Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

Building Distributed Controllers for Systems with Priorities

Imene Ben-Hafaiedh, Susanne Graf and Sophie Quinton

Verimag Research Report n^o TR-2010-15

4-06-2010

Building Distributed Controllers for Systems with Priorities

Imene Ben-Hafaiedh, Susanne Graf and Sophie Quinton

4-06-2010

Abstract

Composition of components by means of multi-party interactions allows specifying intended global properties in a very abstract manner useful in many application domains. Composition by multi-party interactions allows guaranteeing most safety properties "by construction", but deadlock freedom must generally be checked for. In this paper, we propose an algorithm that — if necessary — constructs a memoryless orchestrator given by a set of priority rules which enforce deadlock freedom.

In the context of distributed systems, such as webservices, the resulting prioritized system must later be executed in a distributed fashion. We present here a new algorithm that allows executing systems with (binary) interactions and priorities. We argue that this algorithm is efficient, where efficiency is measured by the mean/maximal number of communications needed between the enabledness and the execution of an action. We have implemented this algorithm and compared it to an implementation of an existing algorithm (α-core). Finally, we motivate the usage of this kind of specification for webservices and compare it to other works.

Keywords: Distributed systems, Memoryless Controllers, Priorities

Reviewers:

Notes:

How to cite this report:

@techreport { ,
title = { Building Distributed Controllers for Systems with Priorities },
authors = { Imene Ben-Hafaiedh, Susanne Graf and Sophie Quinton},
institution = { Verimag Research Report },
number = {TR-2010-15 },
year = { },
note = { }
}

1 Introduction

Our aim is to specify web services at a high level of abstraction as a composition of a set of service and client *processes* in order to be able to study global properties of such services, and then to generate a distributed implementation.

Usual specification languages used in that domain are based on (binary) interactions such as (synchronous or asynchronous) messages and method calls — where there may exist predefined exception mechanisms triggered on timeout or negative acknowledge.

We argue here that such services as well as the clients may alternatively be specified in terms of interactions through high-level multi-party synchronizations and global priorities. Often, using these concepts leads to much more concise specifications which are more adequate to provide an understanding of the global behavior, and which are also more adequate for the verification of global properties (see Section 3). We focus here in particular on deadlock-freedom, which in this application domain can be naturally used to express compliance [15], that is, whether a given set of services and clients can interact with each other so that in the end all clients are satisfied. On the down-side, the automatic construction of a fully distributed implementation from such a high-level specification is more complex than from usual webservice specifications, expressed in languages such as WSDL.

As an example, suppose some complex web service in which a subset of the involved service components P_i have to agree on a value v which should be chosen depending on the set of local variables x_i, such as $max\{x_i\}$ or $sum\{x_i\}$. In a specification formalism that explicitly distinguishes inputs and outputs and allows data flow only from the unique *sender* to possibly a set of *receivers*, one needs to explicitly specify some protocol – that is a set of possible sequences of interactions – to determine this value from n local values and to make it available to all components. The resulting specification is quite close to a distributed implemention and requires at least a sequence of n interactions to achieve the goal — in the case that broadcast interactions are allowed. In the case that some processes may be not available, and finally an alternative interaction must be selected, the protocol to be specified is much more complex.

We propose here to specify this situation by means of a *unique* synchronization of the required P_i which realizes the acquisition for the global value by each P_i. If there are possible alternative scenarios to be considered (due to non availability of some components, ...) we simply specify a set of alternative synchronizations — which may involve subsets of the P_i. Indeed, such a specification does not naturally suggest a particular distributed implementation.

Specifying priorities amongst a set of alternative synchronizations is interesting in that context. For example, it is likely that amongst a set of enabled synchronizations amongst subsets of P_i, one will prefer those involving larger subsets. Another typical example of the use of priorities are processes which for different activities require one or more resources amongst a shared pool of resources. Figure 1 shows the example of a client requesting through a travel agent a combined flight and hotel reservation which is represented by a rendez-vous between the agency and the set of requested reservations. A client may envisage several alternative reservation sets — and a preference amongst them in the case that more than one alternative is actually available. Also, there may be several clients competing for the same reservations through multiple agencies and there may exist preferences amongst agencies or clients

Again, specifying preferences in the usual formalisms for webservices means integrating the preferences in the overall (multi-step) protocol. In quite some cases, taking into account priorities will even simplify (determinise) the protocol, but the ratio behind this simplification (the priorities motivating certain choices) will not be visible any more.

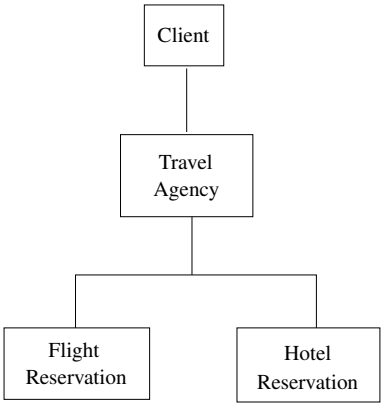

Figure 1: A system with multi-party interaction

There exist several abstract frameworks allowing to represent the specifications we have in mind, such as process algebras with priorities or prioritized Petrinets. We present here our results for BIP [10, 5] which is explicitly component-based, and even more expressive[1]. In BIP, clients and services are specified by components, synchronizations by connectors defining an interaction model and priorities are explicitly expressed by a set of preference constraints between synchronizations which must define a partial order. Also, there exist tools for checking deadlock freedom for BIP specifications such as D-Finder [6] and some algorithms for generating code for centralized and semi-distributed executions.

In this paper, we propose a method for deriving a distributed implementation from high-level specifications expressed in an abstract version of BIP[2] where components interact by exchanging messages only.

In a first step, in Section 3, instead of systematically asking the user to rework the specification when a reachable deadlock is detected, we propose to restrict the possible executions to those avoiding deadlock by means of a priority order. Why do we choose priorities as a means for avoiding deadlocks? One reason is that we suppose that specifications explicitly specify a (close to) maximal degree of concurrency and may therefore have a high degree of non-determinism. Adding buffers and reordering messages as proposed in [21] has the inconvenient of being tied to a lower level of abstraction — at which state explosion is a big issue for verification — and moreover, it is not adequate when interaction is by synchronization and each process already exhibits its maximal potential of concurrency. But restricting non determinism may be very useful, and priorities are an interesting means for doing so.

Another reason is that, when it is guaranteed that prioritized executions are deadlock free, a set of priority rules can be seen as a memoryless controller, that is, deciding which next transition is possible requires neither history nor look-ahead. In fact, the construction of the priority rules does eliminate statically the look-ahead required to avoid deadlocks without additional memory, thus keeping the specification small at that level of abstraction. Note also that priorities are convenient in the sense that by imposing priorities, it is guaranteed that no new deadlocks are introduced.

In a second step, in Section 4, we transform such a global specification into a distributed one where the identified processes communicate with each other by message passing — and we suppose that the underlying protocol layer ensures reliable and order preserving communication. Several protocols have been proposed in the past to achieve this goal for Petrinets and other formalisms where interaction is rendez-vous like. But when there are in addition global priorities to be respected, not many solutions exist. In [4, 22] solutions for distributing prioritized Petrinets have been proposed, which are both based on the use of knowledge to guarantee that in every global state at least one process knows that it can initiate some

[1] Especially when we consider data exchange; in addition BIP has an interesting concept of hierarchical connectors which we however do not need here

[2] that is without variables and value exchanges, which, as long as data size is small will not add significant complexity to the algorithm as it can be easily piggybacked on protocol messages.

interaction obeying the global priority rules, that is, the methods of these papers propose to construct a distributed disjunctive controller (see also Section 5).

Note that both solutions rely on an algorithm such α-core [16] for achieving a distributed execution of the constructed extended Petrinet and both methods may ignore many legal interactions just because in no state there is a process with enough knowledge to initiate them. And as long as in every reachable global state there is at least one transition that can be triggered, this is considered as sufficient.

We propose here an alternative method for distributing systems with priorities which consists in merging the priority and the α-core layer, and we have several reasons for it.

- In the domain of webservices, eliminating alternatives because of lack of knowledge is eliminating them for bad reasons. Indeed, we still want to be able to consider all alternatives that a client may envisage. Thus, like in [20, 19], we are interested in a distributed systems that allows executing any legal transition, not only those for which there happens to preexist sufficient knowledge for taking a local decision.

- In [22] there is an important redundancy between the messages needed for the "upper layer" to check the enabledness of the transitions in which the process may participate in the next step, and the messages of the lower layer, that is, α-core algorithm.

The algorithms and the method presented are not directly tied to their application to webservices. Also the BIP language — even its full version with data — is not meant to be used by web service designers. We aim here rather at discussing some concepts for specifying webservices which we consider interesting, and we provide some relevant algorithms which would allow deploying real languages which include these concepts. Related approaches and algorithms in the context of webservices are discussed in Section 5.

The paper is organized as follows: Section 2 introduces the concepts of BIP on which we rely in this paper. Section 3 discusses the use and the construct of priorities as memoryless controllers for avoiding deadlocks. We discuss there also how we handle "confusion", a well known obstacle for the distribution of a global system. Section 4 presents the main algorithm that transforms a system (with binary synchronizations) and its memoryless controller into a distributed protocol. We compare our implementation of this algorithm to an implementation of α-core and we also propose an extension for arbitrary multi-party interactions. Section 5 discusses related work and Section 6 concludes and hints at worthwile future developments.

2 Preliminaries

In [10, 5, 7], BIP, a language and framework for component-based design and verification has been introduced. In BIP, system specification separately defines a set of component behaviors, a composition structure defined by an *interaction model* defined by a set of connectors, and a priority given in the form of a preorder amongst interactions. BIP is close to prioritized Petrinets or to process algebras such as CCS [13] or CSP [11], but as opposed to them, BIP is explicitly component-based with a strict notion of locality which allows encapsulation. Also, BIP introduces a notion of hierarchical connector and powerful concepts for the definition of data transfer.

We introduce here a simplified abstract version of BIP without hierarchical connectors and without data transfer. Also, we define composition simply by means of a set of interactions, and omit the definition of connectors which allow grouping interactions, and for which special rules apply.

Definition 2.1 (Atomic component)**.** *An atomic component is a* Labeled Transition System *(LTS) represented by a tuple $(Q, q^0, \mathcal{P}, \delta)$ where Q is a set of states, $q^0 \in Q$ is an initial state, $\mathcal{P}$ is a set of labels and $\delta \subseteq Q \times \mathcal{P} \times Q$ is a transition relation.*

As usually, $q_1 \xrightarrow{a} q_2$ denotes $(q_1, a, q_2) \in \delta$ and $q_1 \xrightarrow{a}$ denotes $\exists q' \in Q, q \xrightarrow{a} q'$.

Interaction Given a set of n atomic components $K_i = (Q_i, q_i^0, \mathcal{P}_i, \delta_i)$ for $i \in [1, n]$, in order to build a component representing their composition, we require their sets of ports to be pairwise disjoint, i.e. for any two $i \neq j$ from $[1, n]$, $\mathcal{P}_i \cap \mathcal{P}_j = \emptyset$. The set of ports of the composed system is then defined as $\mathcal{P} = \bigcup_{i=1}^{n} \mathcal{P}_i$. The composition is defined by a set of *interactions* where an interaction $a \subseteq \mathcal{P}$ is a set of ports of the composition. It represents how components are allowed to interact with each other. Note that interactions only specify through which ports components may interact with each other. Without loss of generality, and for simplifying the notations, we denote interactions in the following by $a = \{p_i\}_{i \in I}$ where $I \subseteq [1, n]$ and $\forall i \in I,\ p_i \in \mathcal{P}_i$.

Definition 2.2 (Composition of BIP components)**.** *The composition of n components K_i as above with $\gamma \subseteq 2^{\mathcal{P}}$ is denoted by $K = \gamma(K_1, \ldots, K_n)$ and it is defined as an LTS $(Q, q^0, \mathcal{P}, \delta)$ such that $Q = \prod_{i=1}^{n} Q_i$, $q^0 = (q_1^0, \ldots, q_n^0)$, $\mathcal{P} = \bigcup_{i=1}^{n} \mathcal{P}_i$ and δ is the least set of transitions satisfying the rule:*

$$\frac{a = \{p_i\}_{i \in I},\ \forall i \in I.\ q_i^1 \xrightarrow{p_i} q_i^2 \wedge\ \forall i \notin I.\ q_i^1 = q_i^2}{(q_1^1, \ldots, q_n^1) \xrightarrow{a} (q_1^2, \ldots, q_n^2)}$$

This rule states that $\gamma(K_1, \ldots, K_n)$ may execute an interaction $a \in \gamma$ iff for each port $p_i \in a$, the corresponding atomic component K_i can execute the transition labeled with p_i — the states of components that do not participate in the interaction remain unchanged. In the following, what we call *system* is an LTS representing the composition of a set of atomic components.

Priorities A component system may be non-deterministic, that is, several interactions may be enabled in each (global) state which may be present in individual components or introduced by composition. We allow restricting non-determinism by means of priorities, specifying which of the interactions should be preferred over which others if they are enabled.

Definition 2.3 (Priority order)**.** *A* priority order *denoted by $<$ is a strict partial order on the set of interactions I. We denote that an interaction a has lower priority than b by $a < b$.*

Definition 2.4 (System controlled by a priority order)**.** *A system $S = (Q, q^0, \mathcal{P}, \delta)$* controlled *by a priority order $<$ defines an LTS $(Q, q^0, \mathcal{P}, \delta_<)$ where $\delta_<$ is defined by the following rule:*

$$\frac{q^1 \xrightarrow{a} q^2\ \wedge\ \nexists b \in \gamma.\ (a < b \wedge\ q^1 \xrightarrow{b})}{q^1 \xrightarrow{a}_< q^2}$$

Thus, only interactions that are locally enabled in all concerned components, and furthermore not inhibited by an interaction with higher priority, may be fired. An interesting property of priorities is the well-known fact that they allow restricting the behavior of a system by guaranteeing that no new deadlocks are introduced by this restriction. This is the reason why we want to use priorities to "control" systems. We denote the resulting controlled system by $(S, <)$.

In the remainder of this paper, we use the term *process* rather than component. In addition, we explicitly name interactions rather than ports. That is, as usually, I is a set of names or labels and the alphabet $\mathcal{P}_i$ represents the set of interactions in I in which processes K_i may and must participate. This leads sometimes to a bit more cumbersome specifications, as it requires "duplicating" ports that participate in more than one interaction, but it simplifies substantially the presentation of the results.

The composition rule above can therefore be simplified to the following one:

$$\frac{\forall i \text{ s.t. } a \in \mathcal{P}_i.\ q_i^1 \xrightarrow{a} q_i^2 \wedge\ \forall i \text{ s.t. } a \notin \mathcal{P}_i.\ q_i^1 = q_i^2}{(q_1^1, \ldots, q_n^1) \xrightarrow{a} (q_1^2, \ldots, q_n^2)}$$

This rule says that the composition may execute interaction a iff all the processes having a in their alphabet $\mathcal{P}_i$ can execute a transition labeled by a. This defines a *strong synchronization*.

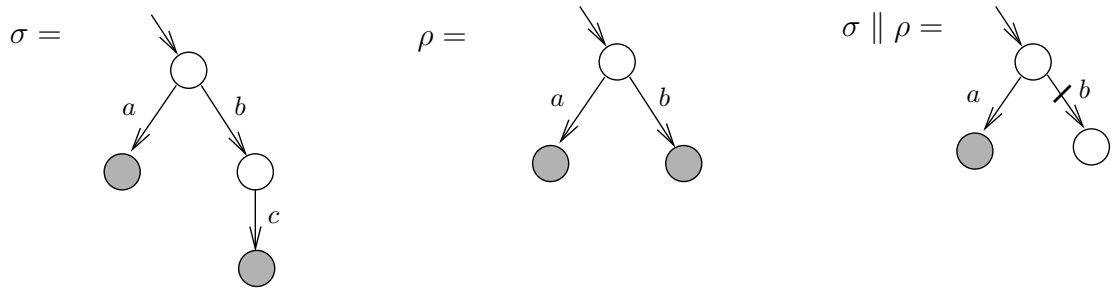

Figure 2: An example where reducing non-determinism eliminates a deadlock.

3 Controlling systems with priorities

C is controller for a system S and a property φ if each global state of S, C may forbid a subset of the interactions of S and by doing so guarantees φ. We propose in this section to use priorities to control systems in order to avoid deadlocks of the original system.

We distinguish between *controllable* and *non-controllable* non-determinism. The fact that in a state several transitions with different labels may be enabled is considered as *controllable* non-determinism, because a controller may choose which one may or may not be taken. However, when in some state there are two or more transitions enabled with the same label, this corresponds to *non-controllable* non-determinism, as an interaction-based controller cannot choose the state after the interaction takes place. Internal actions τ are another kind of non-controllable non-determinism. For the sake of readability and without loss of generality, we do not represent them in this paper[3].

We propose to use *priorities* (see Definition 2.3) to arbitrate between simultaneously enabled interactions so as to avoid undesired states. These priorities are *static* in the sense that they do not depend on the state of the system. On the other hand, *dynamic* priorities depending on the state of the system could be useful. In fact, dynamic priorities always allow controlling S for deadlock freedom. They allow to state that $a < b$ in state q_1 and $a > b$ in another state q_2. Nevertheless, handling dynamic priorities may require much more precise knowledge about the global state of the system, which can only be approximated in a distributed setting [9], and which could require a too strong degree of control of concurrency to be interesting in practice. Now, the use of some dynamic priorities depending on (almost) local conditions could be an interesting tradeoff which we do not further explore here.

We consider only static priorities, as a trade-off between the need for some global constraints and the risk of building a specification for which no useful distributed implementation is derived.

A process may possess some particular states called *final* states (represented in grey in the figures of this paper). We consider as problematics only states which are not final and in which the system cannot continue. Our goal is to avoid these *deadlock* states by reducing non-determinism.

Definition 3.1 (Final state, deadlock state)**.** *Let be a system $S = (Q, q^0, \mathcal{P}, \delta)$ built as a composition of n processes $P_i = (Q_i, q_i^0, \mathcal{P}_i, \delta_i)$ for $i \in [1, n]$. Let $q = (q_1, \dots, q_n) \in Q$ be a state of S.*

- *q is a* final *state of S iff $\forall\ q_i,\ i \in \{1...n\}\ s.t.\ q = (q_1, \dots, q_i, \dots, q_n),\ q_i$ a* final *state of P_i*
- *q is a* deadlock *state of S iff q is not final and $\nexists\ q' \in Q,\ \nexists a \in \mathcal{P}\ s.t.\ q \xrightarrow{a} q'$*

The system is deadlock free *iff it has no deadlock state.*

Based on this definition of deadlock freedom, the composition of processes ρ and σ which are both defined on the set of interactions $\{a, b, c\}$ and depicted in Figure 2 is not deadlock free. Initially, both processes can interact either on a or on b, but if they interact on b, then σ expects an interaction on c before

[3] τ-transitions can be transformed into a set of transitions with the same label, and vice versa.

terminating in a *final* state, while ρ has already terminated. However, by reducing the non-determinism of $\sigma \parallel \rho$ and by choosing a rather than b it is possible to eliminate this deadlock. We can express this global choice by defining b to have lower priority than a.

The problem we want to solve is therefore the following one: given a system S, determine whether there exists a priority order $<$ such that $(S, <)$ is deadlock-free. This priority order defines a *memoryless controller* for S.

Lemma 1 (priorities as memoryless controllers)**.** *A priority order $<$ is a* memoryless controller *of a system S for deadlock freedom if $(S, <)$ is deadlock free.*

Indeed, if $(S, <)$ is deadlock free, $<$ defines a controller for S as $<$ defines deadlock free restriction of S. In addition, the controller defined by $<$ is indeed memoryless, as in each state the set of interactions that are enabled in $(S, <)$ depend only on (an approximation of) the current global state of the system, and neither of the past nor the possible futures.

We now introduce notations allowing to distinguish between the enabledness of a transition locally in some process, in the uncontrolled system S and in the controlled system $(S, <)$

Definition 3.2 (locally ready, globally ready, enabled interaction)**.** *Let be a system $S = (Q, q^0, \mathcal{P}, \delta)$ built as a composition of n processes $P_i = (Q_i, q_i^0, \mathcal{P}_i, \delta_i)$ with $i \in [1, n]$. Let $<$ be a priority order on $\mathcal{P}$. Consider a global state $q \in Q$ such that $q = (q_1, ..., q_n)$ and an interaction $a \in \mathcal{P}$.*

- *For i such that $a \in \mathcal{P}_i$, a is* locally ready *in q_i iff $\exists\ q_i' \in Q_i,\ s.t.\ q_i \xrightarrow{a} q_i'$*
- *a is* globally ready *in q iff $\exists q' \in Q.\ q \xrightarrow{a}_S q'$*
- *a is* enabled *in q iff a is* globally ready *in q and no interaction with higher priority is also globally ready in q, that is, iff $q \xrightarrow{a}_{(S,<)}$.*

Note that only enabledness is related to priorities, and enabledness of a implies global readiness which in turn implies local readiness in all processes which have a in their alphabet.

An example based on dining philosophers We consider a variant of the dining philosophers problem inspired from [15]. Philosophers are seen as processes who provide thoughts if they are given two forks. These forks represent a shared resource which is represented by a process providing forks and expecting to get thoughts in return. We consider here two philosophers and a resource with two forks. A deadlock arises if both philosophers have a fork and wait forever for a second one.

This deadlock can be avoided easily by always giving the highest priority to the request that is the closest to completion. This is a classic method for managing resources. The priority order that is needed here is $\{fork_1^\alpha < fork_2^\beta,\ fork_1^\beta < fork_2^\alpha\}$. For readability reasons, in Figure 3, the interaction $fork_{1,2}^{\alpha,\beta}$ in the behavior of the process $Forks$ corresponds to the interactions $\{fork_1^\alpha,\ fork_2^\alpha,\ fork_1^\beta,\ fork_2^\beta\}$ of the two philosophers.

Let us note here that in this simple example priorities are *local*, in the sense that for each priority $a < b$, there exists a process involved in both a and b — here, the pool of forks. However, in a more complex setting where resources are distributed, the same methodology is still valid and requires *global* priorities.

3.1 Synthesis of priorities

We propose in this section an algorithm that computes a priority order on the set of interactions of a system S that defines a memoryless controller for S. The algorithm first detects deadlocks. Then, if possible, a set of priority rules is computed which make these deadlock states unreachable by *inhibiting* some transitions (a transition is inhibited if it is ready but not enabled) on each path to deadlock. The algorithm provides a result that is a priority order representing a controller for deadlock freedom if such priority exists. If priorities fail to be usable, the specification may be revised or a more classical (memoryless or memoryful) controller may be constructed using classical methods for controller synthesis (such as [18]) which may be much more expensive, and as for dynamic priorities may lead to specifications that cannot reasonably be distributed. Before presenting the full algorithm, we illustrate how it works — or fails — on very simple examples.

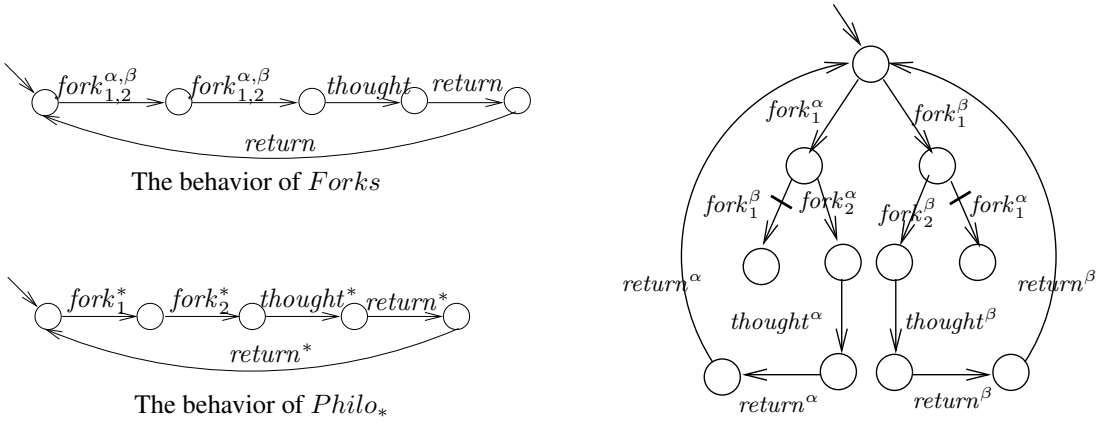

Figure 3: A solution to the dining philosophers problem.

A simple system controllable by priorities Consider the processes σ and ρ depicted in Figure 4 which are both defined on $\{a, b, c, d\}$. A deadlock is reached in system defined by σ and ρ if initially interaction a is chosen. If interaction b is chosen instead, the deadlock is avoided and a *final* state is reached. Thus, the priority order defined by $a < b$ is sufficient to make the deadlock unreachable.

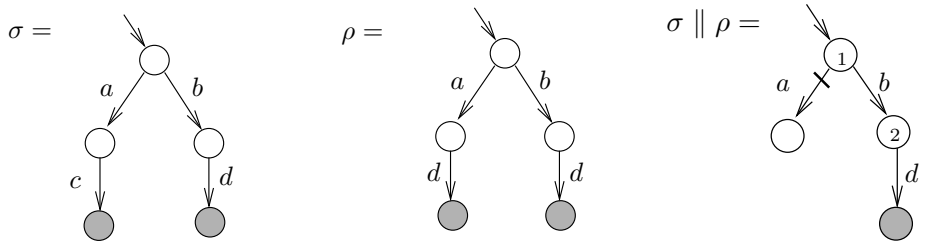

Figure 4: A simple system controllable by priorities.

A simple system not controllable with priorities Figure 5 shows again a system S defined by two processes on the same alphabet. For S, there exists no priority that allows avoiding both deadlocks. We sketch below how this is detected by our algorithm.

1. The transition relation of S is (partly) computed and transitions leading to deadlocks are marked as *error* transitions, as shown in figure 5.

2. In the initial state 1 of S, b must be preferred to a in order to prevent a deadlock, and we conclude that any priority order making S deadlock free includes the rule $a < b$.

3. In state 2, there are two possibilities to avoid the deadlock occurring if b is chosen: either b has lower priority than a or state 2 is not reachable. The first option is impossible as it would mean that $a < b$ and $b < a$ which violates the requirement that a priority is a strict partial order. The second option implies that the transition from 1 to 2 labeled by b should be inhibited by a transition with higher priority enabled in 1 which leads to exactly the same contradiction.

We conclude that no priority can control the given example to guarantee deadlock freedom. Note that *dynamic* priorities can deal with this example, as it is sufficient to define as priorities, $a < b$ in state 1 and then $b < a$ in state 2.

The general algorithm is given below, and it is quite easy to see how it could be extended for generating dynamic priorities — if needed.

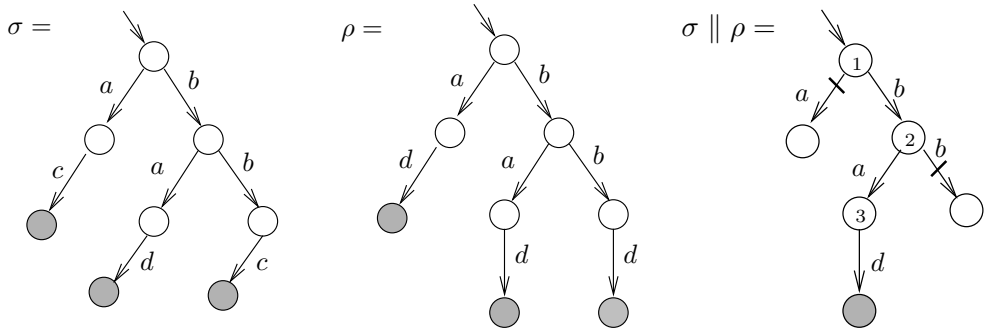

Figure 5: A simple system not controllable with priorities.

Algorithm for inferring priorities The main algorithm *Priority4Tr* is called initially with B, the transition system representing S the composition of the set of processes. Priority orders are represented by a set of rules $a < b$ where for simplicity, we suppose also those rules which can be deduced by transitivity are explicitly represented. Initially, Tr holds the set of *error transitions* leading to a deadlock in a single step, and at any time, Tr holds the set of error transitions which are not yet forbidden by some priority; and $Prio$ contains initially the empty set of priorities, and at the successful termination it contains a desired solution. The main algorithm successively calls *Initialize*, *PotentialOrders* and *FindOrRefine*, which play the following roles: 1) *Initialize* computes the priority rules $Prio$ that are *necessary* to avoid a deadlock from some state considered as reachable. If $Prio$ contains a contradiction the overall algorithm terminates with failure, 2) *PotentialOrders* computes a set of alternative priority rules Pot which may be used to control the execution, and 3) *FindOrRefine* picks one such priority order and explores it. If it fails, then another alternative in Pot is explored until success or failure — if none of them works. In these algorithms, we also use the following notations.

- For any transition $t \in Tr$, Pot_t represents the set of potential priority rules that can inhibit t. $Pot \backslash \{p\}$ denotes that rule p is removed from *pot* and thus from all sets Pot_t.
- $q^0 \longrightarrow^*! s$ denotes that s is reachable from q^0 by a (possibly empty) sequence of transitions for which there exists no alternative, that is, transitions of the form (s_1, α, s_2) such that δ has no other transition with s_1 as origin.
- It is understood that functions manipulating priority orders, like add and union, always include a normalization that guarantees transitivity.

The correctness of the algorithm is guaranteed by (1) the fact that if $Prio$ does not define strict partial order, the algorithm terminates unsuccessfully. (2) at any point of time, $Prio$ together with avoiding Tr guarantees avoidance of deadlocks because initially avoiding Tr obviously guarantees deadlock freedom, and a transition t is only eliminated from Tr if there is a rule in $Prio$ forbidding it or if t is replaced by some transition (set) leading to the start state of t. On termination, Tr is empty, thus $Prio$ — which is a priority — is able to prevent all deadlocks. (3) The fact that the algorithm terminates unsuccessfully implies that indeed there is no appropriate priority is guaranteed by the fact that the algorithm systematically explores transition sets allowing to block the access to a deadlock state without introducing a new deadlock, and such a set is rejected only if avoiding requires contradictory priorities. □

For readability, we present *Initialize* in a call-by-reference fashion, while *Priority4Tr*, *PotentialOrders* and *FindOrRefine* are call-by-value.

3.2 Dealing with confusion

Confusion is a situation occurring in distributed systems. Typically, detecting those situations is important for designing correct algorithms for partial order reduction. In presence of priorities, confusion situations may compromise correctness of a distributed implementation of a specification. We first define some preliminary notions which allow us to characterize different situations of confusion.

Algorithm 1 Priority4Tr($B = (Q, q^0, \Sigma, \delta)$, Tr, $Prio$): priorityOrder or $\perp$

if $Tr = \emptyset$ **then**
  return $Prio$ *// there are no* error *transitions, so $Prio$ is a solution*
else
  $Pot \longleftarrow \emptyset$
  Initialize(B, Tr, $Prio$, Pot) *// initialize sets $Prio$, Pot; simplify B, Tr accordingly*
  if $Tr = \emptyset$ **then**
    return $Prio$
  else if $Pot = \emptyset$ **then**
    return $\perp$ *// error transitions cannot be avoided, so there is no solution*
  end if
  $\mathcal{P} \longleftarrow$ PotentialOrders(Tr, $Prio$, Pot) *// calculate the set of potential priority orders*
  return FindOrRefine(B, Tr, Pot, $\mathcal{P}$) *// find P of $\mathcal{P}$ being a solution or refine it*
end if

Algorithm 2 Initialize(B, Tr, $Prio$, Pot)

for all $t = (s, a, s') \in Tr$ **do**
  if $q^0 \longrightarrow^*! \ s$ and $s \xrightarrow{b} \in \delta$ implies $(b < a \in Prio \vee b = a)$ **then**
    $Pot \longleftarrow \emptyset$ **break**
  else if $q^0 \longrightarrow^*! \ s$ and $\exists b \in \Sigma$ s.t. $\{b\} = \{l |\ s \xrightarrow{l} \wedge\ l \neq a \wedge (l < a \notin Prio)\}$ **then**
    if $b < a \in Prio$ **then**
      $Pot \longleftarrow \emptyset$ **break**
    end if
    add($Prio, a < b$) *// add $a < b$ to Prio and normalize by adding induced priorities*
    $Tr' \longleftarrow \{t\} \cup \{(q, a, q') \in \delta \mid q \xrightarrow{b}\}$ *// remove from δ transitions inhibited by $a < b$*
    simplify(Q, δ) *// simplify δ and Q by removing unreachable transitions and states*
    $Tr \longleftarrow Tr \cap \delta$ *// simplify Tr in accordance with the new δ*
  else
    $Pot_t \longleftarrow \{a < b \mid s \xrightarrow{b} \wedge\ b \neq a \wedge (b < a \notin Prio)\}$
    if $Pot_t = \emptyset$ **then**
      $\delta \longleftarrow \delta \backslash \{t\}$ *// since t cannot be inhibited, its origin state must be made unreachable*
      simplify(Q, δ) *// simplify δ and Q*
      $Tr \longleftarrow (Tr \cup \{(q, l, s) \in \delta\}) \cap \delta$ *// simplify Tr according to δ*
    end if
  end if
end for

Algorithm 3 PotentialOrders(Tr, $Prio$, Pot): Set of priorityOrders

choose $t \in Tr$
$P \longleftarrow \{p \in Pot_t \mid$ add($Prio, p$) is defined$\}$ *// some priorities in Pot_t may contradict $Prio$*
for all $p \in P$ **do**
  $Tr^p_{ok} \longleftarrow \{t' \mid p \in Pot_{t'}\}$ *// transitions in Tr inhibited by p*
end for
$Unreachable_t \longleftarrow$ PotentialOrders($Tr \backslash \{t\}$, $Prio$, Pot) *// suppose t needs not be inhibited*
$Inhibited_t \longleftarrow \bigcup_{p \in P}$ PotentialOrders($Tr \backslash (\{t\} \cup Tr^p_{ok})$, add($Prio, p$), $Pot \backslash \{p\}$) *// t is inhibited by p*
return $Unreachable_t \cup Inhibited_t$

Algorithm 4 FindOrRefine(B, Tr, Pot, $\mathcal{P}$)

if $\exists P \in \mathcal{P}$ s.t. $\forall t,\ Pot_t \cap P \neq \emptyset$ **then**
  return P *// P inhibits all* error *transitions, thus P is a solution*
else
  while $\mathcal{P} \neq \emptyset$ **do**
    choose $P \in \mathcal{P}$
    $\mathcal{P} \longleftarrow \mathcal{P} \backslash \{P\}$
    $Tr_{bad} \longleftarrow \{t \mid Pot_t \cap P = \emptyset\}$ *// transitions not inhibited by P*
    $\delta \longleftarrow \delta \backslash (Tr \backslash Tr_{bad})$ *// remove inhibited transitions from δ*
    simplify(Q, δ) *// simplify δ and Q*
    $Tr \longleftarrow (Tr \cup pre(Tr_{bad}))$ *// add predecessors of transitions not inhibited to Tr*
    $Tr \longleftarrow Tr \cap \delta$ *// simplify Tr according to δ*
    $Result \longleftarrow$ Priority4Tr$(B,\ Tr,\ P)$
    if $Result \neq \perp$ **then**
      return $Result$
    end if
  end while
  return $\perp$
end if

Throughout this section, consider a system $S = (Q, q^0, \mathcal{P}, \delta)$ built as a composition of n processes $\{P_i = (Q_i, q_i^0, \mathcal{P}_i, \delta_i)\}_{i=1}^n$, $S = P_1 \parallel ... \parallel P_n$ and a priority order $<$ defining a prioritized system $(S, <)$

Definition 3.3. *Let be an interaction $a \in \mathcal{P}$ and $q = (q_1, ..., q_n) \in Q$ a global state in which a is globally ready. We denote ind_a^q the set of indexes of the processes which must participate in a, that is, $ind_a^q = \{i_1, ...i_k\}$ such that $\{P_j \mid j \in [1..k]\}$ is exactly the set of processes involved in a (and therefore in which a is locally ready in q).*

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

Definition 3.4 (Concurrent and conflicting interactions)**.** *Let a, b be interactions of $\mathcal{P}$ and $q \in Q$ a global state in which a and b are globally ready.*

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

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

Figure 6 illustrates a situation of structural conflict: the interactions a_1 and a_3 are in *structural conflict* as they both involve process ρ_1. Figure 7 illustrates a *prioritized conflict* of a_1 with a_3 as these interactions are concurrent but $a_1 < a_3$ holds.

Confusion is a situation where concurrency and conflict are mixed. More precisely, confusion arises in a state where two interactions a_1 and a_2 may fire concurrently, but firing one modifies the set of interactions in conflict with the other. A *symmetric* and an *asymmetric* situation of confusion are shown in Figure 6:

In the symmetric case, the interactions a_1 and a_2 of σ_1 and ρ_1 are concurrent but are both in conflict with a_3 and the execution of a_1 (resp. a_2) changes the set of interactions in conflict with the other one. In the asymmetric case, the interactions a_1 and a_2 of σ_2 and ρ_2 are concurrent but a_1 will enter in conflict with a_3 if a_2 fires before a_1.

Definition 3.5 (Confusion)**.** *Let a_1 and a_2 be interactions, and q a global state of S. We suppose that a_1 and a_2 are concurrent — and thus globally ready — in q.*

- *a_1 is in a situation of* structural confusion *with a_2 if $\exists q' \in Q$ s.t. $q \xrightarrow{a_2} q'$ implies $Conflict_q(a_1) \neq Conflict_{q'}(a_1)$*
- *a_1 is in a situation of* prioritized confusion *with a_2 if $\exists q' \in Q$ s.t. $q \xrightarrow{a_2} q'$ implies $PrioConflict_q(a_1) \neq PrioConflict_{q'}(a_1)$*

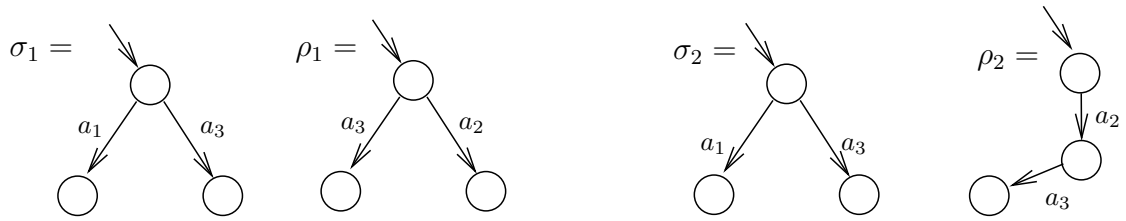

Figure 6: Symmetric and asymmetric confusion.

Figure 7 illustrates a situation of *prioritized confusion*: a_1 and a_2 are concurrent, however firing a_2 enables a_3 which has higher priority than a_1 which means that a_1 is no more enabled after the execution of a_2.

The classical notion of confusion is what we call structural confusion. Note that all situations of confusion are important for designing partial order reductions which are very important for making the verification of global properties of S feasible. The reason is that eliminating arbitrarily one of the two interleavings of a_1 and a_2 may change the set of reachable states, and thus lead to different verification results.

For designing a distributed implementation of $(S, <)$, only the situation of Figure 7 — where executing a_2 disables a_1 due to a new priority conflict — is problematic. The reason is that in this case a_1 and a_2 are not really "concurrent", whereas in all other cases, it does still hold that a_1 and a_2 can be executed in any order and both orders lead to the same global state.

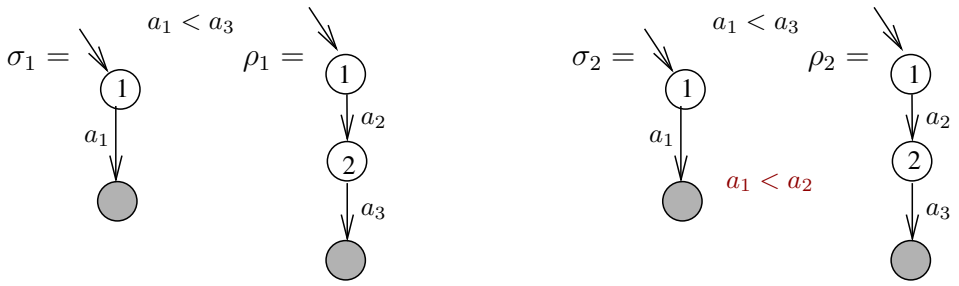

Figure 7: Prioritized confusion.

In Section 4, we propose a distributed implementations of systems $(S, <)$ in which concurrent interactions are executed independently, based on the notion of concurrency of definition 3.4. This means that our algorithm does not support systems $(S, <)$ with such prioritized conflict situations.

In order to deal with this kind of confusion, we could use a more appropriate notion of concurrency which however would lead to inefficient implementations. We rather propose to eliminate such confusions statically by adding a priority $a_1 < a_3$ or $a_2 < a_1$ such that either a_1 and a_2 are not anymore considered concurrent or at least it is guaranteed that executing concurrent interactions does not introduce priority conflicts which destroy this concurrency. Note that adding *priorities* between *concurrent* interactions in a given system does not add new deadlocks.

4 Distributing prioritized systems

Given a system $(S, <)$ defined by a set of processes P_i, a set of interactions and a priority order $<$ to be enforced, our goal is to define a distributed implementation for $(S, <)$. In this section, we define an algorithm which constructs such a distributed implementation by defining for each process a (local) controller such that the joint execution of all processes P_i and their corresponding controllers guarantees the following:

1. all executions are executions of $(S, <)$, that is executions of S respecting $<$
2. if $(S, <)$ is deadlock free, then no deadlock will ever occur

Controllers are described as protocols, and we want them to decide the next interaction to be taken "as quickly as possible", where we measure the complexity by the number of messages required to execute a given interaction a[4].

We rely on $(S, <)$ to guarantee deadlock-freedom — and fairness. That is, any distributed implementation of $(S, <)$ that does not introduce deadlocks is considered correct. Indeed, we suppose that — if needed — $(S, <)$ has been obtained using the algorithm of section 3.1 that eliminates deadlocks. For this reason, we suppose in the following that $(S, <)$ has no deadlock.

4.1 Description of the protocol

The system is supposed to have a fixed number of processes, although it may be arbitrarily large. In order to simplify the presentation of the algorithm, we suppose here only binary interactions; an extension to arbitrary multi-party synchronizations is discussed in Section 4.5. We also assume that the internal activities of processes are terminating and that there exists no prioritized confusion, that is, the notion of concurrency used by the algorithm is correct[5]. As quite usually, we assume that the message passing mechanism ensures the following basic properties:

1. any message is received at the destination within a finite delay;
2. messages sent from location L_1 to L_2 are received in the order in which they have been sent;
3. there is no duplication nor spontaneous creation of messages.

For each interaction α involved in at least one priority rule, one of the involved processes P_i place the role of the negotiator for α. If there exists at least one interaction with higher priority, the role of the negotiator is to check for the enablednes of α, and if there exists at least one interaction with lower priority, its role is to answer readiness requests. The choice of negotiators is discussed in Section 4.4.

We now describe the controllers of individual processes which enforce correct executions, and in particular adherence to the global priority order. It is understood that what is called process is in fact a *controlled* process.

The Controller associated with each process, maintains a set of data structures shared and maintained by the different subtasks of the controller: $readySet$ (resp. $enabledSet$) contains the set of interactions which are known to be globally ready (resp. enabled) in the current local state q, and involved and $possibleSet$ maintains the set of interactions that are locally ready. Note that $possibleSet$ contains purely local information which can be calculated immediately when entering a new local state. The other two sets are calculated by a series of message exchanges, and the complete information is generally not calculated but as soon as an interaction is known to be enabled, its triggering will be initiated.

The general structure of the controller for each individual process P_i is shown in Figure 8. The overall controller — and the process to be controlled — are represented as a set of parallel activities (which we call threads, and which in our implementation are realized as Java threads) with a shared memory and shared message buffers.

[4] it would even be sufficient to count the number of messages between the the first state in which a is globally enabled and the first state in which a "has been executed".

[5] Without this last condition, we may observe global executions which are witnesses of a priority violation.

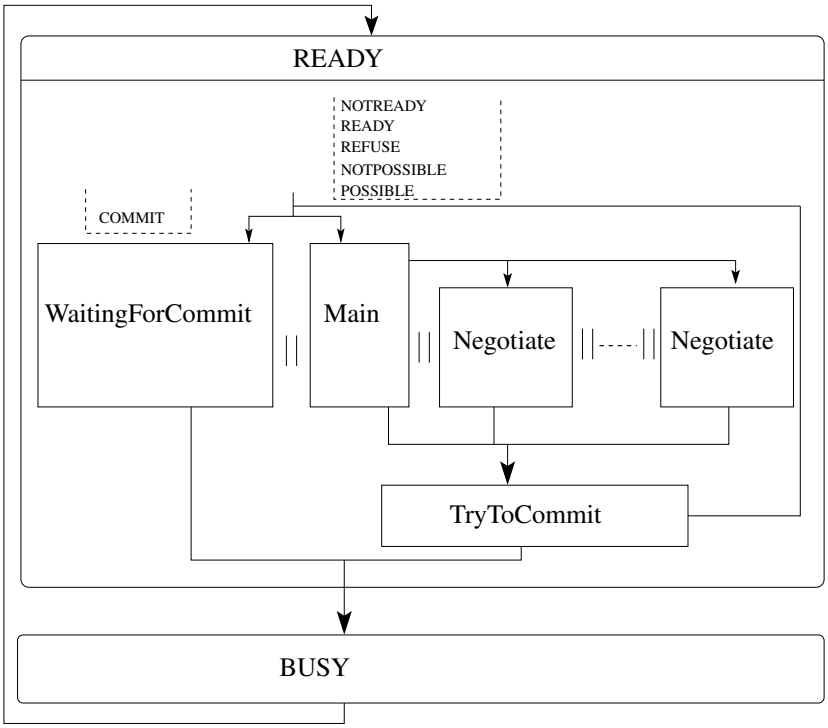

Figure 8: Structure of the protocol for one process

Indeed, incoming messages are stored until one of the activities is ready to handle them. We use several FIFO buffers which are chosen such that the order amongst messages stored in different buffers does not influence the algorithm; in particular, they are used by concurrent threads. A buffer, which is read only by the thread $Main$, stores messages of the form $POSSIBLE(a)$, $NOTPOSSIBLE(a)$, $READY(a)$, $NOTREADY(a)$, and $REFUSE(a)$. A second buffer stores messages of the form $COMMIT(a)$, this buffer is read first by thread $WaitingForCommit$, then by $TryToCommit$. The role of each message is described in Table 1. Given that we are handling binary interactions, we do not explicit the recipient or the sender.

Message	Description
$POSSIBLE$	Offer an interaction (which is locally ready)
$NOTPOSSIBLE$	respond that an interaction is not locally ready
$READY$	Ask about the global readiness of an interaction
$NOTREADY$	Respond that an interaction is not globally ready
$COMMIT$	Commit to an interaction (cannot be undone by P_i)
$REFUSE$	Inform that a process cannot commit to an interaction

Table 1: Messages used by the algorithm

P_i is either in state $Ready$ or in state $Busy$. In state $Busy$, P_i executes the local action of the interaction that has been chosen. Incoming messages are stored and will not be handled until the controller moves to state $Ready$. In state $Ready$, the controller of P_i looks for a next interaction to fire, proceeding as follows.

- The $Main$ thread starts by checking its locally ready interactions ($possibleSet$) for interactions that are globally ready (see Algorithm 5). To check the global readiness of an interaction a, messages

Algorithm 5 Main

Require: $toNegotiate = \{a \in possibleSet \mid negotiator(a) = K\}$
// The set of interactions for which K is a negotiator **Input:** set of interactions $possibleSet \neq \emptyset$
Output: interaction i
$prioFree = \{a \in possibleSet \mid \nexists b \,.\, b < a\}$
$waitingSet \longleftarrow \emptyset$
checking global readiness:
$notReadySet \longleftarrow \emptyset$
$readySet \longleftarrow \emptyset$
$lessPrio(a) = \{b \in readySet \mid b < a\}\}$
for all $a \in possibleSet$ **do**
  send $POSSIBLE(a)$
end for
create WaitingForCommit($possibleSet$)
if receive $POSSIBLE(a)$ and $a \in toNegotiate$ **then**
  create Negotiate(a) and $readySet \longleftarrow readySet \cup \{a\}$ and
  for all $b \in lessPrio(a)$ **do**
    kill Negotiate(b)
  end for
end if
WHEN $\exists\ a\ s.t.$ Negotiate(a)= OK or (**receive** $POSSIBLE(a)$ and $a \in prioFree$)
call TryToCommit(a) and **kill** WaitingForCommit($possibleSet$) and $\forall b \in readySet$ **kill** Negotiate(b)

if TryToCommit(a)= OK **then**
  return a
else
  goto checking global readiness
end if
if $\forall a \in readySet$ Negotiate(a)= NOK **then**
  goto checking global readiness
end if
if receive $REFUSE(b)$ and $b \in readySet$ **then**
  kill Negotiate(b) and $readySet \longleftarrow readySet \backslash \{b\}$
end if
if receive $POSSIBLE(b)$ and $b \in possibleSet \backslash \{toNegotiate \cup prioFree\}$ **then**
  send $POSSIBLE(b)$ and $readySet \longleftarrow readySet \cup \{b\}$
end if
if receive $NOTPOSSIBLE(b)$ and $b \in possibleSet \backslash prioFree$ **then**
  $notReadySet \longleftarrow notReadySet \cup \{b\}$
end if
if receive $POSSIBLE(b)$ and $b \notin possibleSet$ **then**
  send $NOTPOSSIBLE(b)$
end if

of the form $POSSIBLE(a)$ are exchanged, and peers in which a is currently not locally enabled respond with $NOTPOSSIBLE(a)$ after which the requesting process "abandons" a until it changes state or the peer enters a state in which a is locally enabled and sends a $POSSIBLE(a)$.

Whenever it is detected that an interaction a for which it plays the role of a negotiator is globally ready, a thread $Negotiate(a)$ is created which checks whether a is enabled (which corresponds to transition 1 of Figure 10 and Figure 8). If an interaction with maximal priority is globally ready, it is immediately known to be enabled.

- $Negotiate(a)$ checks the enabledness of an interaction a (see Algorithm 6). It asks all negotiators of interactions with higher priority than a if their interactions are globally ready by sending a $READY(b)$ message to all negotiators of interactions b with higher priority than a.

  In turn the negotiators of b, if not $BUSY$, respond positively or negatively as soon as they have the information available.

Algorithm 6 Negotiate

Require: $higherPrio(a) = \{c \mid a < c\}$
  Input: interaction a **Output:** OK or NOK
  $toCheck \longleftarrow higherPrio(a)$
  for all $b \in toCheck$ **do**
    send $READY(b)$
  end for
  while $toCheck \neq \emptyset$ **do**
    if receive $READY(b)$ **then**
      return NOK
    else if receive $NOTREADY(b)$ **then**
      $toCheck \longleftarrow toCheck \backslash \{b\}$
    end if
  end while
  return OK

- $Main$ handles local priorities locally. Whenever an interaction b is known to be globally ready, $Main$ kills all threads $Negotiate(a)$ with $a < b$.
- Concurrently to $Main$, $WaitingForCommit$ handles incoming $COMMIT$ messages (see Algorithm 7). Whenever a $COMMIT(a)$ is received — which implies that a is enabled and that the local process should commit to it — all other negotiation activities are terminated and a response $COMMIT(a)$ is sent back to the peer (which corresponds to transition 11 in Figure 10).

Algorithm 7 WaitingForCommit

Require: set of interactions $waitingSet$
  Input: set of interactions $possibleSet$ **Output:** interaction a
  if $waitingSet \neq \emptyset$ **then**
    choose $a \in waitingSet$ and **kill** main and **send** $COMMIT(a)$ and **send** $REFUSE(b)$ for all b in $possibleSet$ and **goto** $Busy(a)$
  else if $waitingSet = \emptyset$ and **receive** $COMMIT(a)$ and $a \in possibleSet \backslash toNegotiate$ **then**
    kill main and **send** $COMMIT(a)$ and **send** $REFUSE(b)$ for all b in $possibleSet$ and **goto** $Busy(a)$
  end if
  if receive $COMMIT(a)$ and $a \notin possibleSet$ **then**
    send $REFUSE(a)$
  end if

- $Main$ tries to commit to the first interaction found enabled (as a way to handle local conflicts) by activating $TryToCommit$ (transitions 4, 5 and 6 in Figure 10). $WaitingForCommit$ is terminated once $TryToCommit$ is activated, in order to avoid multiple commits at the same time.

- $TryToCommit(a)$ sends a $COMMIT(a)$ message to the corresponding peer and waits for a response (see Algorithm 8). Note that if $TryToCommit$ fails committing to a because it receives a $REFUSE$ message — in that case the peer has committed to a conflicting interaction — the process start again by checking the global readiness of its locally ready interactions. Indeed, as the peer has committed to another action its state may have changed. For the interactions a for which there exists at least one interaction with higher priority, the commit procedure is always initiated by the negotiator of a who is the first one to know about a's enabledness.

Algorithm 8 TryToCommit

Require: **Input:** interaction a **Output:** OK or NOK
  send $COMMIT(a)$
  if receive $COMMIT(a)$ **then**
    return OK and **send** $\forall b \in$ readySet$\backslash\{a\}$ $REFUSE(b)$
  else if receive $COMMIT(b)$ and $b \neq a$ and ($b \notin cycle(a)$ or($b \in cycle(a) \wedge P_b = Cyclebreaker$))
  then
    $waitingSet \longleftarrow waitingSet \cup \{b\}$
  else if receive $COMMIT(b)$ and $b \neq a$ and $b \in cycle(a)$ and $P_b \neq Cyclebreaker$ **then**
    send $REFUSE(b)$ and $readySet \longleftarrow readySet\backslash\{b\}$
  else if receive $REFUSE(a)$ **then**
    return NOK
  end if

- Finally, $AnswerNegotiators$ is always active if the process P_i is the negotiator for at least one interaction that dominates some other interaction. It receives messages of the form $READY(a)$ for interactions a for which P_i is the negotiator. It returns $READY(a)$ if a is currently in the $readySet$ of P_i, $NOTREADY(a)$ if it is in the $notReadySet$ or if it is not in its $possibleSet$, and otherwise defers the answer until the status of a is known.

4.2 Handling deadlocks

In order to avoid deadlocks due to decision cycles amongst interactions in conflict, we introduce the notion of *cycle*.

Definition 4.1. *We denote by* inter(a,P_1,P_2) *the fact that there exists an interaction a involving the two processes P_1 and P_2.*

A cycle*, C_A is a set of interactions $A = \{a_i\}_{i=1}^n$ for which the following holds: there exist n processes $\{P_i\}_{i=1}^n$, such that $\bigwedge_{i=1}^n$*inter(a_i,P_i,$P_{i+1 mod n}$)*. In addition we require that there exists at least one global state in which all conflicting interactions are enabled.*

A *cycle* C_A bears indeed a risk of deadlock or livelock in a state in which all interactions of C_A are enabled. Indeed, it represents a symmetric situation for all involved processes, where a process could wait forever for all others (deadlock) or propose a different choice than all others, reject it and start all over forever. This is a well-known problem in the context of communicating processes, in [3] a total order over the system interactions is defined, which allows to avoid deadlock by executing the interaction with higher order. In [16], a similar solution is proposed by imposing a total order over all processes, which breaks the cycle by executing the interaction proposed by the process with higher order.

The solution we propose is to detect statically the set of (minimal) cycles of the system. Then, in a second step, we define for each cycle statically a *Cyclebreaker*, which is one of the processes of the cycle. This particular process will arbitrate when a blocking situation actually occurs. This approach avoid to a define a total order of all interactions or processes which is useless if there is no cycle.

Illustrative example Figure 9 depicts an example representing a cycle. The system consists of 4 components: 3 processes $\{P_1, P_2, P_3\}$ forming a cycle C_A for the set of interactions $A = \{a, b, c\}$, and a completely independent process P_4. The existence of a cycle can be concluded from the structure and the

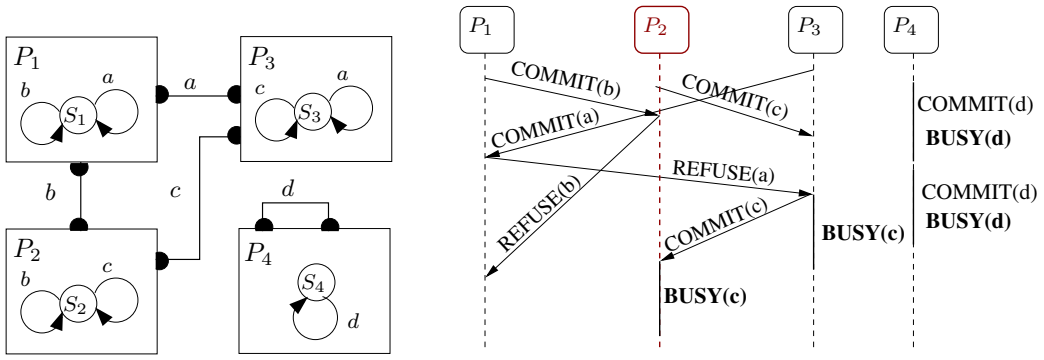

Figure 9: An example with cycle and independence

behaviors of the processes (the interactions a, b, c are always enabled). If no priority rules are defined on the set of interactions A, then C_A may lead to a deadlock. A possible deadlock scenario is depicted on the right side of Figure 9. This occurs when P_i sends a COMMIT message to P_{i+1} and waits for it. Which means that each component is waiting its peer who has made another choice. According to the proposed solution, let suppose that P_2 is chosen as the *Cyclebreaker* of C_A. According to Algorithm 8 (as described in Figure 9), whenever process P_i which is already engaged in committing an interaction and which receives a COMMIT for a different interaction, will send back a REFUSE message only if the COMMIT comes from a process which is not the *Cyclebreaker*. We will show that this breaks the cycle. Independently, the process P_4 can perform whenever it is possible the interaction d.

4.3 Correctness of the algorithm

We now prove that our algorithm guarantees the following properties:

1. Safety (exclusion), i.e., interactions in conflict cannot be committed simultaneously.
2. Liveness (progress), i.e., if an interaction is enabled, it will eventually become disabled either because it is executed or because a process offering it commits to another interaction.

To provide a proof check, we use the state transition diagram of Figure 10 where for a local controller, transitions represent steps of the algorithm and states represent the modes of the algorithm. Transitions may have a *guard* and an *action* and are depicted in Table 2.

The *action* defines the step (set of actions) and the *guard* represents a condition under which the step can occur.

Definition 4.2. *We denote by* waits(P_1, a, P_2)*, the predicate which holds when the process P_1 has sent a COMMIT(a) message to its peer P_2 involved in the interaction a but has not yet received an answer.*

We denote by $a \in cycle(b)$ the predicate that holds if it exists a cycle C_A such that $\{a, b\} \subseteq A$, which means that there exists a cycle involving the interactions a and b.

Lemma 2. *waits(P_1, a, P_2)$\wedge$ waits(P_2, b, P_3)* $\implies (a \notin cycle(b)) \vee (a \in cycle(b) \wedge P_1 = Cyclebreaker)$

Proof. waits(P_2, b, P_3) implies that P_2 is in state $Committing(b)$ (see Figure 10). waits(P_1, a, P_2) means that P_1 has sent a COMMIT(a) message to P_2. If P_2 does not answer this means that P_2 has performed the transition number 9. Indeed all other transitions in state $Committing(b)$ involve an action replying to P_1 by a REFUSE(a) message. Consequently, the guard of transition 9 holds which is according to Table 2: $(a \notin cycle(b)) \vee (a \in cycle(b) \wedge P_1 = Cyclebreaker)$. □

Lemma 3. *If* waits(P_1, a, P_2)*, then P_1 will receive a REFUSE(a) or a COMMIT(a) message within a finite delay.*

Proof. As we assume that all activities terminate and every message reaches its recipient within a finite delay, if P_1 does not receive a reply, after sending it a COMMIT(a) message, this means that the process P_2 is in the state $Committing(a_1)$ in Figure 10 ($a \neq a_1$) an P_2 remains in this state forever. We prove that this is impossible by *reductio ad absurdum*. If P_2 does not answer within a finite delay, then it is in the state $committing(a_1)$ which again means that there must exist an interaction $a_1 \neq a$ and a process P_3 such that waits(P_2, a_1, P_3). As there exists only n processes in the system, this means that there exists some cycle of size k of the form: waits(P_1, a, P_2)∧waits(P_2, a_1, P_3)∧ ... ∧waits(P_k, a_{k-1}, P_1). According to Lemma 2, we can infer the following properties:

$$(a \notin cycle(a_1)) \vee (a \in cycle(a_1) \wedge P_1 = Cyclebreaker)$$
$$(a_1 \notin cycle(a_2)) \vee (a_1 \in cycle(a_2) \wedge P_2 = Cyclebreaker)$$
$$\ldots$$
$$(a_{k-2} \notin cycle(a_{k-1})) \vee (a_{k-2} \in cycle(a_{k-1}) \wedge P_{k-1} = Cyclebreaker)$$

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

Lemma 4. *Let a be a an interaction of P. We have that the set A =* Conflict$_q(a)$ *is a set of interactions of P, that means A is in* possibleSet *in q and if b is in conflict with a in q then either b is an interaction of P or a is in prioritized conflict with b*

This holds because two interactions can only be in structural conflict if they share a common process.

Theorem 4.3 (Safety property)**.** *Let be q a state, a an interaction and denote $A = \{a_i\}_{i=1}^n$ the set Conflict$_q(a) \cup$ PrioConflict$_q(a)$ of interactions that are in conflict with a in state q. Our algorithm guarantees that if a is fired in state q, no interaction in A is fired in q.*

Proof. What we have to prove is that if in state q a process commits to interaction a, by executing one of the transitions 4, 5 or 6 of Figure 10, no interaction in A can be committed before the execution of a is terminated.

Suppose that $a \in \mathcal{P}_i$. According to Lemma 4, for all $b \in A$ we have either $b \in \mathcal{P}_i$ or $b \in$ *prioConflict$_q(a)$*, and we prove the theorem separately for these two cases.

1. First case: $b \in \mathcal{P}_i$, that is a and b share the same process P_i. First of all, only interactions committed by both peers are executed. Then, if P_i has sent a COMMIT(a) message executing one of the transitions 4, 5 or 6 of Figure 10, then according to the same table it is impossible to send a COMMIT(b) message before either a $REJECT(a)$ is received or the BUSY state is entered, then exited and the next state reached.

2. Second case $b \in$*prioConflict$_q(a)$* holds, that is a and b are concurrent (and thus belong to different processes) and either $a < b$ or $b < a$. Suppose that P_j is the negotiator for b.

   If $b < a$, then b should not be executed before the execution of a — which has started — has been completed and P_i enters READY for the successor state of q. We have now to proof that from that moment on P_j cannot "believe that a is *not ready*" which is the condition for committing to b.

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

   Now, we must prove that P_j cannot have old, depreciated knowledge that a is not ready. This can only be the case, if at some point a was not ready and P_i has sent $NOTREADY(a)$ to P_j, and then transitions concurrent to b have been executed leading to the current state q in which a is ready and executed, and P_j may use incorrect knowledge and execute b. This corresponds exactly to a situation of confusion, which we have excluded.

   If $a < b$, the situation is almost symmetric. We must prove that in this case b is not ready. If P_i is the negotiator for a, asks the negotiator of b whether b is ready, and only if the answer is negative, it will

consider a to be enabled and may initiate the commitment of a. Again, only if confusions exist P_i may use old knowledge. If the negotiator of a is the peer, then P_i will only commit to a on reception of a $COMMIT(a)$ from its peer which uses the same procedure for deciding to commit to a.

□

Theorem 4.4 (Liveness property)**.** *Let a be a* enabled *interaction. Our algorithm guarantees that a will eventually become disabled.*

Proof. An enabled interaction a may become disabled because it is executed or because a process offering it commits to another interaction. When a is enabled for a Process P_i, a COMMIT(a) message is sent to the corresponding peer and P_i goes to state $committing(a)$. a becomes disabled when P_i leaves this state. In other words what we have to prove is that P_i cannot stay in this state eternally. When P_i is in state $committing(a)$, there must exists a process P_j such that *waits(P_i, a, P_j)*. Thus the proof follows directly from Lemma 3. In fact, when receiving message COMMIT(a) or REFUSE(a), P_i will leave state $committing(a)$ through transition 7 or 8.

□

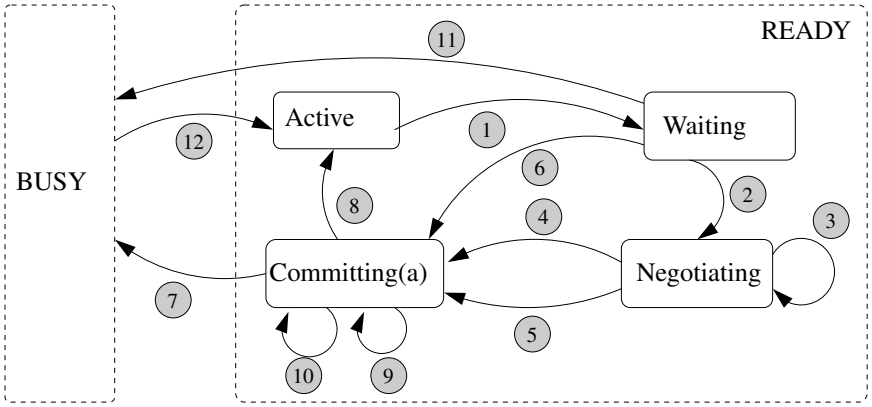

Figure 10: State diagram of the algorithm

4.4 Efficiency of the algorithm

Choosing the negotiators For each interaction a involved in at least one priority rule, we choose one of the processes involved in a as its negotiator which will send requests to negotiators of interaction with higher priority and answer request from negotiators for lower priority interactions. A process may be the negotiator for several interactions. Various strategies may be proposed to allocate negotiators to processes. The criterion we use is to minimize for each interaction the maximal number of distinct processes to which its negotiator has to send requests. This is meant to minimize the number of communications added due to priorities.

As already explained, local priorities — that means when $a < b$ and a and b have negotiators hosted by the same process P_i — are decided locally. In the dining philosophers example (see figure 3), all the priority rules involve the process $Forks$, which will thus be designated as negotiator and it will enforce priorities locally. The priorities for the travel agency (see Figure 1 in Section 1) may include priorities amongst interactions between travel agencies and clients which do not necessarily have a common process. In this case, for each interaction we choose as negotiator the process that is involved in the biggest set of interactions, as it may have more knowledge about readiness of more interactions.

Transition	Guard	Action
1	possibleSet$\neq \emptyset$	$\forall a \in$ possibleSet, send(POSSIBLE(a))
2	receive(POSSIBLE(a))$\wedge$ $a \in$possibleSet$\cap$toNegotiate	call(Negotiate(a)$\wedge$readySet:=readySet$\cup\{a\}$
3	receive(POSSIBLE(a))$\wedge$ $a \in$possibleSet$\cap$toNegotiate	call(Negotiate(a)$\wedge$readySet:=readySet$\cup\{a\}$ $\wedge$ $\forall b \in$lessPrio(a), kill(Negotiate(b))
4	Negotiate(a)=ok	send(COMMIT(a))$\wedge$ kill(WaitingForCommit)$\wedge$ $\forall b$ kill(Negotiate(b))
5	receive(POSSIBLE(a))$\wedge$ $a \in$prioFree	send(COMMIT(a))$\wedge$ kill(WaitingForCommit)$\wedge$ $\forall b$ kill(Negotiate(b))
6	receive(POSSIBLE(a))$\wedge$ $a \in$prioFree	send(COMMIT(a))$\wedge$ kill(WaitingForCommit)
7	receive(COMMIT(a))$\wedge$Committing(a)	goto(BUSY(a))$\wedge$ $\forall b \in$readySet, send(REFUSE(b))
8	receive(REFUSE(a))$\wedge$Committing(a)	goto(Active)$\wedge$reset(readySet)$\wedge$ keep(possibleSet)
9	receive(COMMIT(b))$\wedge$Committing(a) $\wedge(a \neq b)\wedge(b \notin$cycle($a$) $\vee$ ($b \in$cycle(a)$\wedge$P$_b$=Cyclebreaker))	waitingSet:=waitingSet$\cup\{b\}$
10	receive(COMMIT(b))$\wedge$Committing(a) $\wedge(a \neq b)\wedge$ ($b \in$cycle(a)$\wedge$P$_b \neq$Cyclebreaker)	send(REFUSE(b))$\wedge$readySet:=readySet$\backslash\{b\}$
11	receive(COMMIT(a))$\wedge a \in$ possibleSet$\backslash$toNegotiate	send(COMMIT(a))$\wedge$ $\forall b \in$possibleSet and $b \neq a$, send(REFUSE(b))
12	true	set(possibleSet)

Table 2: Transitions of the protocol state diagram

Efficiency and experimental results We have implemented our algorithm using Java 1.6 and Message Passing Interfaces (MPI) in order to experiment the efficiency of the algorithm on examples of different nature. We are also interested in comparing it to other solutions. The comparison to other solutions is not easy, as no other algorithm takes into account priorities, and handling priorities is the most costly part of the algorithm. Letting priorities aside, the comparison to algorithms which provide distributed implementations in the domain of webservices, such WSDL or BEPL, the comparison is almost impossible, because we start from truly global specifications and provide a generic solution for distribution, whereas in those languages the starting point is a much larger specification which is already almost a distributed solution that has been hand crafted for the problem at hand, and is therefore likely to be more efficient, of course.

α-core is an algorithm solving almost the same problem — except that it does not handle priorities — and therefore we found a comparison between our algorithm and α-core on a specification without priorities the most relevant one, where we compare the number of messages needed to execute given interactions.

We used the example of Section 4.2 and depicted in Figure 9 for our experiments. We have generated distributed implementations of this example using our algorithm and an implementation of the α-core algorithm, and then run and compared the implementations generated.

The example consists of three process in which 2 interactions are permanently enabled until each of them has been executed k times, and then the process reaches a final state. all interaction are in conflict with all others enabled interactions.

Table 3 gives the number of messages exchange by a complete run of the implementation generated by each algorithm, for several values of the parameter k.

As shown in Table 3, the number of messages of our algorithm is significantly smaller than the number of messages in the implementation generated by the α-core algorithm. The reason for that is that α-core is "connector-centric", that is it creates an additional process for each interaction whereas our algorithm is process centric, that is all negotioations are hosted by some process.

[16] for multiparty interactions (note that our algorithm can be easily modified to handle multiparty

interactions, see Section 4.5).
If we restrict the set of priority rules to local priorities — priorities are defined only on interactions having at least one process in common — then common processes will enforce priorities locally (and no messages for negotiation purposes are added). In this case, our algorithm is similar to the one proposed in [3] where static local priorities are defined to deal with conflicts.

Number of executions	1	10	100
Number of messages for our algorithm	18	199	1850
Number of messages for α-core algorithm	30	510	4969

Table 3: Comparison with α-core algorithm

As we do not assume knowledge about message transmission time, we measure efficiency of our algorithm in terms of the number of messages a process has to exchange to execute a given interaction. Where we are interested here in only in causal sequences of messages and ignore the messages that are exchanged concurrently and do not lead to success. When global priorities are introduced and there are chains of priorities with no common process, then negotiators are necessary to decide enabledness. Additional communication is needed between negotiators (2 messages for each negotiation, see Algorithm 6). In this case the efficiency (as defined above) of our algorithm depends on the strategy chosen to assign negotiators. The criterion given previously to choose negotiators intends to minimize the number of processes involved in the negotiations for an interaction, which is consistent with the usual criteria to measure efficiency of algorithms. In fact, these criteria state that 1) the number of processes involved in determining whether any given interaction can be executed should be as small as possible 2) there is a bound on the number of messages needed for an interaction to be disabled. Choosing negotiators is done with respect to criterion 1). Criterion 2) is ensured by the progress property of our algorithm and the number of messages required, in the worst-case, for an interaction a to be fired is computed as follows:

Consider two processes P_1 and P_2 and assume that eventually they commit to a. Let I_1 be the number of locally ready interactions of P_1. Without loss of generality, we can assume that P_1 is the negotiator of all its locally ready interactions and let Ng_b be the number of negotiators needed to check the enabledness of $b \in I_1$. In the worst case, a will have the lowest priority over the $possibleSet$. Thus P_1 needs $2 \times I_1$ messages to check readiness. It then checks enabledness of all its globally ready interactions, which implies $2 \times (\sum(Ng_b)_{b \in I_1}) \times I_1$ messages. Then, P_1 tries to commit to all its interactions and receives a $REFUSE$ except for a, for which it will receive a $COMMIT$ message. This means that in order to fire a, our algorithm generates in the worst case $(2 \times I_1) + (2 \times (\sum(Ng_b)_{b \in I_1}) \times I_1) + 2 \times (I_1 - 1) + 2 = 4 \times I_1 + 2 \times (\sum(Ng_b)_{b \in I_1}) \times I_1$.
Notice that if there is no negotiation, the number of messages is reduced to the first term $4 \times I_1$ which is as shown in Table 3 clearly smaller than the algorithm proposed in [16].

4.5 Extension to multiparty interactions

In Section 4.1, we have presented an algorithm for handling binary synchronisations. Extending it to n-ary synchronisations does not modify how priorities are handled, and extending binary interactions to multiparty interactions can be done similar as in α-core. In thsi case, every interaction has a negotiator with the additional task of negotiating global readiness. This corresponds to the role of the entities called *coordinators* assigned to each interaction in [16] or to *interaction managers* assigned to a subset of interactions in [2]. The criterion to assign negotiators could still be the same as proposed in Section 4.4.

Algorithm $Negotiate$ is unchanged as each negotiator has to ask other negotiators about the readiness of a given interaction, this does not depend on the number of processes involved in an interaction. The rest of algorithms proposed in Section 4.1 have to be slightly modified to deal with multiparty interactions. For this purpose we propose that, for each interaction a, the corresponding negotiator collects the responses of all the processes involved in a and checks that all of them are ready to execute the interaction. This is done using the exchange of messages $POSSIBLE$. We propose to add two new messages:

- $START(a)$ message sent by the negotiator of a to inform all other processes involved that a could be fired;

- $CANCEL(a)$ message sent by the negotiator of a to the participants to inform them that the interaction cannot be fired.

In the Algorithm 7, $WaitingForCommit$, whenever the process P receives a $COMMIT$, it kills thread $Main$, sends back a $COMMIT$ and waits for $START$. If it receives the $START$ message, it executes the interaction. If it receives a $CANCEL$ message, it restarts the Main thread again. Note that the operations performed in this algorithm concerns only interactions for which P is not the negotiator.

In the Algorithm 8, $TryToCommit$, the process sends a $COMMIT$ message to all participants involved in the interaction and waits for a $COMMIT$ answer from all of them. If it receives at least one $REFUSE$ message, it sends back $CANCEL$ message to all participants. If it receives $COMMIT$ from all participants, it sends back $START$ to all of them and goes to state $Busy$ to execute the corresponding interaction. Note that all the operations performed in this algorithm concerns only interactions for which P is the negotiator.

5 Related work

Concerning the comparison to different approaches, there are two main topics which merit to be discussed. One concerns synthesis of distributed controllers from global specifications and one to the intended application domain, and concerns the comparison to existing approaches in the domain of webservices, both in terms of expressiveness of the specification framework and the resulting distributed implementation.

First, we discuss in a bit more detail than in Section 1, the originality of the main algorithm presented in Section 4 and its interest and deficiencies with respect to related algorithms. Several algorithms realizing a semantic preserving transformation from Petrinets or process algebra terms into a set of processes communicating by message passing have been proposed in the past. A classical algorithm is the one of [3] which handles binary rendez-vous synchronizations like ours. As already mentioned, this algorithm uses a statically defined order of the interactions of each process, and each process tries to initiate locally ready interactions by following this order. This may save unnecessary communications if early tries lead often to success. In our algorithm, we handle all locally ready interactions concurrently, that is we give priority to the fastest that may lead to success. We will on the average use more communications but be able to trigger the next interaction in a shorter delay.

A more recent algorithm is α-core which handles multi-party synchronizations and transforms each synchronization into a process exchanging messages with the set of interacting processes and the set of "coordinators" of potentially conflicting synchronizations. This means that the synchronization "coordinators" have to acquire all necessary knowledge by message passing. We present in details an algorithm for binary synchronizations, for which there is no need for an explicit "coordinator" to correctly achieve synchronizations. In the multi-party version of our algorithm, we also introduce an explicit "coordinator" for each synchronization as this leads to less communications. But, as a difference to α-core, it is always the case that one of the processes involved in the synchronization “hosts” this "coordinator", and therefore the "coordinator" can profit from the knowledge of this process to avoid certain communications: a "coordinator" may for example get “for free” the knowledge that some conflicting transitions are not enabled or that a transition with higher priority is enabled, which may avoid useless communications.

Both algorithms use a static global order over the set of processes to break decision cycles. We propose a more flexible solution defining a process that will “break the cycle” for each possible decision cycle. We are not aware of any algorithm of similar nature which handles global priorities.

Algorithms for distributing prioritized specifications as we consider them here are proposed in [4, 22] where the main motivation is to select a possibly small subset of executions with the ambition to avoid — totally or as much as possible — any communication in addition to those of the α-core algorithm used for executing synchronizations. These approaches propose the use of statically computed *knowledge* about the possible global states in each local state of an individual process, and if this set of global states is sufficiently discriminating, no communication with peers may be needed to know about the enabledness of some transition. In [22] it is proposed to communicate only when local knowledge is not sufficient. In this sense, our algorithm is quite similar to this one. We propose to choose the interactions with faster successful negotiation, and there those for which negotiation is not needed at all. However, with the significant

difference that there this principle is only applied to handle priorities whereas we handle synchronizations and priorities in a uniform manner. Here, we do not use any statically pre-calculated knowledge, but the dynamic knowledge induced by the communications for calculating the readiness of interactions is exploited in our algorithm. Our algorithm could certainly gain when available static knowledge is exploited. The nature of the knowledge that is interesting would however be a little bit different as in [22] only knowledge about priorities is relevant whereas here we are also interested about readiness of interactions. In fact, we would even be interested knowledge about necessary future enabledness of interactions as our algorithm is in terms of buffered message exchanges.

We want to use synchronizations and priorities for specifying and verifying webservices. We have already explained in Section 1 why we think that this is a good idea and why this leads to different challenges for achieving distribution as those considered generally in the domain of webservices such as, to cite only a few, [8, 17, 1, 12, 14]. Also this does not only hold for typical webservice specifications which are already expressed in terms of "oriented" two-party interactions with a well-determined initiator, also holds for formalisms such BEPL, which may look formally quite similar, as they resemble Petrinets. Nevertheless, these Petrinets are generally used in a particular way. They specify set of tasks to be executed, the order constraints and the potential concurrency amongst them. But the tasks themselves are atomic and not distributed. .

The challenge that we target with our approach is providing some new emerging service through the composition of a set of existing services which may execute a set of task on their local memory and they may impose constraints on the order in which these tasks can be executed. The new service is then defined by set of service components, and a set of synchronizations and priorities. If such a service specification contains a component that is involved in all interactions and that imposes order constraints on them, this component corresponds than typically to what is called an orchestrator in the domain of webservices. This process would then also interact with the client(s) In absence of such a centralizing process, the emerging service is given in the form of a choreography.

High-level functional and non-functional correctness properties are to be verified on this composition, or even on a composition with the expected client(s). This is the reason why we want to keep them as concise and readable as possible, and whenever appropriate, describe interactions amongst several components that should be executed in an atomic fashion by a rendez-vous rather than describing them in the form of some protocol. In this respect, our approach is more efficient as it allows writing more concise specifications. On the other hand, an obvious drawback is that a generic protocol implementing systems with arbitrary multi-party interactions and global priorities is likely to be less efficient than a hand-crafted protocol for a given purpose and a given set of components. An advantage of our protocol would be — in the case that the total number of messages exchanged is not a problem — to reach faster a point where a next interaction can be executed, because any of the interacting components will play the role of the *initiator* of the protocol if it is the first one who is ready for it. Another advantage is that we systematically explore all potentially enabled interactions so as to guarantee quick convergence. A hand-crafted protocol with the same ambition is likely to not send much less messages. To avoid exchanging of useless messages, improving the algorithm by adding a first phase of knowledge computation to find out when a locally enabled interaction is guaranteed to be (not) enabled might in some cases significantly reduce the number of message exchanges — without reducing the potential degree of concurrency.

6 Conclusion and future work

In this paper we propose two algorithms useful in the domain of service composition where services are represented by extended transition systems, where transitions represents tasks or subservices and the transition system represent constraints on the order in which these tasks can be executed. Services are composed by multi-party synchronizations — realizing data exchange — and by global priority rules imposing additional constraints on the order in which interactions, and therefore tasks, have to be executed.

The first algorithm tries to eliminate deadlock from a composed specification by additional priority constraints, thus eliminating bad non-determinism. This algorithm cannot always succeed, in which case the user will be required to rework the specification, but may avoid rework in several situations. More experimentation is needed to assess the actual usefulness of this algorithm.

The second, main algorithm defines a transformation of such a global service composition into a distributed composition, in which every component is composed with a local controller exchanging messages with its peer controllers in order to realize interactions exclusively by message exchange. The algorithms is proven correct, where by correctness we understand that any sequence of interactions that can be observed on a distributed execution (obtained by linearization) can also be observed on the global specification, and vice versa. We have implemented a version of this algorithm handling only binary interactions. We plan to implement also the protocol for multi-party interactions, to include data transfer and to exploit static knowledge for reducing the number of messages to be sent. More experimentation could be useful in order to compare the complexity of our algorithm to one of similar algorithms.

More experimentation and further research is needed in particular for experimenting with this type of specifications for web services. We expect that these kind of specifications may be useful and accepted for embedded applications, where webservices are only emergent, and where tasks depending on multiple resources are an issue to be dealt with at a high level of abstraction. We do not believe that the BIP language as such would be accepted, as a set of connectors and priorities for a given set of components may now be considered as a composed service, and therefore needs to be represented and named explicitly, and it will be required to create (dynamically) multiple instances of both basic and composed services. This requires new design concepts derived from the basic concepts considered here.

References

[1] Marco Autili, Michele Flammini, Paola Inverardi, Alfredo Navarra, and Massimo Tivoli. Synthesis of concurrent and distributed adaptors for component-based systems. In *EWSA*, volume 4344, pages 17–32, 2006. 5

[2] Rajive Bagrodia. Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Trans. Software Eng.*, 15(9):1053–1065, 1989. 4.5

[3] Rajive Bagrodia. Synchronization of asynchronous processes in CSP. *ACM Trans. Program. Lang. Syst.*, 11(4):585–597, 1989. 4.2, 4.4, 5

[4] Ananda Basu, Saddek Bensalem, Doron Peled, and Joseph Sifakis. Priority scheduling of distributed systems based on model checking. In *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 79–93. Springer, 2009. 1, 5

[5] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in BIP. In *Proc. of SEFM'06*, pages 3–12. IEEE Computer Society, 2006. 1, 2

[6] Saddek Bensalem, Marius Bozga, Joseph Sifakis, and Thanh-Hung Nguyen. Compositional verification for component-based systems and application. In *Proc. of ATVA'08*, volume 5311 of *LNCS*, pages 64–79, 2008. 1

[7] Simon Bliudze and Joseph Sifakis. The algebra of connectors: structuring interaction in BIP. In *Proc. of EMSOFT'07*, pages 11–20. ACM Press, 2007. 2

[8] Javier Cámara, José Antonio Martín, Gwen Salaün, Javier Cubo, Meriem Ouederni, Carlos Canal, and Ernesto Pimentel. Itaca: An integrated toolbox for the automatic composition and adaptation of web services. In *ICSE*, pages 627–630, 2009. 5

[9] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985. 3

[10] Gregor Gößler and Joseph Sifakis. Composition for component-based modeling. *Sci. Comput. Program.*, 55(1-3):161–183, 2005. 1, 2

[11] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1984. 2

[12] Jan Mendling and Michael Hafner. From inter-organizational workflows to process execution: Generating bpel from ws-cdl. In *OTM Workshops*, volume 3762 of *Lecture Notes in Computer Science*, pages 506–515. Springer, 2005. 5

[13] R. Milner. A calculus of communication systems. In *LNCS 92*. Springer, 1980. 2

[14] Mangala Gowri Nanda, Satish Chandra, and Vivek Sarkar. Decentralizing execution of composite web services. In *OOPSLA*, pages 170–187, 2004. 5

[15] Luca Padovani. Contract-directed synthesis of simple orchestrators. In *Proc. of CONCUR'08*, volume 5201 of *LNCS*, pages 131–146, 2008. 1, 3

[16] José Antonio Pérez, Rafael Corchuelo, and Miguel Toro. An order-based algorithm for implementing multiparty synchronization. *Concurrency - Practice and Experience*, 16(12):1173–1206, 2004. 1, 4.2, 4.4, 4.4, 4.5

[17] Zongyan Qiu, Xiangpeng Zhao, Chao Cai, and Hongli Yang. Towards the theoretical foundation of choreography. In *WWW*, pages 973–982, 2007. 5

[18] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1):206–230, 1987. 3.1

[19] S. L. Ricker and K. Rudie. Know means no: Incorporating knowledge into discrete-event control systems. *IEEE Transactions on Automatic Control*, 45:1656–1668, 2000. 1

[20] K. Rudie and W.M. Wonham. Think globally, act locally: Decentralized supervisory control. *IEEE Transactions on Automatic Control*, 37:1692–1708, 1992. 1

[21] Gwen Salaün and Tevfik Bultan. Realizability of choreographies using process algebra encodings. In *Proc. of IFM'09*, volume 5423 of *LNCS*, pages 167–182, 2009. 1

[22] Doron Peled Susanne Graf and Sophie Quinton. Achieving distributed control through model checking. In *CAV*, 2010. 1, 5