From Orchestration to Choreography: Memoryless and Distributed Orchestrators

Sophie Quinton, Imene Ben-Hafaiedh and Susanne Graf

Université Joseph Fourier / VERIMAG,
Grenoble, France
{quinton, benhfaie, graf}@imag.fr

Abstract. In the context of Web services, making client and service interact so as to satisfy the client, that is, making the service *compliant* with the client, can be done using either orchestration or choreography. In this paper we propose to build, whenever possible, memoryless orchestrators, and then distribute them using protocols so as to obtain choreographies.
When necessary for guaranteeing compliance, we infer from the initial (sequential) transition system possible concurrency between certain interactions, whose validity must be checked by the designer. Our approach allows for a clear distinction between the design phase and the implementation phase while being, in the general case, more efficient than orchestration. An example dealing with resource management illustrates the usefulness of memoryless orchestrators. We also discuss a methodology allowing contract-based design and verification of Web services at a higher level of abstraction.

1 Introduction

When designing Web services, important issues are those of *compliance* with a set of potential clients, and preservation of compliance by *refinement* (see [1], [2]). Indeed, in order to prove compliance for a large class of clients and services based on the use of abstractions of real clients and services, it is crucial to determine a notion of refinement for clients and for services (potentially different).

Here we focus on another issue specific to Web services, namely, deciding whether the *service adaptation* for a given class of clients and services should be realized by an *orchestration* or a *choreography*.

An orchestrator is a global mediator between the service and client components. Orchestrators can be automatically synthesized for given abstract definitions of client and service, see e.g. [2]. Their main drawback is that they are centralized, that is, all interactions between service and client components are controlled by the orchestrator. Very often, this is not needed. Let us consider, e.g., n clients sharing k resources. It is possible to predefine for each client a preferred "local" resource which can be obtained through local negotiation. Thus only in the rare cases where the preferred resource is not available would a more global communication be needed.

Choreographies are distributed controllers and avoid that problem. The problem there lies in their implementation. In [3], e.g., *realizability* of choreographies is studied and can be enforced by extra communications. However, the expressivity of the given specifications is limited.

Note that in general, distribution of an orchestrator leads to a very inefficient choreography. The reason is that orchestrators tend to be over-constrained with respect to message orderings. As a result, large numbers of messages are needed to provide every component with the required knowledge about the global state to decide about the next global interaction to be taken.

More simply, we restrict the class of orchestrators which we try to distribute to those definable by a memoryless orchestrator represented by a priority order and a set of local bounded buffers. The BIP framework [4,5] is a natural candidate for this purpose as it allows naturally representing clients and services as component behaviours, their interactions as the interaction layer and memoryless orchestrators as the priority layer (see section 2).

The priority rules representing orchestrators can be computed in two phases: the first one focuses on violations of compliance that can be resolved by adding finite (reordering) buffers. The second phase infers, if possible, a set of static priorities between interactions making the remaining deadlock states unreachable.

Properties expected from Web services are analyzed on this generated BIP model, which is still not too abstract and for which efficient verification methods exist (e.g. structural ones [6], [7][1]). The synthesized

[1] Those approaches do not require the computation of the global state space, although we construct it anyway so far to generate the memoryless orchestration.

memoryless orchestrators can then be implemented by means of protocols communicating amongst each other as locally as possible, as determined by static analysis. When there are no (or few) interactions dominated by a priority, that is, if the given client and service were initially (almost) compliant — up to some reordering being achieved by local buffers, the implemented system will be as efficient as a rendez-vous based choreography (e.g. [3]). When components have in many states a deterministic interaction set, as little as a two-way message exchange between the two peers may then suffice to realize an interaction (see Section 4).

On the other hand, there are cases in which the implemented system will be less efficient than a memoryful orchestrator. This may typically happen if the depth of the priority order is large, leading to global interactions. The other key issue with choreographies is the existence of global choices. In this respect, our algorithm is rather efficient, at least with respect to the criterion we have chosen here: make progress as quickly and as locally as possible, also at the cost of additional, potentially useless, message exchanges. Additional (static) analysis of specifications may allow decreasing the message overhead. For example, knowledge about persistency of readiness or enabledness of certain interactions avoids multiple requests for the same information, or allows choosing a preferred initiator for an interaction.

We believe that an interesting contribution of our work is the clear separation between the design phase of Web services, in which all the important properties are guaranteed by using our proposed synthesis algorithm and additional verification of any required safety or progress property, and the implementation phase which is expected to be fully automated. This distinction hopefully allows obtaining more abstract specifications and thus more efficient verification. We propose to go a step further by designing Web services using structured interactions as in BIP.

The paper is organized as follows: Section 2 shortly presents the BIP framework on which our intermediate representation of memoryless orchestrators relies. Section 3 explains how memoryless orchestrators are synthesized while Section 4 sketches how they are implemented by protocols. Finally, Section 5 discusses a possible design methodology integrating verification and implementation of Web services and discusses refinement issues.

2 The BIP framework

In [4,5,8], the framework BIP for component-based design and verification has been proposed. In BIP, systems are built by superposing three layers of modeling: Behavior, Interaction, and Priority. The classic notion of input/output of synchronous frameworks is replaced by the more expressive notion of multi-party interaction which allows each of the involved interaction partners to impose constraints on when the interaction may take place and imposes no notion of input completeness.

BIP is related to process algebras such as CCS [9] or CSP [10] by its rendez-vous-like interaction mechanism (on a set of *ports*) and the restriction to a strictly local notion of state. It has been shown however in [11] that the set of composition operators that can be defined within the BIP framework is, according to a definition taking into account the ability to coordinate components, more expressive than composition operators of CCS, CSP and SCCS [12]. BIP also addresses the problem of composition of operators and of their properties, which can be exploited for structural verification [8].

For the sake of clarity, we present here a simplified version of the BIP framework, without variables, guards or data transfer.

Definition 1 (Labeled Transition System). *A* Labeled Transition System *(LTS) is 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* $\delta \subseteq Q \times 2^{\mathcal{P}} \times Q$ *is a transition relation.*

LTS are used to represent behaviors of components. In this context, labels refer to the *ports* with which transitions are associated. Let us emphasize that transitions are labeled by sets of ports because a component or subsystem may be able (or even required) to interact through several ports simultaneously.

Components K interact via their ports. An *interaction* is characterized by the set of ports which synchronize, generally involving more than one component. A *connector* c is characterized by a set of ports and a set of interactions, describing all possible synchronizations involving the port set of c. Typical connectors represent rendez-vous or broadcast but also mutual exclusion, when only interactions involving a port p of a resource and the corresponding $\bar{p}$ of a single component are part of c.

Definition 2 (Interaction model). *A connector* c *is a set of interactions; we denote* $ports(c)$ *the set of ports that are involved in at least one of the interactions in* c*. An* interaction model *on a set of ports* $\mathcal{P}$ *is a set of connectors* $\mathcal{C}$*.*

Definition 3 (Legal interactions). *The set of* legal *interactions of an interaction model $\mathcal{C}$, denoted $\mathcal{L}(\mathcal{C})$, is $\{\bigcup_{l=1}^{k} i_l \mid k > 0 \wedge \forall l = 1..k, \exists c_l \in \mathcal{C}, i_l \in c_l,\ c_l$ pairwise distinct connectors $\}$.*

According to this definition, any combination of interactions of different connectors is a legal interaction. This notion encompasses concurrency, as interactions from different connectors may be fired simultaneously as well as not, unless stated otherwise by the component's LTS.

Definition 4 (Priority order). *A* priority order *on an interaction model $\mathcal{C}$ is a strict partial order such that:*

- $\forall c \in \mathcal{C}, \forall i, j \in c, i \subseteq j \implies i < j$
- $\forall i, j, \alpha \in \mathcal{L}(\mathcal{C}), i < j \wedge c(\alpha) \cap (c(i) \cup c(j)) = \emptyset \implies i \cup \alpha < j \cup \alpha$

where for any interaction i_1, $c(i_1)$ is the set of connectors that contain an interaction i_2 such that $i_2 \subseteq i_1$.

Priorities are used to arbitrate between simultaneously enabled interactions, for example to enforce scheduling policies. The first condition above ensures maximal progress between interactions of a single connector. The second condition ensures monotonicity, that is priorities are preserved from smaller to larger interactions.

Definition 5 (BIP system). *A system represented in BIP consists of a set of components K_i whose behaviors are given by LTS on disjoint port sets $\mathcal{P}_i$, an interaction model $\mathcal{C}$ on $\mathcal{P} = \bigsqcup_{i=1}^{n} \mathcal{P}_i$ and a priority order $<$ on $\mathcal{C}$.*

We refer to the three items thus defined as layers which are called respectively *behavior*, *interaction* and *priority*. We refer to $(\mathcal{C}, <)$ as a *composition operator* on $\mathcal{P}$, because interaction and priority express how to compose a set of LTS so as to make them interact. As usual, $q_1 \xrightarrow{\alpha} q_2$ denotes $(q_1, \alpha, q_2) \in \delta$ and $q_1 \xrightarrow{\alpha}$ denotes $\exists q' \in Q, q \xrightarrow{\alpha} q'$.

Definition 6 (Operational semantics). *Let $\{K_i\}_{i=1}^{n}$ be a set of LTS, where $K_i = (Q_i, q_i^0, \mathcal{P}_i, \delta_i)$. Let $\mathcal{P} = \bigsqcup_{i=1}^{n} \mathcal{P}_i$ and $\{\mathcal{C}, <\}$ a composition operator on $\mathcal{P}$. The composition of $\{K_i\}_{i=1}^{n}$ with $\{\mathcal{C}, \pi\}$ is an LTS $(Q, q^0, \mathcal{P}, \delta)$ such that $Q = \prod_{i=1}^{n} Q_i$, $q^0 = (q_1^0, \dots, q_n^0)$ and δ is defined as follows.*
$\forall \alpha \in \mathcal{L}(\mathcal{C}), \forall q^1 = (q_1^1, \dots, q_n^1), q^2 = (q_1^2, \dots, q_n^2) \in Q, q^1 \xrightarrow{\alpha} q^2$ iff:

- $\forall i, q_i^1 \xrightarrow{\alpha_i} q_i^2$ *where* $\alpha_i = \alpha \cap P_i$ *and with the convention that* $\forall q, q \xrightarrow{\emptyset} q$
- $\nexists \alpha' \in \mathcal{L}(\mathcal{C}), \alpha < \alpha' \wedge q^1 \xrightarrow{\alpha'}$

Thus, only interactions that are locally enabled in all concerned components, and furthermore not inhibited by an interaction with higher priority, may be fired. Independent interactions may be fired jointly .

3 Synthesis of memoryless orchestrators

3.1 Contracts for Web services

Web services communicate via peer-to-peer exchange of messages. We assume that no message can be lost or reordered when sent through the same channel. An elegant way to represent message loss or absence of interaction peers in BIP is discussed in section 5. In this paper, we propose only time independent algorithms. In particular, we do not use on timeouts to guarantee progress. The framework could be extended to systems satisfying real-time requirements by also adding constraints on execution times and transmission delays.

In the context of Web services, *contracts* are used to describe how a client or a service is expected to behave. As in [2], we use a fragment of CCS to define such contracts. . We use as the input for our methods the LTS defined by a CCS term. In the following, x is a process variable, α denotes an action, and σ, ρ are process terms.

$$\sigma ::= 0 \mid \alpha.\sigma \mid \sigma + \sigma \mid \sigma \oplus \sigma \mid \mathtt{rec}\, x.\, \sigma \mid x$$

The terminated process is represented by 0, and $\alpha.\sigma$ represents sequence. As usual, $+$ denotes the *external* choice (whether the process $\sigma + \rho$ will behave as σ or ρ depends on the environment), $\oplus$ the *internal* (non-deterministic) choice. Variables are assumed to be guarded, that is, every free occurrence of x in a term $\mathtt{rec}\, x.\, \sigma$ appears in a subterm of the form $\alpha.\rho$. This ensures that no sequence of internal transitions is infinite. The process $\mathtt{rec}\, x.\, \sigma$ behaves like $\sigma\{^{\mathtt{rec}\, x.\, \sigma}/_{x}\}$, which is σ with every free occurrence of x replaced by $\mathtt{rec}\, x.\, \sigma$.

The set of actions appearing in σ is denoted $act(\sigma)$. The semantics of σ can be defined by an LTS $(Q, q^0, \mathcal{P}, \longrightarrow)$, where states are terms of σ: Q is the set of derivations of σ, $q^0 = \sigma$, $\mathcal{P} = act(\sigma) \cup \{\tau\}$ and $\longrightarrow$ is the transition relation given by the following SOS rules (symmetric rules for $+$ and $\oplus$ are omitted).

$$\alpha.\,\sigma \xrightarrow{\alpha} \sigma \qquad\qquad \sigma \oplus \rho \xrightarrow{\tau} \sigma \qquad\qquad \mathtt{rec}\,x.\,\sigma \xrightarrow{\tau} \sigma\{^{\mathtt{rec}\,x.\,\sigma}/_{x}\}$$

$$\frac{\sigma \xrightarrow{\tau} \sigma'}{\sigma + \rho \xrightarrow{\tau} \sigma' + \rho} \qquad\qquad \frac{\sigma \xrightarrow{\alpha} \sigma'}{\sigma + \rho \xrightarrow{\alpha} \sigma'}$$

Actions can be either outputs (denoted $\bar{a}$) or inputs (denoted simply a). By $\sigma \parallel \rho$ we denote the composition of σ and ρ where inputs and corresponding outputs must synchronize, which in BIP can be represented by a set of connectors of the form $\{\{\bar{a}, a\}\}$. In the sequel, when no ambiguity is possible, we refer to the connector $\{\{\bar{a}, a\}\}$ or to the interaction $\{\bar{a}, a\}$ as a.

Satisfaction of a client is represented by a special action e. A client is satisfied if it offers e. To check whether a given client and a given service can interact so as to satisfy the client, we need to define *compliance.*

Definition 7 (Compliance). *A service ρ is* compliant *with a client σ, denoted $\rho \vdash \sigma$, iff $\sigma \parallel \rho$ has no deadlock.*

By *deadlock*, we mean a state in which the client is not satisfied and no transition can be fired. A service τ *refines* a service ρ iff every client satisfied by ρ is also satisfied by τ. Based on these definitions, a service $a+b$ does not refine a, because the client $\bar{a}.\,e + \bar{b}.\,\bar{c}.\,e$ would be satisfied by the latter but not by the former. To avoid this, a looser notion of compliance, called *weak compliance*, can be used (see [2]). A service ρ is *weakly compliant* with a client σ if there exists an orchestrator which may interfere with the execution of ρ and σ as to satisfy the client, by storing outputs until a corresponding input is enabled, and, in the situation where both client and service offer an external choice, not choosing any interaction that may lead to deadlock.

We are interested in building light-weight memoryless orchestrators which can be expressed by a priority preorder over the set of interactions.

More precisely, given a service σ and a client ρ, we proceed in two steps.

Step 1: We infer concurrency. While building the synchronized product $\sigma \parallel \rho$, violations of the compliance relation are detected. If they can be avoided by reordering of messages that can be implemented using bounded buffers, then they are collapsed in the specification. Otherwise the transitions leading to deadlock are marked as *error*. Transitions that have been (in the view of the designer) erroneously collapsed can be marked as *error* as well.

Step 2: We infer priorities. If possible, a set of priorities is computed so as to make the erroneous states unreachable according to the operational semantics given in Section 2, by inhibiting some transitions. Static priorities are a nice trade-off between the need for some non-local dependencies and the risk to build a specification that would require a global state.

3.2 Inferring concurrency

Checking compliance between a client σ and a server ρ means checking of deadlock freedom of $\sigma \parallel \rho$. Let us consider the following client σ and service ρ.

$$\text{Client} : \bar{a}.\,b.\,c.\,e \qquad\qquad \text{Service} : a.\,\bar{c}.\,\bar{b}$$

Clearly, σ and ρ are not compliant, which is a bit of a pity because, a priori, the service does not need to interact with the environment between the outputs $\bar{c}$ and $\bar{b}$, and the order in which these outputs are presented to the client has no influence on the service[2]. In order to make the service and client compatible, we cannot modify the service itself. In [2] it is proposed to systematically add buffers between the component and the environment to allow the environment to pick interactions which can be made available in the desired order. In presence of choices, however, adding possible behaviors may also add new deadlocks occurring a few steps later. These deadlocks must then be eliminated by adding priorities. Therefore, we only add order inverting buffers when this allows eliminating a deadlock.

Definition 8 (Reordering of concurrent transitions). *Two transitions $t_1 = (q_1, a, q_1')$ and $t_2 = (q_2, b, q_2')$ of an LTS K are* concurrent *if $q_1' = q_2$ and a is an output, and furthermore there exists no external choice in conflict with a.*

[2] At least this is the case if the abstraction is precise enough to explicit all communications including an input.

When a is an output, it is possible to withhold the actual interaction a until after the execution of the interaction b because the behavior of K does not depend on the effect of K. If the interaction a is accepted by the environment after the interaction b but not before b, this allows avoiding a deadlock. If there exists an alternative to the interaction a, the future of K in q_1 *does* depend on the effect of a: speculating on b before a is indeed accepted by the environment may lead to a state outside the specification if an alternative for a ends up to be chosen. This is the reason why we propose such reorderings only if q_1 is a deadlock state, and no alternative is available.

Definition 9 (Trace sets, matching pairs). *As usually, we define the set of traces as the set obtained by reordering of concurrent interactions, but only from the left to the right. When $s_1.t_1.t_2.s_2 \in traces(s)$ and $t_1.t_2$ is concurrent, then $s_1.t_2.t_1.s_2 \in traces(s)$*

Two sequences s_1 and s_2 can be matched *if there exists $s'_i \in traces(s_i)$ such that s'_2 is obtained from s'_1 by inverting inputs and outputs.*

Instead of presenting the full algorithm, we illustrate it on the main situations to be handled. Let us first consider the simple Service above: while building the synchronized product of Client and Service, deadlocks are detected, here $b.\,c.\,e \parallel \bar{c}.\,\bar{b}$.

In this state, the Client can only interact on b, whereas the Service only on $\bar{c}$; the Service is deterministic and the smallest sequence of deterministic interactions containing both c and b — the only offer of the Client — is $s_1 = \bar{c}.\,\bar{b}$. Symmetrically on the Client side, we find $s_2 = b.\,c$. We find $traces(s_1) = \{\bar{c}.\,\bar{b}, \bar{b}.\,\bar{c}\}$ and $traces(s_2) = \{b.\,c\}$ and a matching pair $(\bar{b}.\,\bar{c}, b.\,c)$. A buffer realizing the required reordering in the Client, eliminates the deadlock.

In the case where s_1 and s_2 do not contain the same set of interactions, then if they can still be made matching by adding continuation, we can redo the same step until we know that the sequences can never match or we find a matching pair. In practice, it is not very reasonable to consider long sequences which in fact have to be remembered as such and are more a "patch" than a general solution. Also, if either the client or the server is distributed, they ressemble very much a memoryful orchestration, which we wanted to avoid. This is also the reason why we do not handle loops here, even if in principle this is doable.

If in the deadlock state several interactions are offered by the client or the service, any pair of interactions may be checked for a possible matching sequence pair, and we will retain only one of them.

If no multiset as described exists, or if the designer thinks that the concurrency inferred is erroneous, the corresponding states are marked as deadlocks. Otherwise, the size of buffer required for reordering is bounded by the largest number of actions collapsed on a single transition in the product.

3.3 Finding a sufficient priority order

Besides reordering, the orchestrator can also restrict external choice so as to avoid deadlocks. Indeed, if both client and service offer two actions as an external choice, then the orchestrator may choose which one is taken. Instead of presenting the full algorithm, we show how it works in several cases.

A simple example that works with priorities. An orchestrator can be represented with static priorities in the following example:

$$\text{Client} : \bar{a}.\,c.\,e + \bar{b}.\,d.\,e \qquad\qquad \text{Service} : a.\,\bar{d} + b.\,\bar{d}$$

The priority $a < b$ is sufficient to make the deadlock unreachable.

A simple example that does not work with priorities. Here is an example where an orchestrator cannot be represented with static priorities.

$$\text{Client} : \bar{a}.\,c.\,e + \bar{b}.\,\bar{b}.\,c.\,e + \bar{b}.\,\bar{a}.\,d.\,e \qquad\qquad \text{Service} : a.\,\bar{d} + b.\,b.\,\bar{d} + b.\,a.\,\bar{d}$$

Priorities cannot account for that, because avoiding deadlock both after a and after $b.\,b$ would require at the same time $a < b$ and $b < a$, which is impossible.

A short look at these examples makes it clear that weak compliance enforced via static priorities will not be preserved by refinement unless the priorities have a semantic meaning – which is not the case in our first trivial examples. Here is a detailed illustration of how the algorithm proceeds on the second example.

1. The product of client and service is computed. Deadlocks are detected and transitions leading to them are marked as *error*, as shown in figure 1.

2. State 1 of $\sigma \parallel \rho$ is the initial state and in this state b must be preferred to a in order to prevent a deadlock, so we conclude that any possible set of priorities making σ and ρ compliant includes $a < b$.
3. Besides, there are two possibilities to avoid the deadlock occurring if b is fired in state 2: either b has lower priority than a or state 2 is not reachable. The first option would lead to a contradiction, namely, $a < b$ and $b < a$ at the same time. The second option would imply that the transition from 1 to 2 labeled by b should be inhibited by a transition with higher priority enabled in 1. Such a transition can only be $1 \xrightarrow{a} 2$, which is also impossible. Thus we conclude that no memoryless orchestrator can make the given client and service compliant.

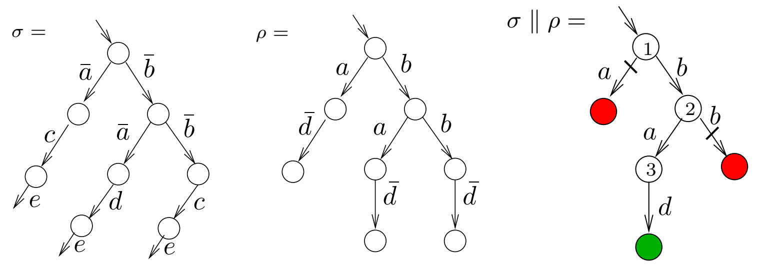

Fig. 1. A client σ, a service ρ and their product $\sigma \parallel \rho$.

The general algorithm is slightly more complicated because for each transition *error* one has to consider either adding a priority or removing a state.

If the algorithm fails to produce a priority order making the deadlocks unreachable, other approaches are possible. One is to infer more concurrency. We have inferred concurrency only in deadlock states. Some sequences of interactions that have been rejected might be made possible by adding priorities and thus satisfy the client.

It is important to underline the fact that the product of the client and the service is expected to be of a size comparable with both. As a matter of fact, it is even expected to be smaller because composition is likely to add constraints to both client and service. If there are several clients or services to be considered, then compositional approaches are possible. This will be discussed in section 5.

The dining philosophers example. Let us consider the dining philosophers example. The variant presented here is inspired from [2]. Philosophers are services who provide thought if they are given two forks by the resource. We consider here two philosophers and a resource with two forks. As usual, the deadlock arises if both philosophers are allowed to get one fork and never return them.

To handle this problem, always giving the highest priority to the request that is the closest to completion is a classic method for managing resources. Memoryless orchestrators are powerful enough to enforce such a policy. In the context of the dining philosophers, we thus give a different label for the action of getting the first or the second fork. The priority order then inferred $\{fork_1^\alpha < fork_2^\beta,\ fork_1^\beta < fork_2^\alpha\}$ is sufficient to prevent the deadlocks. For readability reasons, in figure 2, instead of interactions, we use the names of the ports that distinguish them and we give the same name $fork$ to 4 different ports of $Forks$ as they cannot be enabled at the same time.

Forks $= \texttt{rec}\, x.\overline{fork}.\overline{fork}.thought.return.return.x$
Philo $= \texttt{rec}\, x.fork_1.fork_2.\overline{thought}.\overline{return}.\overline{return}.x$

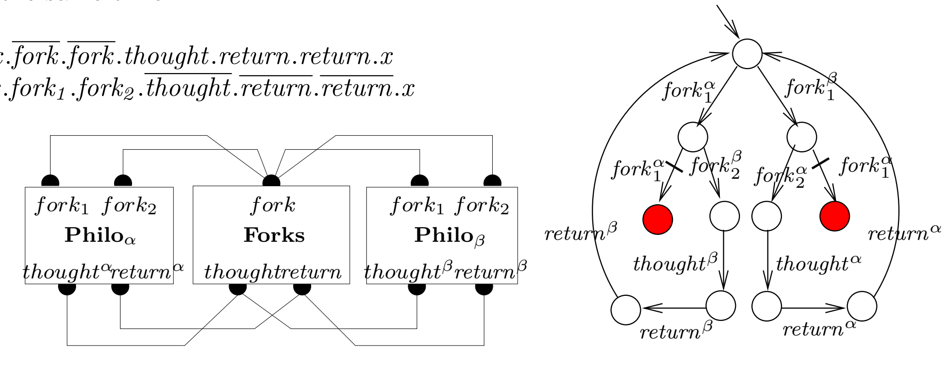

Fig. 2. A solution to the dining philosophers problem.

4 Implementation of memoryless orchestrators

BIP is defined by global semantics useful for ensuring progress properties with concise component specifications and proving them correct structurally [6]. Thus it is implemented by a global simulation engine [5] interacting with components.

In the context of web-services, we need a distributed engine. A partially concurrent implementation of BIP with message passing is given in [13], but still based on the existence of a centralized engine. Other proposals consist in adding a component for handling individual connectors, but this is not sufficient in presence of non prioritized choice or global priorities.

We propose a truly distributed BIP implementation by means of protocol agents associated with each component and which negotiate the next interaction to be executed as locally as possible. We extend existing algorithms for distributing process algebras such as [14] by handling global priorities. The protocol agents are machines communicating by exchanging messages via non lossy channels preserving the order of messages with the same source and destination; protocol read messages from their local message buffer by also preserving the order for any given message source.

Another specificity is that we rely on the BIP semantics of the system to satisfy properties, including deadlock freedom and fairness, required to guarantee client satisfaction. And we also want to consider distributed choice amongst a set of conflicting interactions without using static priorities. Besides, we assume components' internal activities to be terminating.

To define a correctness criterion for our algorithm, we provide two properties characterizing LTS obtained by composition using a BIP composition operator.

Notation 1 (Ready/enabled/conflicting/independent interaction) *An interaction c is* ready *in a (global) state q iff each port in c is locally ready. c is* enabled *in q iff c is ready in q and no interaction with higher priority is ready in q. Readiness, unlike enabledness, does not take priorities into account.*

As usual, two interactions are called conflicting *in state q if both are enabled but only one of them can be chosen in q. If both can be chosen jointly (according to the BIP multi-shot semantics) they are called* independent*.*

Property 1. Suppose a BIP system S defined as the composition of a set of components $\{K_i\}_{i=1}^{n}$ with a composition operator $(\mathcal{C}, <)$.

Safety. If in a state $q = (q_1, \dots, q_n)$ of S two interactions without a common component are *enabled*, then they must be independent. That is, all the states obtained by firing any of a, b or $a|b$ are reachable, and moreover, after the execution of $a|b$, or a and b in any order the same state is reached.

Progress. Any execution of S inevitably leaves q by executing at least one of the successors of S, if S has at least one successor in q. Any execution of S starting in q inevitably leaves q_i if there is no loop nor deadlock state containing q_i in S[3].

4.1 The distributed algorithm

For the sake of readability, we present an algorithm handling only binary rendez-vous connectors . We use c to denote this unique interaction as well as, when no mistake is possible, the corresponding local (input and output) ports. We do not consider true multi-shot semantics where a component is allowed to *require* interactions to occur jointly. See section 5 for a discussion about this issue.

Notation 2 *We call $ready_K(q_i)$ the set of* ports *enabled locally in a component K in state q_i and refer to it as the ready-set of K in state q_i.*

We denote by $k(c)$ the two components involved in a connector c. Given K and one of its connectors c, we write K_c for K's peer for c.

We denote by $dp(c) = k(\{b \mid c < b\})$ the set of components having an interaction with higher priority than c.

Our algorithm uses for deciding enabledness of c direct "negotiations" amongst components in $k(c)$ – which determine readiness – and then between one component in $k(c)$ (a "priority negotiator" chosen statically), and one component for each $a \in dp(c)$ (also a statically chosen negotiator). The algorithm is implemented by a protocol amongst protocol agents Pr_i. Each Pr_i is associated with a corresponding component K_i, which itself is obtained by enriching the original component by the set of buffers computed in the algorithm of section 3.2. The algorithm consists of the following main phases:

[3] we consider finite state LTS S.

1. a *busy* phase, in which an interaction to be executed has been chosen and the communication part is terminated. K_i is now executing the action part and the local ready-set in the successor state is not known yet. In this phase the protocol agent is quiet and defers all enquiries of other protocol agents by letting messages accumulate in its buffer[4]. Whenever the *busy* phase is entered or left all pending enquiries are handled. We suppose that no agent remains indefinitely *busy*.
2. a *ready* phase, in which the current ready-set $readySet$ has been communicated by K_i. Pr_i remains in this state until an enabled interaction has been found.
   To find an enabled interaction amongst $readySet$, the protocol checks $readySet$ in decreasing order of priority $<$. Determining if an interaction c is enabled by communication with its peer Pr_c and with $dp(c)$ is relatively straightforward (see algorithms in the annex).
   As soon as a c is found enabled, or if $readySet$ contains only one potential successor, Pr_i (tries to) execute c and sends a $COMMIT$ message to its peer[5].
   The interaction c is locally "done", if a $COMMIT$ has been sent to and received from Pr_i's peer,in any order. For simplicity, we suppose data to be piggybacked on the $COMMIT$ message[6].
3. whenever Pr_i is (from its own point of view) the first one to choose interaction a (and to send $COMMIT$), then it enters the *commit* phase, in which it waits for its decision to be committed — in which case the interaction is done and P_i provides the choice and the data to K_i and becomes *busy*. Or it is rejected, by an explicit $REFUSE$ which brings Pr_i back to *ready*. Any "counter proposal" that is received at this stage is $REFUSE$d, unless static analysis has detected the existence of a potential "decision cycle" in which case, there will be one statically determined *cycle breaker* who gets his choice done, and if Pr_i receives a counter proposal a from the *cycle breaker* — which may depend on the pair a, c — it will wait its own c to be accepted (no actual cycle exists) or refused (a cycle may exist), and only then send a response for a. Once, c or a is commited from both sides, all interactions for which a readyness request has been made or responded to, are explicitly $REFUSE$d.

Note that, when all protocols Pr_i are *ready*, the global state should be a state of S. Such a situation may never occur globally, and this is not needed. Whenever an interaction a is chosen, it is sufficient that all components which determine the enabledness of a in q — that is also those which may compromise a through a priority rule — are *ready* and in a state q compatible with S; the state of all other components is irrelevant.

5 Discussion

Memoryless orchestrators are not as powerful as, e.g., the *simple* orchestrators introduced in [2] in the sense that there are fewer pairs client/service that they make compliant. Besides, the complexity of our algorithm is even larger than the one for generating orchestrators. This is the price for having orchestrators from which we can automatically derive efficient distributed algorithms. Because concerns are clearly separated, our implementation does not have to take care of properties such as fairness. Obviously, the methodology presented here does not preserve compliance by refinement. However, using a notion of refinement under context as in [7] that takes into account priorities would allow reusability to some extent.

When designing Web services, it is possible and helpful to abstract away some implementation details. We propose to go a step further in this direction by using structured interaction, as is the case in BIP. The BIP framework described in Section 2 is much more powerful than the subset that has been used here. While using the priority layer we have kept interactions binary. Connectors can represent not only binary rendez-vous, but also n-ary rendez-vous or broadcast.

Instead of specifying group protocols in terms of sequences of binary interactions, we can express them at the specification level using, possibly a single, n-ary connectors, thus increasing the level of abstraction of the specifications. It is possible in BIP to represent loss of messages occurring when a component (process) receives a message that it cannot handle at the time of the reception. This is done by building a connector with two legal interactions, namely the output alone and the rendez-vous. Structural verification of BIP systems ([6]) could handle this efficiently, as well as multiple clients and services. In particular, the question of *realizability* (see [3]) can be decided very simply in BIP.

Designing Web services in a framework as expressive as BIP reinforces the distinction between design and implementation, allowing for a more abstract (and thus easier) design phase, more efficient verification,

[4] the size of the buffer can be statically bounded by the number of ports of K_i

[5] we aim at maximising progress, knowing that overall progress is enforced by the safety property of S.

[6] This is reasonable if the data volume is small or if conflicts are rare.

and automated implementation of the protocols realizing the specification. Expressing full BIP by protocols would require methods to decide when an orchestration component will be more efficient than protocols.

References

1. Bravetti, M., Zavattaro, G.: A theory for strong service compliance. In: Proc. of COORDINATION'07. Volume 4467 of LNCS. (2007) 96–112
2. Padovani, L.: Contract-directed synthesis of simple orchestrators. In: Proc. of CONCUR'08. Volume 5201 of LNCS. (2008) 131–146
3. Salaün, G., Bultan, T.: Realizability of choreographies using process algebra encodings. In: Proc. of IFM'09. Volume 5423 of LNCS. (2009) 167–182
4. Gößler, G., Sifakis, J.: Composition for component-based modeling. Sci. Comput. Program. **55**(1-3) (2005) 161–183
5. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Proc. of SEFM'06, IEEE Computer Society (2006) 3–12
6. Bensalem, S., Bozga, M., Sifakis, J., Nguyen, T.H.: Compositional verification for component-based systems and application. In: Proc. of ATVA'08. Volume 5311 of LNCS. (2008) 64–79
7. Quinton, S., Graf, S.: Contract-based verification of hierarchical systems of components. In: Proc. of SEFM'08, IEEE Computer Society (2008)
8. Bliudze, S., Sifakis, J.: The algebra of connectors: structuring interaction in BIP. In: Proc. of EMSOFT'07, ACM Press (2007) 11–20
9. Milner, R.: A calculus of communication systems. In: LNCS 92. Springer (1980)
10. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall (1984)
11. Bliudze, S., Sifakis, J.: A notion of glue expressiveness for component-based systems. In: Proc. of CONCUR'08. Volume 5201 of LNCS. (2008) 508–522
12. Milner, R.: Calculi for synchrony and asynchrony. Theor. Comput. Sci. **25** (1983) 267–310
13. Basu, A., Bidinger, P., Bozga, M., Sifakis, J.: Distributed semantics and implementation for systems with interaction and priority. In: Proceedings of FORTE'08. LNCS, Springer (June 2008)
14. Bagrodia, R.: Synchronization of asynchronous processes in CSP. ACM Trans. Program. Lang. Syst. **11**(4) (1989) 585–597