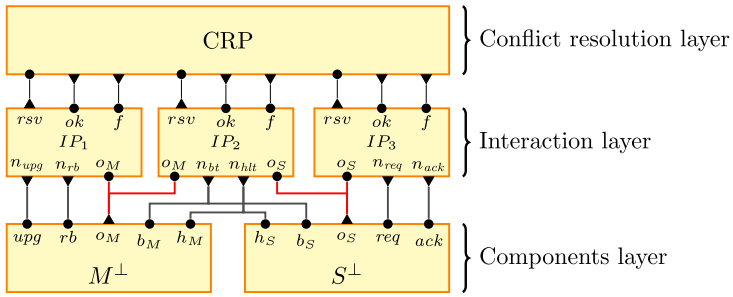

Fig. 9. 3-layer distributed implementation of component from Fig. 2.

taking a coarser partition reduces the parallelism between execution of interactions but also reduce the load on the conflict resolution layer as more conflicts can be solved internally.

Finally, the conflict resolution layer can be decentralized as well. Intuitively, the key property to ensure tis correctness is the atomicity of the operation consisting in checking that the offer-counts are greater than the participation-counts and updating the latter ones if true. Following Bagrodia [3], we propose three implementations:

- A centralized implementation, where a single process receives all the requests. In that case, atomicity is ensured as only this process can access the N_i variables.
- A token ring implementation, where each interaction yields a separate process in the conflict resolution layer. Each process receives only the requests for executing the interaction it handles. A token carrying all the N_i variables circles through all processes in the layer. Only the process holding the token can access these variables, which ensures the needed atomicity.
- An implementation based on a solution to the dining philosophers problem [20]. This is the most decentralized solution. Each interaction yields a process in the conflict resolution layer. Interactions are mapped to philosophers, and forks represent conflicting components. Hence, two processes corresponding to two conflicting interactions share a fork, that is a token with the N_i variables of the components causing the conflict. In order to execute an interaction, each process must acquire all forks shared with its neighbors. Then it is the only one to access the needed N_i variables which ensures atomicity. Using a solution to the dining philosopher problem ensures that any process will eventually be able to acquire all the forks shared with its neighbors (no starvation).

5. Experiments

We compare the execution time and the number of exchanged messages for several distributed implementations of a component with priority. As shown in Fig. 1, several sequences of transformations and optimizations can be applied to generate a distributed implementation. The first step (Transformation 1 in Fig. 1) involves transformation of this component into a component with Restriction. The obtained component may be optimized using knowledge as explained in Section 3 (Transformation 2 in Fig. 1). Whether the component is optimized or not, we consider the two following sequences of transformations leading to a distributed implementation.

- Transform the component with Restriction into a component with only interactions as explained in Section 2.4. Then generate a 3-layer distributed model embedding the conflict resolution protocol described in Section 4.3, which in this case falls back to the original version by Bagrodia. This method (Transformations 3 and 4 in Fig. 1) results in a multiparty-based implementation.
- Directly transform the component with Restriction into a 3-layer distributed model embedding the conflict resolution protocol described in Section 4.3. This method (Transformation 5 in Fig. 1) results in a Restriction-aware implementation.

For both cases, 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 Fig. 10. 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 potentially requires to check that *all* "*eat*" interactions are disabled. This check requires observing all components. This example compares the above implementations under strong priority constraints.

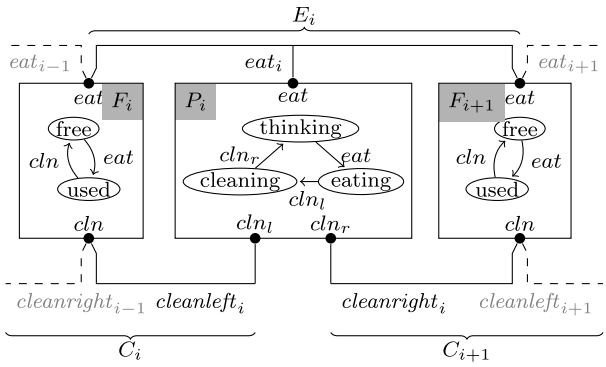

Fig. 10. Fragment of the dining philosopher component. Braces indicate how interactions are grouped into interaction protocols.

Table 1
Minimal number of observations for ensuring completeness in the interaction protocol C_0.

Component	*Size*	*True*	*BI*	*LI*	*Optimal*
Philo3	6	3	3	1	1
Philo4	8	5	5	2	2
Philo5	10	7	7	3	3
Philo10	20	17	17	8	8
Philo20	40	37	37	18	18
Philo100	200	197	197	108	98

As explained in Section 4.5, 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.

5.1.1. Minimizing observed components

We first define the quantity that we minimize. In the distributed implementation, each atomic component sends its state to interactions that are observing it. If interactions a and b, both observing a component B_i, are handled by the same interaction protocol, the component B_i sends only one message to that interaction protocol. In that case, we say that the atomic component is observed by the interaction protocol. We count as an observation each couple (component, interaction protocol) such that the interaction protocol observes the component, and the component is not involved in any interaction handled by the interaction protocol.

Minimizing the number of observations in a complete Restriction ρ' is done independently for each interaction protocol. Table 1 shows the results, that is the number of observations involving the interaction protocol C_0 in the solution obtained with the heuristic described in Section 3.4.1. Note that same number of observations are needed for each other interaction protocol C_i. The total number of atomic components in the composite component is indicated in column *Size*. The columns *true*, *BI* and *LI* provide the cost of the solutions obtained when using respectively *true*, the boolean invariant and the linear invariant as over-approximation of the global states. Using *true* as invariant does not allow actual optimization, therefore it shows the number of observations in the initial Restriction ρ. The column *optimal* indicates the cost of an optimal solution, that we know for this particular example.

Here, the linear invariant gives better results than the boolean invariant, which is not precise enough to allow reducing observation comparatively to the *true* invariant. For $N = 3$, we provide the boolean and linear invariants respectively in Figs. 11 and 12. As an example, consider the linear constraint (15). It ensures that interaction $cleanleft_0$ and interaction eat_1 cannot be enabled concurrently, otherwise, control locations $P_0.eating$ and $F_1.free$ would be active and the sum in constraint (15) would be equal to 2. Thus, the priority $cleanleft_0 <_\pi eat_1$ never forbids execution of $cleanleft_0$. A related boolean constraint, that is constraint (3) of boolean invariant guarantees that at least one of these locations is active. However, this constraint is not strong enough to discard the case where two of them are active.

In general, the approximations of reachable states provided by boolean and linear invariants are not comparable. Consider the global state $P_0.cleaning \wedge F_0.used \wedge P_1.cleaning \wedge F_1.used \wedge P_2.cleaning \wedge F_2.used$. This state satisfies all the constraints of the linear invariant, but does not satisfy the constraint 8 of the boolean invariant.

The results for computing basic solutions are presented in Table 2. The column *Size* contains the total number of atomic components in the composite component. The columns *true*, *BI* and *LI* contains respectively the cost of the solutions obtained when using respectively *true*, the boolean invariant and the linear invariant. For Philo3, baseness is achieved when each engine observes only the components involved in the interactions it handles (i.e. no additional observation is needed), therefore the cost is 0.

$$\forall i \in \{0,1,2\}\ (at(F_i.free) \vee at(F_i.used)) \quad (1)$$
$$\wedge\ \forall i \in \{0,1,2\}\ (at(P_i.thinking) \vee at(P_i.eating) \vee at(P_i.cleaning)) \quad (2)$$
$$\wedge\ (at(P_1.eating) \vee at(P_0.eating) \vee at(P_0.cleaning) \vee at(F_1.free)) \quad (3)$$
$$\wedge\ (at(P_2.eating) \vee at(P_1.eating) \vee at(P_1.cleaning) \vee at(F_2.free)) \quad (4)$$
$$\wedge\ (at(P_0.thinking) \vee at(F_0.used) \vee at(P_0.cleaning) \vee at(P_2.thinking)) \quad (5)$$
$$\wedge\ (at(P_0.thinking) \vee at(F_1.used) \vee at(P_1.cleaning) \vee at(P_1.thinking)) \quad (6)$$
$$\wedge\ (at(P_2.cleaning) \vee at(F_0.free) \vee at(P_2.eating) \vee at(P_0.eating)) \quad (7)$$
$$\wedge\ (at(F_1.free) \vee at(F_2.free) \vee at(F_0.free) \vee at(P_1.eating) \vee at(P_2.eating) \vee at(P_0.eating)) \quad (8)$$
$$\wedge\ (at(F_2.used) \vee at(P_2.cleaning) \vee at(P_1.thinking) \vee at(P_2.thinking)) \quad (9)$$
$$\wedge\ (at(F_2.used) \vee at(P_2.cleaning) \vee at(P_1.thinking) \vee at(F_0.free) \vee at(P_0.eating)) \quad (10)$$
$$\wedge\ (at(F_1.free) \vee at(P_1.eating) \vee at(F_0.used) \vee at(P_0.cleaning) \vee at(P_2.thinking)) \quad (11)$$
$$\wedge\ (at(P_0.thinking) \vee at(F_2.free) \vee at(F_1.used) \vee at(P_2.eating) \vee at(P_1.cleaning)) \quad (12)$$

Fig. 11. Boolean invariant for the dining philosophers example with $N = 3$.

$$(at(P_0.thinking) + at(P_0.eating) + at(P_0.cleaning) = 1) \quad (13)$$
$$\wedge\ \forall i \in \{0,1,2\}(at(F_i.free) + at(F_i.used) = 1) \quad (14)$$
$$\wedge\ (at(P_1.eating) + at(P_0.eating) + at(P_0.cleaning) + at(F_1.free) = 1) \quad (15)$$
$$\wedge\ (at(P_1.thinking) - at(P_0.eating) - at(P_0.cleaning) + at(F_1.used) + at(P_1.cleaning) = 1) \quad (16)$$
$$\wedge\ (at(P_2.eating) - at(P_0.eating) - at(P_0.cleaning) + at(F_1.used) + at(P_1.cleaning) - at(F_2.used) = 0) \quad (17)$$
$$\wedge\ (at(P_2.cleaning) + 2 * at(P_0.eating) + at(P_0.cleaning) - at(F_1.used) - at(P_1.cleaning) + at(F_2.used) - at(F_0.used) = 0) \quad (18)$$
$$\wedge\ (at(P_2.thinking) - at(P_0.eating) + at(F_0.used) = 1) \quad (19)$$

Fig. 12. Linear invariant for the dining philosophers example with $N = 3$.

Table 2
Minimal number of observations for ensuring baseness in the whole model.

Component	*Size*	*True*	*BI*	*LI*
Philo3	6	9	9	0
Philo4	8	20	20	4
Philo5	10	35	35	6
Philo10	20	170	170	23

5.1.2. Comparing obtained implementations

The goal of this subsection is to compare the different implementations that we obtained for the dining philosophers example. First, we consider different levels of optimization for the Restriction operator:

- **No optimization**: the Restriction operator is the direct rewriting of priorities rules, we do not apply any knowledge-based optimization (Transformation 2 in Fig. 1).
- **Basic**: observation required by the Restriction operator is minimized while still ensuring baseness.
- **Complete**: observation required by the Restriction operator is minimized while still ensuring completeness.

As showed in the previous subsection, the Boolean invariant is not strong enough to reduce the number of observed components comparatively to the non-optimized version. Therefore, the basic and complete version of the Restriction operator have been computed using the linear invariant. For each optimization level considered, we generate a multiparty-based (MB) and a Restriction-aware (RA) implementation. 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.

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 20 runs. The number of interactions executed by each implementation is presented in Fig. 13. The total number of messages exchanged for the execution of each implementation is presented in Fig. 14.

First, remark that switching from a multiparty-based (gray) to a Restriction-aware (black) implementation improves performance, that is the number of interactions executed in 60 seconds. Furthermore, it always reduces the number of messages exchanged. The improvement is very visible with the unoptimized version (No opt). This can be explained as follows. Evaluating Restriction predicates 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. After a "*clean*" interaction, only components that participated may need to send a new offer before another "*clean*" interaction can be executed. This explains the speedup.

Second, when comparing multiparty-based (gray) implementations, one sees that the Restriction operator ensuring completeness gives the best performance. The basic implementations exhibit poor performance because restricting observation also restricts parallelism in that case. For the example with 9 philosophers, multiparty-based implementation with optimized Restriction (Complete – MB) shows a significant gain in performance compared to the non-optimized version (No opt – MB). The performance gained by optimizing the Restriction operator into a complete one is not visible anymore when

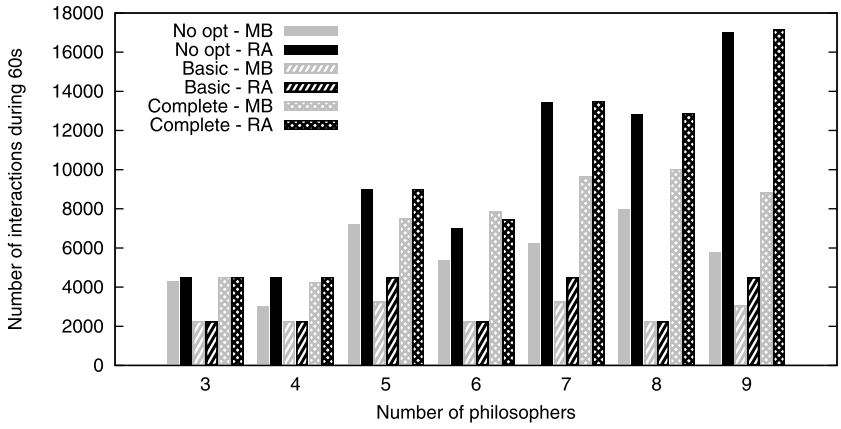

Fig. 13. Number of interactions executed in 60 s for different implementations of the dining philosophers example. MB: Multiparty-based. RA: Restriction-aware. More interactions = better performance.

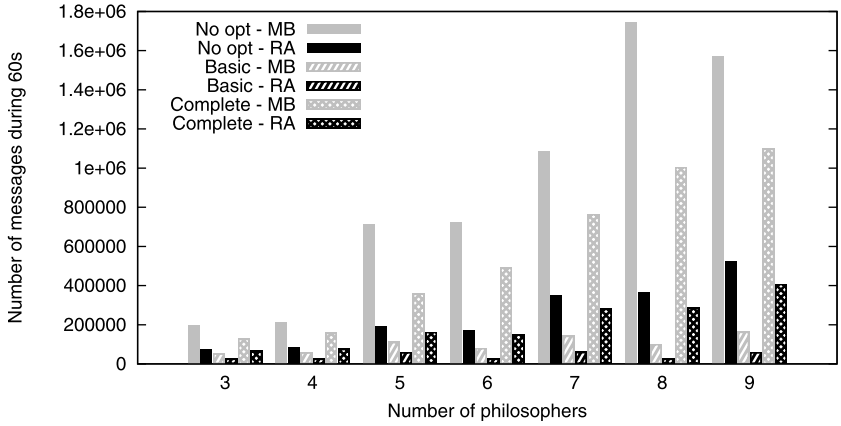

Fig. 14. Number of messages exchanged in 60 s for different implementations of the dining philosophers example. MB: Multiparty-based. RA: Restriction-aware. Fewer messages = more efficient implementation.

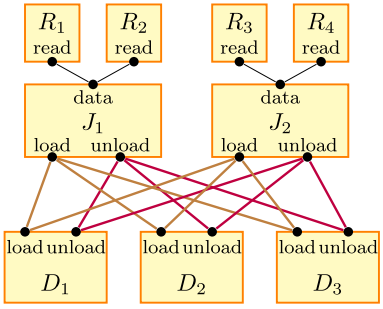

Fig. 15. Jukebox component with 3 disks.

switching to Restriction-aware (black) implementation. However, the optimization remains interesting in that case since it reduces by up to 10% the number of messages needed.

5.2. Jukebox

The second example is a jukebox depicted in Fig. 15. It represents a system, where a set of readers $R_1 \dots R_4$ access data located on 3 disks D_1, D_2, D_3. Readers may need to access any disk. 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 $read_j$

Table 3
Minimal observation cost to ensure baseness or completeness.

Interaction	*True*	*BI* (basic)	*BI* (complete)	*LI* (basic)	*LI* (complete)
$unload_{i,k}$	5	3 ($k=1$) or 5 ($k=2$)	5	2	2
$load_{i,k}$	1	0	1	0	1

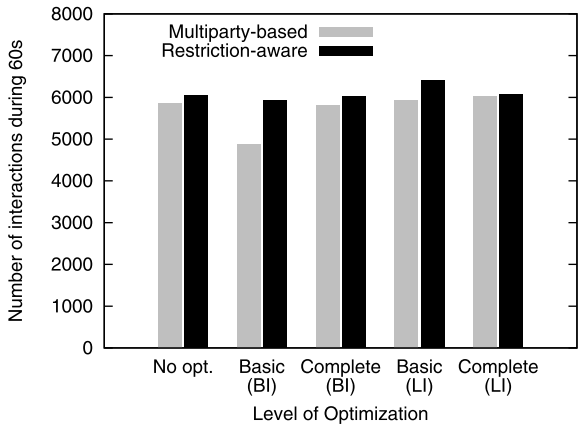

Fig. 16. Number of interactions executed in 60 s for the jukebox example. More interactions = better performance.

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.

The main difference with the dining philosopher examples is that here priority rules do not restrict parallelism since they are expressed between structurally conflicting interactions. 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. Removing this priority rule results in a system that does less "*read*" interactions.

5.2.1. Minimizing observed components

The results of the simulated annealing heuristic are presented in Table 3. Interaction protocols handling a "read" interaction do not need to observe additional atomic components since there is no interaction with higher priority. The boolean invariant allows removing some observed atomic components, in the basic solution. As for Philo$N$ components, the linear invariant is stronger than the boolean invariant. Therefore, attaining the same level of detection requires less observed atomic components.

5.2.2. Comparing obtained implementations

For this example, the Boolean invariant (BI) provides enough information to reduce the observation. We consider the optimization levels: No optimization, Basic ($\mathfrak{I}$), and Complete ($\mathfrak{I}$), where $\mathfrak{I}$ is either the boolean invariant BI or the linear invariant LI. For each optimization level, we compare multiparty-based and Restriction-aware implementations. The number of interactions executed during 60 seconds is presented in Fig. 16. Here the performance of the Restriction-aware implementation is not significantly better than the performance of the multiparty-based implementation. The best results are obtained with the basic optimization level using linear invariant. These results come from the fact that no parallelism is allowed between low priority interactions since they are structurally conflicting. Therefore patterns enabling parallelism as in Fig. 8 do not arise. More precisely, the only gain in performance consists in time involving actually sending and receiving messages, not in waiting unneeded offers.

Fig. 17 shows that significantly fewer messages are exchanged with the Restriction-aware implementation. Intuitively, this difference corresponds to the notifications and subsequent offers to and from observed components, that are not necessary with the Restriction-aware implementation. Interestingly, the implementation giving the best performance (Basic (LI) optimization with Restriction-aware implementation) is also the one requiring the least number of messages.

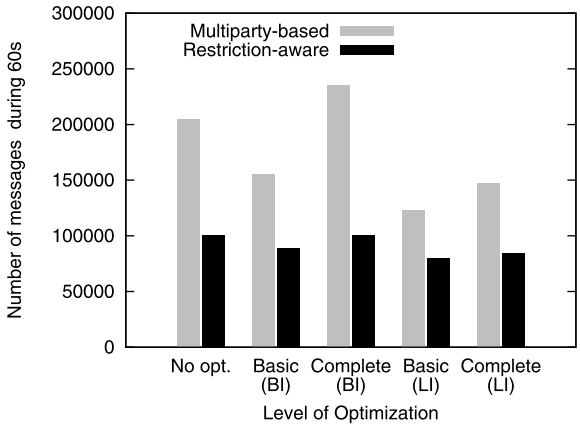

Fig. 17. Number of messages exchanged in 60 s for the jukebox example. Fewer messages = more efficient implementation.

6. Related work

Conflict resolution for multiparty interactions. Distributed conflict resolution boils down to solving the *committee coordination problem* [20], 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 [20,3,39,41], a circulating token [35], or a randomized algorithm without managers [32].

The solutions provided by Bagrodia [3] rely on counters to ensure mutual exclusion of conflicting interactions, as explained in Section 4. The token ring and dining philosopher-based solutions allow the designer to use an arbitrary partitions of the interactions for building the different managers. Our solution differs as we separate the conflict resolution from the execution of the interactions.

The α-core protocol [41] builds one manager (called coordinator) per interaction. Contrarily to Bagrodia's solution, this solution does not rely on counters. The principle of the protocol is that each coordinator locks sequentially the components according to a global order. Each component can be locked by only one coordinator. If the coordinator manages to lock all components of the interaction, it executes the interaction. Otherwise, the coordinator frees all components locked so far. As there are no counters, every component must explicitly withdraw all unsuccessful offers when executing an interaction. Furthermore, it must wait for an acknowledgment of the withdrawal before resuming execution, which may incur an overhead compared to a counter-based solution. In [11], knowledge is used to optimize the α-core protocol.

The solution by Kumar [35] implements an idea similar to the α-core protocol, without managers. Each interaction is represented by a token. There is a global order on the components, and to execute an interaction, the token must traverse (and lock) all the components according to the global order. If a token arrives in a component that has already been locked by another token (i.e. a conflicting interaction), it waits until the interaction either succeeds or fail. If the conflicting interaction fails, the token is propagated. If the conflicting interaction succeeds, the token is stopped and all components it already traversed are unlocked.

The solution provided by Joung and Smolka in [32] mainly focuses on ensuring fairness between interactions. This algorithm relies on randomization: each component picks an interaction and send a message to all participants. If one of the participants detects that at a given point all participants agree on the interaction, the latter is executed. Otherwise, each component picks another interaction.

Other frameworks for process coordination. Although we use multiparty interactions to describe communication between processes, many solutions enhancing message-passing exist. The MPI framework [28] provides collective operations such as `MPI_Barrier()` that implements strong synchronization of a set of processes. However, conflicting interactions are not handled, the programmer has to ensure that all involved processes commit to the same synchronization.

In [26], German presents a framework providing multiparty interactions with priorities as primitives. The processes are described using a notation similar to the CCS. Interactions are specified as composite action labels made of conjunctions and negations of actions. For instance, the interaction *reboot* of the example from Fig. 2 could be encoded as the action label $rb \wedge \neg req \wedge \neg ack$, assuming that BIP ports are mapped to simple actions. Encoding an interaction requires an additional process, that contains a single rule with the corresponding action label. The idea is to provide a language suitable for specifying distributed systems, with a high-level description that can be executed for rapid prototyping. This framework was used to model and verify a telephone switching application. To our knowledge, there is no distributed implementation for this framework, although it was apparently one of the goals in [26].

Reo [2] is a framework where components communicates using a basic set of dataflow connectors that are combined to form a complex connector. At each round, each component enables a set of input and output ports. In Reo, components are

black boxes, only their interface is known at each state. A Reo connector defines a set of allowed dataflow interactions for each configuration of the enabled ports. A round consists of executing such an interaction, which transfers data. Furthermore, Reo basic connectors include $FIFO_1$ connectors that can store one data item, allowing the composed connector to save some data between two rounds. The FIFO connectors introduce a control state in the composed connector, which differs from BIP where interactions have no memory. The Dreams framework [42,43] provides a distributed implementation for Reo. Each basic connector is implemented as an actor. A round synchronizes the actors through a consensus algorithm, in order to choose the next interaction. In order to achieve a more decentralized behavior, a GALS architecture is obtained by cutting the complex connector into synchronous regions. Two regions can be separated if their border consists only of FIFO connectors.

I/O automata [36] were introduced to formally model distributed systems. In this framework, each process is represented by an automaton whose transitions are labeled by actions, similarly to processes from this chapter. Each interaction is represented through a common label that is used in several processes to denote synchronization. I/O automata clearly distinguish between input (uncontrollable) actions and output (controllable) actions. Given an interaction label, there is exactly one process for which this label is an output action, in other processes it can appear only as an input action. Furthermore, from every state of an automaton, all its input actions are required to be enabled. With these restrictions, an interaction is completely controlled by the process for which it is an output action. In that sense, I/O automata interactions are similar to Send/Receive interaction where the sender controls the execution and the receiver should not block the sender. In particular, the fact that an interaction is enabled or not is local to the process that controls the corresponding output action. In that sense, there is no conflict between interactions. However, if two interactions a and b are scheduled simultaneously by two separate processes, the order should be consistent among all common participants in a and b. The solution proposed in the first sketch of a distributed implementation [24] is to require that each automaton reaches the same state for both orderings. In a later solution [25,45], this problem is solved by adding a handshake protocol.

In [23,21], *Synchronizers* are used to filter incoming messages for a set of actors. A message is delivered only if it matches an enabled pattern. Such patterns include atomic synchronization of a set of messages, that requires all involved messages to be pending before granting their transmission. According to [21], synchronizers are implemented through dispatchers located on the target actors, that is the actors for which incoming messages are filtered. Upon reception of an incoming message, the dispatcher is responsible for checking whether the message is allowed for transmission according to the synchronizers. In case of atomic synchronization, this requires a protocol similar to the one for multiparty interactions. The actors communicate through asynchronous message-passing, which makes it difficult to exploit the synchronization of messages for verification purposes. This framework is mainly concerned with providing practical constructs for programming with actors.

Behavioral programming [37] is another model for programming interactions between processes. In that model, at each global state, each process provides three sets of actions: requested actions, watched actions and blocked actions. A (centralized) scheduler selects an action that is requested by at least a process and not blocked by any process. The selected action is executed by all processes that requested it and all processes that watched it. The system reaches the next global state by executing the selected action. This model differs from multiparty interactions as the set of participants in the common action is not fixed, but depends on the state. Decentralizing the scheduler while preserving centralized semantics requires to solve problems similar to Interaction and Restriction conflicts resolution. In particular, scheduling an action based on a partial set of offers requires to ensure that this action will not be blocked by a subsequent offer.

Knowledge. The formalization of different kinds of knowledge and related logic, called *epistemic logic*, have been intensively studied [22,29,30]. In rough sets theory [40], objects are defined through a set of attributes. Intuitively, an attribute can be the shape, the color, the weight… of the object. Each object is fully identified by the definition of its attributes. Given a subset of the objects, deciding whether a given object is part of that subset is always achieved by observing all attributes. Restricting the set of attributes that are observed creates a rough set, that approximates the subset. In that case indeed, there could be some objects whose membership in the subset depends on an unobservable attribute. One of the question in this theory is to find a minimal set of attributes whose observation is sufficient to distinguish any two objects [47].

In [44], Knowledge is applied to decentralized control of a plant. A plant is an automaton whose interactions are labeled by actions, some of them being forbidden. Multiple decentralized controllers are in charge of controlling the plant, through allowing or not a given subset of the actions. Each controller is defined by the set of actions it can observe and the set of actions it can control (i.e. execute). Knowledge is applied to allow each controller to infer which actions are legal from the current state and thus can be executed. An extension to distributed knowledge is proposed, whenever the information available to only one controller is not enough to decide. A criterion, called “Kripke observability” decides whether the extension to distributed knowledge is enough to control the plant.

In [8,4], the focus is on distributed controllers for executing Petri nets constrained by a given property. An example of such a constraining property is a priority order. Processes are defined as sets of Petri nets transitions. A transition can be common to several processes, in which case it describes a synchronization. Each process can observe its neighborhood, that is the places that are adjacent to its transitions. In [4], Knowledge is used to build a support table for each process. This table indicates, for each local configuration, which interaction can be safely executed. Knowledge based on the state of the neighborhood is not always sufficient, two possible extensions are proposed. The first one consists in using knowledge with perfect recall. The second one, also proposed in [27] consists in accumulating knowledge through additional synchronizations between processes. This additional synchronization is handled by a multiparty interaction protocol; α-core [41] is

proposed. In [8], an optimization is proposed by considering only executions satisfying the constraining property in order to build the knowledge. This approach allows reducing the state space and possibly increases the knowledge of each process.

7. Conclusion

We proposed different methods for generating a distributed implementation from a model described using multiparty interactions with Restriction. 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 priorities which are essential for modeling scheduling policies. We have proposed a transformation leading from a model with Restriction into an equivalent model with interactions. The transformation consists in creating events making visible state-dependent conditions.

Components whose state is needed to evaluate the Restriction predicate associated to an interaction are observed by this interaction. A synchronized up-to-date view of the observed components that satisfy the predicate is needed to launch the execution of the interaction. We proposed methods, based on the work in [12], to compute a new Restriction with a reduced number of observed components. The obtained Restriction ensures either deadlock-freedom preservation or observational equivalence with the original model.

Expressing Restriction by interactions allows the application of existing distributed implementation techniques, such as the one presented in [17]. We have proposed an optimization of the conflict resolution algorithm from [3] that takes into account the fact that an observed component does not actively participate in the interaction. Preliminary experiments compare the performance and communication volume of the implementation obtained with various optimization levels. They show significant performance improvement when using the optimized conflict resolution algorithm.

Future work includes further optimization of the conflict resolution protocol through Knowledge, as in [11]. It also includes generating distributed models with timing constraints as in [1]. In particular, the multi-threaded implementation in [46], where all timing constraints are handled by a single thread, could be extended to a fully distributed implementation. Furthermore, Knowledge could be applied as well to determine for instance whether an interaction protocol has to wait for a particular message.

References

[1] T. Abdellatif, J. Combaz, J. Sifakis, Model-based implementation of real-time applications, in: L.P. Carloni, S. Tripakis (Eds.), EMSOFT, ACM, 2010, pp. 229–238.
[2] F. Arbab, Reo: a channel-based coordination model for component composition, Math. Struct. Comput. Sci. 14 (2004) 329–366.
[3] R. Bagrodia, Process synchronization: Design and performance evaluation of distributed algorithms, IEEE Trans. Softw. Eng. 15 (9) (1989) 1053–1065.
[4] A. Basu, S. Bensalem, D. Peled, J. Sifakis, Priority scheduling of distributed systems based on model checking, Form. Methods Syst. Des. 39 (3) (2011) 229–245.
[5] A. Basu, P. Bidinger, M. Bozga, J. Sifakis, Distributed semantics and implementation for systems with interaction and priority, in: Formal Techniques for Networked and Distributed Systems (FORTE), 2008, pp. 116–133.
[6] A. Basu, M. Bozga, J. Sifakis, Modeling heterogeneous real-time components in BIP, in: Software Engineering and Formal Methods (SEFM), 2006, pp. 3–12.
[7] I. Ben-Hafaiedh, S. Graf, S. Quinton, Building distributed controllers for systems with priorities, J. Log. Algebr. Program. 80 (2011) 194–218.
[8] S. Bensalem, M. Bozga, S. Graf, D. Peled, S. Quinton, Methods for knowledge based controlling of distributed systems, in: Automated Technology for Verification and Analysis – 8th International Symposium, ATVA 2010, Proceedings, in: Lect. Notes Comput. Sci., vol. 6252, Springer, September 2010, pp. 52–66.
[9] S. Bensalem, M. Bozga, A. Legay, Thanh-Hung Nguyen, J. Sifakis, Rongjie Yan, Incremental component-based construction and verification using invariants, in: Formal Methods in Computer-Aided Design (FMCAD), Oct. 2010, pp. 257–266.
[10] S. Bensalem, M. Bozga, T.-H. Nguyen, J. Sifakis, D-finder: A tool for compositional deadlock detection and verification, in: Computer Aided Verification, in: Lect. Notes Comput. Sci., vol. 5643, 2009, pp. 614–619.
[11] S. Bensalem, M. Bozga, D. Peled, J. Quilbeuf, Knowledge based transactional behavior, in: Armin Biere, Amir Nahir, Tanja Vos (Eds.), Hardware and Software: Verification and Testing, in: Lect. Notes Comput. Sci., vol. 7857, Springer, Berlin, Heidelberg, 2013, pp. 40–55.
[12] S. Bensalem, M. Bozga, J. Quilbeuf, J. Sifakis, Knowledge-based distributed conflict resolution for multiparty interactions and priorities, in: FMOODS/FORTE, 2012, pp. 118–134.
[13] S. Bensalem, M. Bozga, J. Quilbeuf, J. Sifakis, Optimized distributed implementation of multiparty interactions with observation, in: Proceedings of the 2nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions, AGERE! '12, ACM, New York, NY, USA, 2012, pp. 71–82.
[14] S. Bensalem, M. Bozga, J. Sifakis, T.-H. Nguyen, Compositional verification for component-based systems and application, in: ATVA, Berlin, Heidelberg, 2008.
[15] S. Bensalem, D. Peled, J. Sifakis, Knowledge based scheduling of distributed systems, in: Essays in Memory of Amir Pnueli, 2010, pp. 26–41.
[16] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, J. Sifakis, From high-level component-based models to distributed implementations, in: EMSOFT, 2010, pp. 209–218.
[17] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, J. Sifakis, A framework for automated distributed implementation of component-based models, Distrib. Comput. 25 (5) (2012) 383–409.
[18] B. Bonakdarpour, M. Bozga, J. Quilbeuf, Automated distributed implementation of component-based models with priorities, in: EMSOFT, 2011, pp. 59–68.
[19] K.M. Chandy, J. Misra, The drinking philosophers problem, ACM Trans. Program. Lang. Syst. 6 (4) (1984) 632–646.
[20] K.M. Chandy, J. Misra, Parallel Program Design: A Foundation, Addison–Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.
[21] P. Dinges, G. Agha, Scoped synchronization constraints for large scale actor systems, in: Proceedings of the 14th International Conference on Coordination Models and Languages, COORDINATION'12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 89–103.
[22] R. Fagin, J.Y. Halpern, Y. Moses, M.Y. Vardi, Reasoning About Knowledge, MIT Press, 1995.

[23] S. Frølund, G. Agha, A language framework for multi-object coordination, in: Proceedings of ECOOP, Springer-Verlag, 1993, pp. 346–360.
[24] S.J. Garland, N. Lynch, Using I/O automata for developing distributed systems, in: Foundations of Component-Based Systems, Cambridge University Press, New York, NY, USA, 2000, pp. 285–312.
[25] C. Georgiou, N. Lynch, P. Mavrommatis, J.A. Tauber, Automated implementation of complex distributed algorithms specified in the IOA language, Int. J. Softw. Tools Technol. Transf. 11 (2) (2009) 153–171.
[26] S.M. German, Programming in a general model of synchronization, in: Rance Cleaveland (Ed.), CONCUR, in: Lect. Notes Comput. Sci., vol. 630, Springer, 1992, pp. 534–549.
[27] S. Graf, D. Peled, S. Quinton, Achieving distributed control through model checking, Form. Methods Syst. Des. 40 (2) (April 2012) 263–281.
[28] W. Gropp, E. Lusk, R. Thakur, Using MPI-2: Advanced Features of the Message Passing Interface, MIT Press, 1999.
[29] J.Y. Halpern, R. Fagin, Modelling knowledge and action in distributed systems, Distrib. Comput. 3 (4) (1989) 159–177.
[30] J.Y. Halpern, Y. Moses, Knowledge and common knowledge in a distributed environment, J. ACM 37 (July 1990) 549–587.
[31] M. Herlihy, J.E.B. Moss, Transactional memory: architectural support for lock-free data structures, Comput. Archit. News 21 (2) (May 1993) 289–300.
[32] Y.-J. Joung, S.A. Smolka, Strong interaction fairness via randomization, IEEE Trans. Parallel Distrib. Syst. 9 (2) (1998) 137–149.
[33] S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi, Optimization by simulated annealing, Science 220 (4598) (1983) 671–680.
[34] F. Krückeberg, M. Jaxy, Mathematical methods for calculating invariants in Petri nets, in: Advances in Petri Nets 1987, in: Lect. Notes Comput. Sci., vol. 266, Springer, Berlin/Heidelberg, 1987, pp. 104–131.
[35] D. Kumar, An implementation of n-party synchronization using tokens, in: ICDCS, 1990, pp. 320–327.
[36] N.A. Lynch, M.R. Tuttle, An introduction to input/output automata, CWI Quart. 2 (1989) 219–246.
[37] A. Marron, G. Weiss, G. Wiener, A decentralized approach for programming interactive applications with JavaScript and blockly, in: Proceedings of the 2nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions, AGERE! '12, ACM, New York, NY, USA, 2012, pp. 59–70.
[38] R. Milner, Communication and Concurrency, Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1995.
[39] J. Parrow, P. Sjödin, Multiway synchronization verified with coupled simulation, in: International Conference on Concurrency Theory (CONCUR), 1992, pp. 518–533.
[40] Z. Pawlak, A. Skowron, Rudiments of rough sets, Inf. Sci. 177 (1) (2007) 3–27.
[41] J.A. Pérez, R. Corchuelo, M. Toro, An order-based algorithm for implementing multiparty synchronization, Concurr. Comput., Pract. Exp. 16 (12) (2004) 1173–1206.
[42] J. Proença, Synchronous coordination of distributed components, PhD thesis, Leiden University, 2011.
[43] J. Proença, E. Clarke, D. de Vink, F. Arbab, Dreams: a framework for distributed synchronous coordination, in: Proceedings of the 27th Annual ACM Symposium on Applied Computing, SAC'12, ACM, New York, NY, USA, 2012, pp. 1510–1515.
[44] S.L. Ricker, K. Rudie, Know means no: Incorporating knowledge into discrete-event control systems, IEEE Trans. Autom. Control 45 (9) (2000) 1656–1668.
[45] J.A. Tauber, Verifiable compilation of I/O automata without global synchronization, PhD thesis, Massachusetts Institute of Technology, 2005.
[46] A. Triki, J. Combaz, S. Bensalem, J. Sifakis, Model-based implementation of parallel real-time systems, in: Vittorio Cortellessa, Dániel Varró (Eds.), FASE, in: Lect. Notes Comput. Sci., vol. 7793, Springer, 2013, pp. 235–249.
[47] J. Zhou, D. Miao, Q. Feng, L. Sun, Research on complete algorithms for minimal attribute reduction, in: Peng Wen, Yuefeng Li, Lech Polkowski, Yiyu Yao, Shusaku Tsumoto, Guoyin Wang (Eds.), Rough Sets and Knowledge Technology, in: Lect. Notes Comput. Sci., vol. 5589, Springer, Berlin/Heidelberg, 2009, pp. 152–159.