Verifying correctness of Web services choreography

Melliti Tarek
IBISC
University of Evry, France
Tarek.Melliti@ibisc.fr

Celine Boutrous-Saab
LAMSADE
University of Paris-Dauphine, France
Boutrous@dauphine.fr

Sylvain Rampacek
CReSTIC
University of Reims, France
sylvain.rampacek@univ-reims.fr

Abstract

This paper is about Web services used in distributed, inter-organizational business cooperation (choreography). In this application scenario, we have a multipart functional convention between all the involved Web services (called partners) in order to reach the purpose of the choreography. In such a scenario, two main problems can occur: i) Can we determine whether the resulted composition of partner is conforming or not to the expected behavior, with respect to the initial cooperation schema? ii) Can we determine whether the cooperation is possible by considering the individual partners' behaviors? i.e. are the different partners' behaviors compatible between them? In this paper, we address the second problem. We propose a method to model partners' behaviors and to check the correctness of the choreography (deadlock-free) based on the properties of one partner's behavior.

1 Introduction

Web services signal a new area of lightweight distributed application development. They show a strong orientation of cross-platform, cross-language compatibility due to their framework interoperability. This makes them a robust framework for services oriented computation on the Internet. One of the design goals for Web services is to allow companies and developers to share services with other companies in a simple way over the Internet. The service description language WSDL plays a central role in Web services interoperability. It describes the service interface by listing the collection of operations that are network accessible through standard XML messaging[13].

The WSDL model is restricted to only very simple computation services (it presents services as a set of independent operations). However, certain types of applications require combining today's simple Web services into more complex ones in order to achieve more sophisticated application purposes (e.g. B2B). To this aim, two extensions of Web services' technologies are currently investigated: the *orchestration* and the *choreography*. The first aims to combine existent Web services by adding a central coordinator (orchestrator) which is responsible of invoking basic Web services, according to a set of control flow patterns. The second is referred to as a Web service choreography, which does not assume the existence of a central coordinator but defines a conversation that should be explicitly considered by each participant Web service (called partner). In most cases, choreography and orchestration are used complementarily[4]. Orchestration languages are used to implement a choreography. In this paper, we consider the case of a distributed implementation of a choreography using the orchestration language BPEL4WS[1], BPEL for short.

Technically, BPEL allows each partner to orchestrate its behavior in order to fulfill its role, with respect to the choreography specification. It provides a set of communication primitives such that the "invoke" (for requesting services from partners), "receive" and "reply" (for offering services to partners). This model of primitives defines the interaction protocol viewed by each partner. It also provides the data manipulation facilities necessary to describe the computation performed by an executable process. All the previous primitives, communication and data handling, are structured using a set of control flow constructors, to describe the order of their execution.

Goals

This paper is about the use of Web services in distributed, inter-organizational business cooperation (choreography). When such a cooperation is established between a set of partners, the formers begin by defining the choreography. Then, each partner defines its own role as a BPEL service (orchestrator) and shares an abstract description of its behavior (by hiding computation details) with the other partners. This description, called abstract BPEL, describes the partner's behavior and interaction points with the others. In such an application scenario, two main problems can occur: conformance and properties checking.

The first problem is to verify whether the interaction between the orchestrated individual services is conform to the conversation specified in the choreography specification. *Van der Aalst et al* addressed this problem in [21] and a similar approach is adapted by *Baldoni et al* in [2]. In the present paper, we address the second problem. We consider a set of partners, given their observable behavior description (their abstract BPEL description), we propose a method to verify the compatibility between them, from the interaction point of view (the interaction is deadlock-free).

Motivating scenario

In order to motivate our work, let us consider the following cross-organization cooperation. The example involves three partners: a Customer agent (flight agency) C, an Airline company (flight-tickets seller) A and a Bank center B. The three companies aim to offer a secure tickets buying service for the bank clients. C plays the role of an intermediate between end-users' needs and the available airline flights. Once the user choice done, C asks B to pay A. According to the success or the fail of the payment operation between B and A, the latter sends back either a confirmation or a fail message to C, who in turn forwards the response to the end-user. If we suppose that each partner provides an abstract description of its observable behavior which means its input and output and time constraints, we verify that the composition of these three services is deadlock-free (they are compatible). This example will be used to illustrate each step of our work.

Approach

After reviewing the related work, section 3 gives an informal introduction to abstract BPEL features. Then, we point out the lack of semantics (the specification semantics is written in English proses), in section 4 we give an operational semantic to BPEL process definition language and argue our choice on abstraction consideration and time model, we end this section by proposing a formal definition of a partner and a choreography. Using the given formal semantics, we model the observable behavior of each partner using a TIOTS (Timed Input-output Transition System). While a partner exposes only its external behavior, the resulted TIOTS may be non-deterministic. This non-determinism can cause deadlock during the interaction with such a partner. In section 5, we propose a conformity relation that characterizes the class of correct Web services behaviors. Based on that relation, we propose an algorithm that checks the compatibility between two partner's behavior. In section 6, we reduce the problem of choreography correctness checking to a two partners compatibility problem by introducing an aggregation operator. We conclude the paper by summarizing our main contributions.

2 Related works

Using either a choreography or an orchestration approach, the resulted service is always called a composite Web service. The composite service adds two dimensions by comparison to the simple ones; they are statefull and they obey to an operational behavior (interaction protocol). This raises many theoretical and practical issues which are part of ongoing research [15]. Due to the lack a formal semantic to BPEL (its semantic is defined using English prose), it is hard to define formal tools and methods that can validate and verify behavioral properties by acting directly on BPEL expressions. The principal approach, followed by most of the state-of-the-art works, is to translate a service behavior (BPEL process) into a mathematically well-founded model, considering only the semantic of elements that are relevant for the property to be verified. Then, model-checking methods can be applied to the formal representation of the composite service behavior. There are three major formalisms which were successfully applied: Finite State Machines (FSM), Process Algebras (PA) and Petri Nets (PN). A great number of works therein aims to verify specific properties of a BPEL process.

In [20], the authors translate a given BPEL process into a process algebraic expression in order to verify its control flow. Based on this work, they provide an in-depth-analysis of BPELs Dead-Path-Elimination by formal means. In [7], a similar approach is given and is translated to LOTOS. In previous works, we gave an operational semantic to Xlang (ancestor of BPEL) in order to verify the ambiguity (non usability) of a Web service behavior (not deadlock-free)[12, 9]. In [17], a similar work is done using Petri Net. In [19, 6, 10] and [18], a pattern-based translation of activities into Petri Nets are proposed, then they use Petri Net properties to make properties checking.

Most of these works try to verify properties related to a single BPEL process (the coordinator). In this paper, we focus on choreography correctness based on a single partner's observable behaviors (see [5] for work dealing with choreography). Using the same approach as previous related works, we begin by giving an observable operational semantic to abstract BPEL. The main difference with the previous works is that we do not map BPEL process constructors to Process Algebra ones, but we consider them as a grammar of timed process algebra and we define their operational semantic according to their informal definition. This step allows us to model the partners' behavior using TIOTS (by applying operational rules) with regard to their interaction. The correctness of the choreography is then characterized using relational properties between each participant's behavior and its partners' ones.

3 Web service Choreography Using BPEL

In the following, and according to the sketch in the introduction, a choreography is defined by a set of partners' abstract BPEL files. Each partner implements its service using the executable BPEL file and shares an abstract BPEL file with its partners. In this work[1], we consider the maximum abstraction level by hiding all the computational details (variables, assigning expression, boolean condition, internal exceptions raise). This seems to be more realistic due to confidentiality of business rules between partners (see [12][9]). In the remainder of this paper, we will only consider the abstract facette of BPEL.

3.1 Informal introduction to abstract BPEL specification

An abstract BPEL file is composed of a set of partners' interface declaration (the required interfaces), a set of local WSDL operations (the provided interface) and the service behavior definition (defined using the BPEL process definition language, see below).

3.1.1 WSDL Operations

An operation is defined by the exchanged messages needed to invoke it and the order in which they have to be performed. Web service's framework (WSDL) supports four types of operation; *notification*, *solicit*, *request-response* and *solicit-response*. The operation, local to the service or remote (partner's one), constitutes the basic communication imperative. The interaction between two services is fulfilled by invoking their mutual operations.

Let M be a set of XML message types. We use symbol "?" for input and "!" for output (e.g. $?m$ receiving a message of type m). $opNames$ a set of operation names. We denote the set of operations by O (we range over using o_i). A WSDL operation $o \in O$ can be either one-way $o = opName[*m]$ or two-way $o = opName[!m_1, ?m_2] | opName[?m_1, !m_2]$ ($*$ may be substituted by "!" or "?"). We use Ω to denote the set of partner names, we range over using ω_i. To precise the owner of an operation or a message we prefix it by the name of the owner partner (e.g $\omega.o$ is the operation o of the partner ω). We also introduce the following functions:

- $Input : O \rightarrow M$ with $Input(o) = m$ where $o = opName[?m]$, $o = opName[?m, !m']$ or $o = opName[!m', ?m]$)
- $Output : O \rightarrow M$ with $Output(o) = m$ where $o = opName[!m]$, $o = opName[?m', !m]$ or $o = opName[!m, ?m']$

The $Input$ function returns the input message of an operation if it exists (not defined for *notification*). The $Output$ function returns the output message of an operation (not defined for *solicitation*).

3.1.2 Abstract BPEL process definition language

It allows partners to define their behavior and their interaction (with their partner) in the whole choreography. The Abstract BPEL process definition language (AbBPEL) is composed of a set of activities definitions. In AbBPEL we distinguishes two types of activity, basic ones and structured ones. In the following, we present the grammar of the language using a personal syntax notation of the BPEL activities in order to avoid XML sugar notations[2].

Basic activities They are the basic element used to compose a partner's behavior.

In the following, we use x_i to design partner name variable and $\overline{X_i} = (x_1 ... x_n)$ a vector of the partners name variables.

$$BasicAct ::= \\ receive[o](x) | reply[o](x) | invoke[o] | wait[d|t] | throw[e]$$

- $receive[o](x)$: it is a communication imperative activity. It enables the process to receive an invocation order for its operation o from one of its partners with name x[3]. The use of this variable x will be explained the section 3.1.4.

[1]In the current version of BPEL specification, the abstract BPEL is similar to the executable one. The specification leaves to the developer the choice of the abstraction level.

[2]We recall that the abstract facette of the process language is a variant of the process execution language by hiding some implementation details. We choose to consider an abstraction that hides the branching conditions, the iteration condition, the local exception rasing and internal data manipulation (i.e assigning).

[3]In BPEL, the $receive$ activity is not an addressed activity, we introduce the variable x in order to specify the choreography.

- $reply[o](x)$: it is a communication imperative to send back (to partner or client) the response to partner x for the invocation of operation o . The $receive$ and $reply$ activities must be applied to a local operation.

- $invoke[\omega.o]$: it is a communication imperative to invoke the operation o of the partner ω (ω must be different from the name of the current partner). Two-ways operations (sending request and receiving response) are synchronous one, the invoker send request and wait for response.

- $wait[d|t]$: it is an event oriented activity which produces a timeout event for a given period t or a date d. Here, we consider just the period $wait(t)$. The event is catched in the process itself (no timeout communication).

- $throw[e]$: it does throw an exception e. An exception is an XML type, so we can consider that $e \in M$. The difference is that they can be internal events and not communication events.

- $empty$: it is an activity which does nothing, and then terminating.

- $time$: this activity is not present is the BPEL (we will explain below). It represents an activity that does nothing but lets time pass.

Within an AbBPEL, each $invoke[\omega.o](x)$ is called an invoke instance of the operation o or partner ω and each couple of $receive[o](x)$ and $receive[o](y)$ where $y = x$ an instance of o call.

Structured activities Structured activities are a set of control flow constructors. Each structured activity defines a specific order in which activities (in parameters) will be executed (or activated). They can be applied on structured or basic ones. We use $P, Q, ...$ as symbols to range over activities (basic or structured ones).

$$StructAct ::= \\ P; Q|\text{flow}[\{P_i\}_{i\in I}]|\text{while}(P), \text{switch}[\{P_i\}_{i\in I}]|\text{scope}(P, E) \\ \text{with } E = [\{(m_i, P_i) \mid i \in I\}, (d, Q)]$$

Note here that P and Q can be parameterized by the partner variable if they contain $receive$, $reply$ or $invoke$ activities.

- $(P; Q)$: $sequence$ activity represents the execution sequence, P then Q.

- $\text{flow}[\{P_i\}_{i\in I}\}]$: executing different activities in parallel (P_i). The joint links are ignored for simplicity.

- $\text{switch}[\{P_i\}_{i\in I}\}]$: conditional execution control. According to some condition, the process activates some sub-activities (P_i).

- $\text{scope}(P, E)$: encapsulates P and guards it by a set of events and their associated activities, E[4]. The process behaves like the P activity if none of the events happens. The events can be a received message, a raised exception, or a time event (the wait). For each event, the associated activities are executed when it happens.

- $\text{while}(P)$: repeats the execution of the P activity until a condition is hold or broken.

Note that, in our syntax, we limit to the observable parameter e.g. for the $switch$ and $while$ activities, we do not represent their conditions. From an external point of view, the $switch$ is assimilated to an internal non-deterministic choice operator. The chosen behavior is done internally (out of the control of partners or clients). The same reasoning is applied for the $while$. Also, we would like to signal that we do not handle explicitly the $pick$ constructor because we consider that it is a special case of the $scope$. The $pick$ is a $scope$ that does nothing but lets time pass (that is why we introduce $time$ as a basic activity) and it is guarded with all the $pick$ events.

The process definition language of BPEL is defined as follow:

$$BPELProcess ::= StructAct$$

That means that each process behavior is represented by an imbrication of structured and basic activities. We use $BPELProcess$ to represent the BPEL process language expressions and we range over using B_i.

3.1.3 Partner definition

Each partner involved in the choreography provides an abstract BPEL file describing : (i) its partner, (ii) its interface (local operations) and (iii) the AbBPEL process defining its behavior (here its observable behavior).

Definition 1 *An abstract BPEL file of a partner ω is a tuple $\langle P, In, B(\overline{X})\rangle$ with*

- *$\omega \in \Omega$ is the name of the partner (we use the name here with substitution to the notion of XML spacenames) and let Ω the set of all the partners names.*

- *$In \subset O$ represents the WSDL interface of the process. The set of operations offered by the service. $In = \{\omega.o_i\}$ for $i = 0..n$*

[4] E is an event handler process $E \overset{def}{=} [\{(m_i, P_i) \mid i \in I\}, (d, Q), \{(e_j, R_j) \mid j \in J\}]$, it associates to each event type (message, exception and timeout) a process

- *$P \subset O$ is the set of partners' operations $P = \{\omega_i.o | \omega_i \neq \omega$ and $\exists$ a partner $\omega_i\langle P_i, In_i, B_i\rangle$ with $\omega_i.o \in In_i\}$. It represents the set of imported partners' operations prefixed by the partner identifier (name).*
- *$B(\overline{X}) \in BPELProcess$ represents an expression of the BPEL process definition language with a vector of free partners name variables $\overline{X} = (x_1....x_n)$. It describes the partner behavior.*

3.1.4 Choreography definition

As sketched in the introduction, the choreography is a global overview on the inter-connection between partner's communication imperatives. In the BPEL abstract, the invoke activities are addressed (the owner of the operation) but the exposed operations ($receive$ and $reply$ activities) are not addressed. Defining the choreography is resumed to associate to each $receive$ and/or $reply$ instance the intended partner that will invoke it according to the global composition schema.

Definition 2 *A choreography is defined by $C = \{[\omega_i = \langle P_i, In_i, B_i\rangle, \phi_i]_{i\in I}\}$ where*

1. *$\phi_i : X_i \rightarrow \Omega^n$ a function that instantiates the partners parameters.*
2. *B_i is the expression $B_i(\phi_i(\overline{X}))$ where each x_i is replaced by $\phi_i(x_i)$. No free variables are allowed in the B_i expression.*

The choreography specifies, for each exposed operation, the target partner that will invoke it (the connection between partner). In our model, this is done by a set of functions ϕ_i which instantiate the $receive$ and $reply$ variables.

Definition 3 *A choreography $C = \{(\omega_i = \langle P_i, In_i, B_i\rangle, \phi_i)\}$ is valid iff, for each two-ways operation $o \in In_i$, we have $\phi_i(x) = \phi_i(y)$ where x and y are the variables of one $receive[o](x)$ and $reply[o](y)$ for each o call instance.*

3.2 Illustrative example

In this section, we present the choreography of our example.

partner A: provides the following operations: $o1[?getF, !ListF]$ returns the list of available flights, $o_2[?getCho]$ gets the user choice, $o_3[?cancel]$ receives a cancel from the customer and $o_4[?payment]$ receives the payment confirmation from the bank.

$$A = (P_A = \{C.o_2\}, In_A = \{o_1, o_2, o_3, o_4\}, B_A(x, y)) \text{ with } B_A(x, y) = rec[o_1](x); rep[o_1](x); rec[o_2](x); scope[time, \{(rec[o_4](y), inv[C.o_2]), (rec[o_3](x), empty)\}]$$

partner C: $o_1[?nocredit]$ receives a "no credit" message for a payment request and, $o_2[?conf]$ receives the success of a payment confirmation from the airline company.

$$C = (P_C = \{A.o_1, A.o_2, A.o_3, B.o_1\}, In_C = \{o_1, o_2\}, B_C(x, y)) \text{ with } B_C(x, y) = inv[A.o_1]; inv[A.o_2]; inv[B.o_1]; scope[time, \{(rec[o_1](x), inv[A.o_3]), (rec[o_2](y), empty)\}]$$

partner B: $o_1[?orderPay]$ receives a payment order.

$$B = (P_B = \{C.o_1, A.o_4\}, In_B = \{o_1\}, B_B(x)) \text{ with } B_B(x) = rec[o_1](x); switch[inv[C.o_1], inv[A.o_4]]$$

The choreography is composed by the three partners: $Agency = \{(A, \phi_A = \{(x, C), (y, B)\}), (B, \phi_B = \{(x, C)\}), (C, \phi_C = \{(x, A), (y, B)\})\}$

4 Modelling BPEL abstract process behavior

BPEL process definition language is a set of activities describing, in a modular way, the observable behavior of the involved Web services. In fact, this approach is close to the process algebra paradigm illustrated for instance by CCS [14], CSP [11] and ACP [3]. The main objective of the process algebra approach is to cope with the complexity of the conception of parallel systems (in our case the choreography). Since time is an important issue in such systems, the process algebra model has been enlarged by introducing (discrete or dense) time passing. The discrete time models are usually defined by a special transition representing one unit time passing [16]. Thus, it appears that the syntactic features of AbBPEL make it a good candidate to be an algebra of timed processes. Our approach is to consider AbBPEL as a timed process algebra, then we:

1. associate with each activity a set of operational rules which assigns for each process a behavioral interpretation.
2. introduce a compatibility relation, in order to compare different partners' processes, and which expresses that two processes (here partner behaviors) are compatible (their interaction is deadlock-free).
3. develop algorithms which decide the compatibility of two partners processes.
4. propose an algorithm that checks the correctness of a choreography.

as the first step for the development of the operational semantic we give the elements necessary to this semantic.

A labelled transition system is an oriented graph where the nodes represent the possible states of the system (with an initial state) and the arcs represent the state transitions.

Each arc is labelled by the action whose occurrence has triggered this transition. Depending on the process algebra language, some labels have a special meaning. We will detail our alphabet later.

Definition 4 *A labelled transition system LTS is defined a tuple $LTS = (S, L, \rightarrow, s_0)$ where:*

- *S is a set of states with $s_0 \in S$ the initial state*
- *L is a finite set of labels*
- *$\rightarrow \subseteq S \times L \times S$ is the transition relation*

A LTS is the representation of the behaviour of a process. The states of the process are simply the current process after some part of an execution. To each operator *op*, one associates a set of transition rules which define the possible behaviour of a process whose outer constructor is *op*. Let us suppose that we want to define a rule $[op_x]$ for a generic process $P = op(P_1, P_2, \ldots)$. At first, we have a boolean expression over some potential transitions of selected components of P: $Bexp(\{P_{o(i)} \xrightarrow{\alpha_i} P'_{o(i)}\})$. This condition is enforced by a second condition on the occurring labels denoted $guard(\{\alpha_i\})$. If the two conditions are fulfilled then a state transition for P is possible where the label $Lexp(\{\alpha_i\})$ is an expression depending on the labels of sub processes transition and the new state is an expression $Nexp(P, \{P'_{o(i)}\})$ depending on the original process and the new sub processes. Below, a generic rule is presented with the usual style.

$$[op_x] : \frac{Bexp(\{P_{o(i)} \xrightarrow{\alpha_i} P'_{o(i)}\})}{P \xrightarrow{Lexp(\{\alpha_i\})} Nexp(P, \{P'_{o(i)}\})} \text{ where } \atop guard(\{\alpha_i\})$$

4.1 Operational Semantics of Abstract BPEL activities

While the operational semantics are defined with response to a set of events, we will give the list of considered actions according to the informal semantic of abstract BPEL's process definition language:

- There are two events associated with a message m: the emission denoted by $!m$ and the reception denoted by $?m$. We also denote $!M = \{!m \mid m \in M\}$ and $?M = \{?m \mid m \in M\}$.
- Since the service may evolve in an unobservable way (e.g. the evaluation of a condition), we introduce τ, the internal action.
- Since BPEL takes time into account, χ denotes one time unit passing. We have chosen to represent time passing by units because the time constraints of a Web service are generally "soft" and thus the discretization of time is a valid abstraction. The interested readers can refer to [8] for dense time semantic.
- The exception events set of BPEL4WS is denoted by E_x.
- In order to control that the client correctly detects the end of the service, we introduce $\surd$, the termination event. This action will also simplify the definition of the operational semantics.

4.1.1 Basic Activities

Basic processes are time, empty, $\mathrm{receive}$, reply and throw.

time : It can only let time pass (χ stands for any delay):

$$\mathrm{time} \xrightarrow{\chi} \mathrm{time}.$$

empty : It can only terminate: $\mathrm{empty} \xrightarrow{\surd} 0$.

$\mathrm{receive}[o](x)$ **and** $\mathrm{reply}[o](x)$**:** The $\mathrm{receive}[o](x)$ activity consists of receiving the input message of the operation o and then terminating. A $\mathrm{reply}[o](x)$ activity sends the output message of the operation o and then terminates. The messages are prefixed by the name of the partner x (the partner that is supposed to invoke this instance of o). The two activities can be delayed.

$$\mathrm{receive}[o](x) \xrightarrow{?x.Input(o)} \mathrm{empty}$$
$$\mathrm{receive}[o](x) \xrightarrow{\chi} \mathrm{receive}[o](x)$$
$$\mathrm{reply}[o](x) \xrightarrow{!x.Output(o)} \mathrm{empty}$$
$$\mathrm{reply}[o](x) \xrightarrow{\chi} \mathrm{reply}[o](x)$$

$\mathrm{invoke}[o]$**:** It is applied to an operation that belongs to one of the current process partners ($o \in P$). Let $\omega.o \in P$ and let ω' be the name of the current partner.

$$\mathrm{invoke}[\omega.o] \xrightarrow{\chi} \mathrm{invoke}[\omega.o]$$
$$\mathrm{invoke}[\omega.o] \xrightarrow{!\omega'.Input(\omega.o)} \mathrm{empty} \text{ where } o = opName[?m_1]$$
$$\mathrm{invoke}[\omega.o] \xrightarrow{?\omega'.Output(\omega.o)} \mathrm{empty} \text{ where } o = opName[!m_1]$$
$$\mathrm{invoke}[\omega.o] \xrightarrow{!\omega'.Input(\omega.o)} \mathrm{invoke}[\omega.o'] \text{ where } \omega.o' = opName[!m_2] \text{ and } \omega.o = opName[?m_1, !m_2]$$
$$\mathrm{invoke}[\omega.o] \xrightarrow{?\omega'.Output(\omega.o)} \mathrm{invoke}[\omega.o'] \text{ where } \omega.o' = opName[?m_2] \text{ and } \omega.o = opName[!m_1, ?m_2]$$

The semantic of the *invoke* activity depends on the operation type. For one-way operation, the semantic is quite simple; the process executes the event according to the operation signature and becomes empty. For two-ways operation, after executing the first event (input or output), the activity becomes another invoke where the parameter is a one-way operation that have the same name and the same owner. The events are prefixed by the name of the current partner followed by the name of the operation owner[5].

throw: It raises an exception e which must be handled in some way (see below the scope process): $\forall e \in E,\ \text{throw}[E] \xrightarrow{e} 0$

4.1.2 Structured activities

We review the sequence, switch, while, flow, scope and pick operators of BPEL4WS. For space considerations, we do not detail all the rules. They are similar to most process algebra ones. The reader can refer to our previous works [12][8] for details. In the sequel, we will complete the AbBPEL algebra operational semantic by defining the structured activities operational semantics. In the table 4.1.2 we give the operational semantic of BPEL structured activities.

By a modular application of the transition rules, we can associate to each term of the process definition language (defined by the BNF $BPELProcess$) a Labeled Transition System.

Definition 5 *The behavior of a partner $\omega = (P, In, B(X))$ is a Labeled Transition System $LTS_{B(X)} = \langle Act = L \cup \{\tau, \chi, \surd\}, T, s_0\rangle$:*

- *S Set of States. Each state is labeled by an expression in $BPELProcess$;*
- *$L \subset (?M \cup !M)$ the set of sending and receiving messages (prefixed by partner names). $L = \biguplus\{Input(o)\} \cup \{Output(o)\}$ for $o \in (P \cup In)$;*
- *$T \subseteq S \times (L \cup \{t, \chi, \surd\}) \times S$ a transition Relation; and*
- *$s_0 \in S$ initial state, $s_0 = B(X)$.*

Since the LTS gives a semantic for the BPEL process definition language, it must take into account the time (discrete) and the sending/receiving of messages. Such LTS is called a Timed Input/Output Transitions System (TIOTS). In the remainder of this paper, a partner involved in a choreography within the instantiation function ϕ_i is defined by $\omega_i = (P_i, In_i, TIOTS_{B_i})$ where $TIOTS_{B_i}$ is the $TIOTS$ obtained by applying rules on the expression $B(\phi_i(\overline{X}))$. The figure 1 represents the partners' TIOTS according to the choreography of the $Agency$ example.

$P;Q$	$\mathbf{R}_{seq_1}$: $\forall a \neq \surd,\ \dfrac{P \xrightarrow{a} P'}{P\ ;\ Q \xrightarrow{a} P'}$
	$\mathbf{R}_{seq_2}$: $\forall a,\ \dfrac{P \xrightarrow{\surd} \text{ and } Q \xrightarrow{a} Q'}{P\ ;\ Q \xrightarrow{a} Q'}$
$switch$ $[\{P_i\}_{i\in I}]$	$\mathbf{R}_{switch}$: $\forall i \in I,\ \text{switch}[\{P_i \mid i \in I\}] \xrightarrow{\tau} P_i$
while$[P]$	$\mathbf{R}(_{while_1})$: while$[P] \xrightarrow{\tau} P\ ;\ \text{while}[P]$
	$\mathbf{R}_{while_2}$: while$[P] \xrightarrow{\tau}$ empty
flow $[\{P_i\}_{i\in I}]$	$\mathbf{R}_{f_1}$: $\forall a \in E_x \bigcup\{\tau\},\ \dfrac{\exists\, j{\in}I,\ P_j \xrightarrow{a} P'}{\text{flow}[\{P_i \mid i{\in}I\}] \xrightarrow{a} \text{flow}[\{P_i \mid i{\in}I\backslash\{j\}\} \bigcup\{P'\}]}$
	$\mathbf{R}_{f_2}$: $\forall m \in M,\ \dfrac{\exists\, j{\in}I,\ P_j \xrightarrow{*m} P' \text{ and } \forall\, i{\neq}j,\ \forall\, a{\in}E_x \bigcup\{\tau\}, \text{ not } (P_i \xrightarrow{a})}{\text{flow}[\{P_i \mid i{\in}I\}] \xrightarrow{*m} \text{flow}[\{P_i \mid i{\in}I\backslash\{j\}\} \bigcup\{P'\}]}$
	$\mathbf{R}_{f_3}$: $\dfrac{\forall\, i{\in}I,\ P_i \xrightarrow{\surd} P'_i}{\text{flow}[\{P_i \mid i{\in}I\}] \xrightarrow{\surd} 0}$
	$\mathbf{R}_{f_4}$: $\dfrac{\exists J{\notin}\emptyset,\ J{\subseteq}I,\ \forall i{\in}J,\ P_i \xrightarrow{\chi} P'_i \text{ and } \forall i{\in}I\backslash J,\ P_i \xrightarrow{\surd}}{\text{flow}[\{P_i\}_{i\in I}] \xrightarrow{\chi} \text{flow}[\{P'_i\}_{i\in J} \bigcup\{P_i\}_{i\in I\backslash J}]}$
scope(P, E)	$\mathbf{R}(_{scop_1})$: $\dfrac{P \xrightarrow{\surd}}{\text{scope}(P,E) \xrightarrow{\surd} 0}$
	$\mathbf{R}_{scop_2}$: $\dfrac{P \xrightarrow{\chi} P'}{\text{scope}(P,E^1) \xrightarrow{\chi} Q}$
	$\mathbf{R}_{scop_3}$: $\forall i \in I,\ \dfrac{\forall\, a{\in}E_X \bigcup \{\tau,\surd\}, \text{ not}(P \xrightarrow{a})}{\text{scope}(P,E) \xrightarrow{?m_i} P_i}$
	$\mathbf{R}_{scop_4}$: $\forall j \in J,\ \dfrac{P \xrightarrow{e_j}}{\text{scope}(P,E) \xrightarrow{\tau} R_j}$
	$\mathbf{R}_{scop_5}$: $\forall e \notin E_J,\ \dfrac{P \xrightarrow{e}}{\text{scope}(P,E) \xrightarrow{e} 0}$

Table 1. Operational semantics of Abstract BPEL structured activities

5 The notion of Ambiguity of an abBPEL process

By hiding computational details and especially the boolean conditions and the state of the involved variables, the observable behavior of the partners' process becomes non-deterministic over the internal transition (in addition to non-determinism over communication actions due to the service design itself). While the services' behavior contain external control (incoming messages), the non-deterministic internal transition may lead to a deadlock. In fact, the partners can not decide about the real state of the service. In order to clarify this point, let us define another version of the airline service; After receiving the customer request, and according to the destination, the payment can be done either using "Creditcard" or "Cash". Operation list:

[5]Note that, if a Web service uses invoke on an operation of a partner and this operation starts with an input message then, the associated event is an output message and *vice versa*

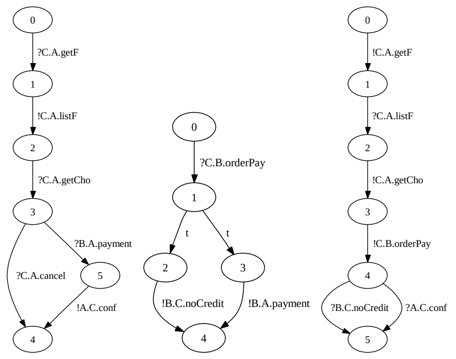

Figure 1. $TIOTS_{B_A}$**,** $TIOTS_{B_B}$ **and** $TIOTS_{B_C}$ **of partners according to the choreography of the** $Agency$ **example**

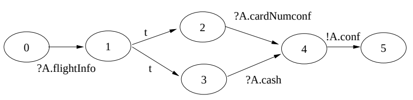

Figure 2. TIOTS of the Airline process: Ambiguous observable behaviour

- $A.o_1 = byticket[?flightInfor]$;
- $A.o_2 = cardPay[?cardNum, !conf]$;
- $A.o_3 = cashPay[?cash, !conf]$

The A partner abstract process is (see 2): $B_A = receive[o_1]$; $switch[\{(receive[o_2]$; $reply[o_2])$, $(receive[o_3]; reply[o_3])\}]$

The partner that will be in charge of the payment, the Bank in our example, has no way to know which payment mode will be expected by the airline. We call such a service "ambiguous". The service has no possible compatible partner. The interaction with such a service may lead to a deadlock. For a given level of abstraction (the hidden features), the designer of the process must adapt the services behaviors in order to avoid such deadlocks.

In the following, we propose a relation that characterizes the possible correct interaction (deadlock-free) between two partners' behaviors (TIOTS). Based on this relation, we define two algorithms, the first checks the ambiguity of partners' behavior and the second checks whether two TIOTS (two partners' processes) are "compatible".

5.1 An interaction relation for Web services

Let us briefly explain why the two main relations - the language equivalence and the bisimulation equivalence - do not match our needs. The language equivalence is unable to express the different branching capabilities of the systems (e.g. an immediate choice versus a delayed one) since it does not require an equivalence relation between the intermediate states of the two systems. The bisimulation equivalence does not take into account the different nature of the event: in an asynchronous communicating system, the sending of a message is an action whereas the reception is a reaction. Thus, the interaction relation that we introduce can be viewed as a bi-simulation relation modified in order to capture the nature of the events. As usual in the LTS formalism, we define an observable transition relation between states given by $s \overset{a}{\Rightarrow} s'$ iff $s \overset{\tau^* a \tau^*}{\longrightarrow} s'$ and $s \overset{\epsilon}{\Rightarrow} s'$ iff $s \overset{\tau^*}{\rightarrow} s'$. Moreover, we suppose that the exception events are not observable in the LTS of the service. If this is the case, it means that the service does not catch an exception and then must be modified.

We now derive the interaction relation from general considerations. Let us focus on some instants of the execution. If one LTS is able to send a message (action $!m$), the other one must be able to receive this message (action $?m$). If one LTS is able to let the time pass (action χ), the other one must also be able to let the time pass (action χ). At last, if one LTS is terminating (action $\surd$), the other one must also be able to terminate (action $\surd$).

The subtle point is about the reception of a message. Assume that one LTS expects the reception of $?m$, does it mean that the other one is able to send this message? The answer is not necessary yes since the latter LTS may evolve in an indistinguishable way from one state to two states, one where it is able to send m and the other one where it is not. However, we require that in the other state, it is able to send a message in order to avoid an infinite waiting of the first LTS.

We introduce the following notation $?m^c \;=!m$, $!m^c \;=?m$ $(?\omega.m^c \;=!\omega.m$, $!\omega.m^c \;=?\omega.m)$ and $\forall a \notin (!M \cup ?M); a^c = a$.

Definition 6 *Let $LTS_1 = (S_1, L_1, \rightarrow_1, s_{01})$ and $LTS_2 = (S_2, L_2, \rightarrow_2, s_{02})$ be two labeled transition systems. Then LTS_1 and LTS_2 correctly interact (noted $LTS_1 \asymp LTS_2$) iff $\exists \sim \subseteq S_1 \times S_2$ such that:*

- $s_{01} \sim s_{02}$
- $\forall s_1, s_2$ *such that* $s_1 \sim s_2$
  - *Let $a \notin \{?m\}_{m \in M}$, if $\exists s_1 \overset{a}{\Longrightarrow}_1 s_1'$ then $\exists s_2 \overset{a^c}{\Longrightarrow}_2 s_2'$ with $s_1' \sim s_2'$ and if $\exists s_2 \overset{a}{\Longrightarrow}_2 s_2'$ then $\exists s_1 \overset{a^c}{\Longrightarrow}_1 s_1'$ with $s_1' \sim s_2'$*
  - *Let $m \in M$, if $s_1 \overset{?m}{\Longrightarrow}_1 s_1'$ then*
    - $*$ $\exists s_2^- \overset{w}{\Longrightarrow}_2 s_2$, $\exists s_2^- \overset{w}{\Longrightarrow}_2 s_2^+$, $\exists s_2^+ \overset{!m}{\Longrightarrow}_2 s_2'$ *with $s_1 \sim s_2^+$ and $s_1' \sim s_2'$ where w is a word*

* $\exists s_2 \overset{!m'}{\Longrightarrow}_2 s'_2$

- *Let $m \in M$, if $s_2 \overset{?m}{\Longrightarrow}_2 s'_2$ then*
  * *$\exists s_1^- \overset{w}{\Longrightarrow}_1 s_1$, $\exists s_1^- \overset{w}{\Longrightarrow}_1 s_1^+$, $\exists s_1^+ \overset{!m}{\Longrightarrow}_1 s'_1$ with $s_1^+ \sim s_2$ and $s'_1 \sim s'_2$ where w is a word*
  * $\exists s_1 \overset{!m'}{\Longrightarrow}_1 s'_1$

The interaction relation when is hold between two systems insure two properties :

1. In spite of the internal non-determinism the interaction is dead-lock free.

2. The two processes detect their mutual termination.

Note that the interaction relation restrict the two properties to be hold for all the two systems functionalities (all the produced traces). In spite of the fact that this may be too restrictive in the context of Web services, this may have a sense since we are in an open world and we cannot predict the behavior of partners. We maintain that assumption for the choreography correctness checking.

Definition 7 *let $\omega_1 = (P_1, In_1, TIOTS_{B_1})$ and $\omega_2 = (P_2, In_2, TIOTS_{B_2})$ be two partners definitions. The two partners are compatible iff their behaviors are related with the interaction relation, $TIOTS_{B_1} \asymp TIOTS_{B_1}$.*

Two partners are compatible if their behaviors are compatible. So the interaction between two compatible partners is deadlock-free.

Definition 8 *let $\omega_i = (P_i, In_i, TIOTS_{B_i})$ be a partner's definition. The partner's behavior is ambiguous iff $\nexists \omega_j = (P_j, In_j, TIOTS_{B_j})$ such that $TIOTS_{B_j} \asymp TIOTS_{B_i}$*

A service (or partner) is ambiguous if there is no possible partner's behavior that insure a deadlock-free interaction according the interaction relation.

According to the previous relation, the example of the figure 2 is obviously ambiguous. In fact, the states s_2 and s_3 are internally accessible, we call them black zone. Services adopt a passive behavior which is not possible from all the states of the black zone. Thus, s_2 cannot receive $cash$ and s_3 cannot receive $cardNum$. For more details on the interaction relation, the interested readers can refer to [12] and [9]. In the following we introduce two algorithms the first check the compatibility between two TIOTS (i.e two services behaviour) and the second check the ambiguity of a TIOTS (i.e a service bahaviour).

5.2 The compatibility algorithm

We propose an algorithm that verifies whether two TIOTS are related with the interaction relation (definition 6). While initial states must be related with the interaction relation, the algorithm starts with the $\varepsilon - closur$ of s_{01} and s_{02} respectively noted $\varepsilon_{s_{01}}$ and $\varepsilon_{s_{02}}$. The algorithm maintains a stack of couples of states subset (to be processed). Initially, the stack contains $(\varepsilon_{s_{01}}, \varepsilon_{s_{01}})$. The algorithm maintains also a set of couples called R representing the processed ones. Let us describe one step of the algorithm.

- for a given couple (S_1, S_2) in the stack:
- if $(S_1, S_2) \in R$ (which mean that $\exists (S'_1, S'_2) \in R$ with $S_1 \subseteq S'_1$ and $S_2 \subseteq S'_2$) then $unstack$ next couple, else;
- for each $s1 \in S_1$:
  - Let $a \in !M \cup \{\chi, \surd\}$ and $s_1 \overset{a}{\Longrightarrow}_1 s'_1$, if there is a $s_2 \in S'_2$ where $\nexists s_2 \overset{a^c}{\Longrightarrow}_2 s'_2$ then stop and return "no compatible". Else $stack(\bigcup_{s_i \in S_1} \varepsilon_{Next(s_i,a)}, \bigcup_{s_j \in S_2} \varepsilon_{Next(s_j,a^c)})$[6]
  - Let $a \in ?M$ and $s_1 \overset{a}{\Longrightarrow}_1 s'_1$, if there is a $s_2 \in S'_2$ where $\nexists s_2 \overset{a'}{\Longrightarrow}_2 s'_2$ ($a' \in !M$) then stop and return "no compatible". Else $stack(\bigcup_{s_i \in S_1} \varepsilon_{Next(s_i,a)}, \bigcup_{s_j \in S_2} \varepsilon_{Next(s_j,a^c)})$
- The same steps are executed for each $s_2 \in S_2$.

The algorithm checks if all the possible current states of the two TIOTS accessible by the same complementary actions are related by the interaction relation. It starts by the two initial states of the two TIOTS then it calculates their ε closures, if the interaction relation is correct for each couple of states of the two set of states, then the algorithm calculates the next set of states otherwise the two TIOTS are "non compatible". Note that the algorithm is similar to the bi-simulation check algorithm but, in each step, instead of checking the bi-simulation relation on the calculated set of states we check our interaction relation properties.

5.3 The ambiguity check algorithm

According to the definition 8 a partner behaviours is ambiguous if it do not admit a possible compatible partner. The problem can be resolved by checking wether the partner behaviour can admit a correct minimal client (the client is a deterministic TIOTS that allow the use of the service

[6]The $stack()$ function add only the couple if they are not in R.

without dead-lock). The correct client synthesis algorithm builds the deterministic TIOTS following a kind of determinization of the TIOTS of the service. Each state of the potential client is associated to a subset of states of the service. There is a stack of couples (s_1, S'_2) to be processed. Let us describe one step of the algorithm.

- At first, one completes S'_2 with $\overset{\epsilon}{\Longrightarrow}_2$ transitions.
- If a state of the client s'_1 is already associated to S'_2 then one redirects all the input edges of s_1 to s'_1 and one deletes s_1.
- Otherwise for each $s_2 \overset{a}{\Longrightarrow}_2 s'_2$ with a and $s_2 \in S'_2$, one builds a new vertex s'_1 and a new edge $s_1 \overset{a^c}{\Longrightarrow}_1 s'_1$ and one stacks (s'_1, S^*_2) where $S^*_2 = \{s'_2 | \exists s_2 \in S'_2, \exists s_2 \overset{a}{\Longrightarrow}_2 s'_2\}$
- Let $a \notin \{?m\}_{m \in M}$ such that $s_1 \overset{a}{\Longrightarrow}_1 s'_1$, if there is a $s_2 \in S'_2$ with $\nexists s_2 \overset{a^c}{\Longrightarrow}_2 s'_2$ then stop and return "service ambiguous".
- Let $s_1 \overset{?m}{\Longrightarrow}_1 s'_1$, if there is a $s_2 \in S'_2$ with $\nexists s_2 \overset{!m'}{\Longrightarrow}_2 s'_2$ then stop and return "service ambiguous".

The algorithm starts with the couple (s_{01}, s_{02}) in the stack and stops either when the stack is empty (i.e. the client has been built) or when it has detected the ambiguity of the service. The completeness of algorithm is easy to proof since the deterministic equivalent algorithm is complete in the case of discreet time event systems.

6 Choreography correctness

Let us recall the aim of our work. We have a set of partners that would like to realize a specific choreography. Each partner defines its observable behavior (its communication and time constraints). The choreography is defined by fixing the interaction points between partners. We would like to check if, according to their observable behaviors and the interaction points (the set of functions ϕ_i), they are compatible from the interactional point of view i.e. the whole choreography is deadlock-free. We consider, for the following, the choreography C as $C = \{(\omega_i = (P_i, In_i, TIOTS_{B_i(X_i)}), \phi_i)\}$

According to the previous section, we can deduce from the definition 8 the following corollary.

Corollary 1 *Let C be a choreography and let ω_i be a part of it. If $TIOTS_{B_i}$ is ambiguous then C is not deadlock free.*

Given a choreography, if one of the involved partners expose an ambiguous behavior, then the choreography is not deadlock-free. The proof is simple, while an ambiguous partner does not admit a possible compatible partner.

$P \parallel_A Q$	**R($\parallel_1$):** $\forall a \in A \dfrac{P \xrightarrow{a} P', Q \xrightarrow{a^c} Q'}{P \parallel_A Q \xrightarrow{\tau} P' \parallel_A Q'}$
	R($\parallel_2$): $\dfrac{P \xrightarrow{\surd} P', Q \xrightarrow{\surd} Q'}{P \parallel_A Q \xrightarrow{\surd} P' \parallel_A Q'}$
	R($\parallel_3$): $\dfrac{P \xrightarrow{\surd} P', Q \xrightarrow{a} Q'}{P \parallel_A Q \xrightarrow{\tau} Q}$ where $a \neq \surd$
	R($\parallel_4$): $\dfrac{P \xrightarrow{\chi} P', Q \xrightarrow{\chi} Q'}{P \parallel_A Q \xrightarrow{\chi} P' \parallel_A Q'}$
	R($\parallel_5$): $\dfrac{P \xrightarrow{\chi} P', Q \not\xrightarrow{\chi}}{P \parallel_A Q \xrightarrow{\chi} P' \parallel_A Q}$
	R($\parallel_6$): $\forall a \notin A \cup \{\chi, \surd\} \dfrac{P_j \xrightarrow{a} P'}{P \parallel_A Q \xrightarrow{a} P' \parallel_A Q}$
	R($\parallel_7$): $\forall a \notin A \cup \{\chi, \surd\} \dfrac{Q \xrightarrow{a} Q'}{P \parallel_A Q \xrightarrow{a} P \parallel_A Q'}$

Table 2. The transition rules of the partners composition operator

Whatever the partner's behaviors, the interaction with the ambiguous partner may lead to a deadlock i.e. the whole choreography may also deadlock.

6.1 Partner Composition

Each partner interacts with a subset of partners involved in the choreography. From its point of view, it could be considered as interacting with a unique partner that simulates the behavior of the partners composition. Thus, based on the composition properties of process algebra, we introduce a new operator to *BPELProcess* that aggregates two partners in a new one in such a way that preserves the whole choreography. We call the new language $BPELProcessC$ (see the table 2 for the operational semantics of this operator) defined by $BPELProcessC = BPELProcess | P \parallel_A Q$ where $A \subset (?M \cup !M)$

The two composed partners act in parallel. Rules **1** describe the synchronization over communication actions. The communication between partners is not observable from the composed one. The rules **2** and **3** state that the two composed services can terminate in the same time. If one of the two services terminates before the other then its termination action will not be observed and the termination of the composed one will be termination of the reminder service. Rules **4** and **5** concern the composed Services face to time. The time advance synchronously if the two partners can do so (**4**) and asynchronously if only one of the two can let time pass i.e. if at least one partner is timed, the resulted partner is timed too (**5**). Rules **6** and **7** states that the resulted partner can execute the union of transitions of the two partners. We use $Com(\omega_i, \omega_j) = \{*\omega^*.m \in (L_i \cap L_j) | \omega^* \in \{\omega_i, \omega_j\}^*\}$ to represent the set of all the exchanged messages between two partners.

Definition 9 *Let $\omega_i = (P_i, In_i, TIOTS_{B_i})$ and $\omega_j =$*

$(P_j, In_j, TIOTS_{B_j})$ *two partner. The composition of the two partner noted by* $\omega_i \parallel \omega_j$ *is a partner* ω_k *with:*

- $\omega_k = (P_k, In_k, TIOTS_{B_k})$
- $P_k = \{\omega_l.o \in (P_j \cup P_i) | l \neq i, j\}$
- $In_k = (In_i \cup In_j)$
- $TIOTS_{B_k} = TIOTS_{B_i \|_A B_j}$ *with* $A = Com(\omega_i, \omega_j)$

If more than two partners are involved in the choreography, it is important to guarantee the associativity of pair wise composition. Note that this is easy to prove since in C we guarantee a unique identifier for each partner and a unique target for each communication exchanged message. We use the $\prod_{i=1...n} \omega_i = \omega_1 \| .... \| \omega_n$.

6.2 Choreography correctness

From each partner, we can consider the choreography as a dyadic interaction between the partner and the composition of all others partners.

Definition 10 *We call the environment of a partner* $\omega_i \in C$*, noted by* $Env^C_{\omega_i} = \prod_{j \neq i} \omega_j$*. We note its TIOTS by* $TIOTS_{\overline{\omega_i}^C}$.

We can now use our interaction relation to validate the behavior of one partner in the choreography.

Definition 11 *Let C be a valid choreography and* $\omega_i \in C$ *part of it.* ω_i *is compatible with C iff*

$$TIOTS_{B_i(\phi_i(X)} \asymp TIOTS_{\overline{\omega_i}^C}$$

A partner is compatible with its choreography if the interaction with the aggregation of its partners is deadlock free (according the interaction relation 6).

Definition 12 *Let C be a valid choreography, C is a correct choreography* iff $\forall \omega_i \in C$ *we have* ω_i *compatible with C.*

A choreography C is correct if each involved partner in the choreography interacts correctly (full use, deadlock-free and termination) with its partners. Our method can be applied in a distributed manner while each partner can decide locally about its compatibility. If at least one partner is not compatible then the choreography can deadlock. Note that the compatibility check algorithm can gives an explanation of the ambiguity (non-compatibility) which can be used to help the service designer to adapt the behaviour to the partners one. It is important to signal that this method is incremental while it can be used in online choreography establishing. An other use case of our method can be the choreography repair, when one partner fails and we try to replace it. In addition to the functionality equivalence the new partner must be compatible with the choreography environment.

7 Conclusion

In this paper, we address the problem of choreography correctness (deadlock-free and termination detection) in the context of distributed complex Web services. In order to achieve this goal, we give an operational semantic to abstract BPEL in terms of Timed Input Output Transition Systems. Due to abstraction purposes, we show that the observable behaviors can be ambiguous (i.e. may deadlock) under the assumption of full use of the services functionalities. We characterize such ambiguity by defining a relation between two communicating systems. The relation allows to check the compatibility (in the interaction level) between two services involved in the choreography. Using this relation, we propose an algorithm that checks the compatibility between two TIOTS. The choreography correctness problem is then reduced to a bipartite compatibility problem by introducing an aggregation operator. The method allows a distributed checking since each partner can check locally its correctness by checking its compatibility with the aggregation of all its partners. The choreography is correct if all the partners are compatible. The originality of our approach against a central correctness checking, is that it can be used incrementally, this is very useful in the case of online choreography establishing or repairing (replacing a partner by an other).

References

[1] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services v1.1, may 2003.

[2] M. Baldoni, C. Baroglio, A. Martelli, V. Patti, and C. Schifanella. Verifying the conformance of web services to global interaction protocols: A first step. In *International Workshop on Web Services and Formal Methods (WS-FM'05)*, Versailles, France, 2005. Springer-Verlag.

[3] J. Bergstra and J. Klop. Process algebra for synchronous communication. *Information and Control, 60(1-3)*, pages 109–137, 1984.

[4] M. Bravetti, C. Guidi, R. Lucchi, and G. Zavattaro. Supporting e-commerce systems formalization with choreography languages. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 831–835, New York, NY, USA, 2005. ACM Press.

[5] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Towards a formal framework for choreography. In *WETICE '05: Proceedings of the 14th IEEE International Workshops*

on Enabling Technologies: Infrastructure for Collaborative Enterprise, pages 107–112, Washington, DC, USA, 2005. IEEE Computer Society.

[6] O. C., van der Aalst, S. Breutel, D. M. ter Hofstede, and V. H. Formal semantics and analysis of control flow in ws-bpel. echnical Repor BPM-05-13, BPM, 2005.

[7] A. Ferrara. Web services: a process algebra approach. In ACM, editor, *ICSOC*, pages 242–251, 2004.

[8] S. Haddad, T. Melliti, P. Moreaux, and S. Rampacek. A dense time semantics for web service specification languages. In *Proceedings of International Conference Of Telecomunication Theories, ICTTA*, pages 287–295, Damascus, Syria, 2004.

[9] S. Haddad, T. Melliti, P. Moreaux, and S. Rampacek. Modelling web services interoperability. In *Proceedings of the Sixth International Conference on Entreprise Information Systems*, pages 287–295, Porto, Portugal, 2004.

[10] R. Hamadi and B. Benatallah. A Petri Net-based model for web service composition. In K.-D. Schewe and X. Zhou, editors, *Fourteenth Australasian Database Conference (ADC2003)*, volume 17 of *CRPIT*, pages 191–200, Adelaide, Australia, 2003. ACS.

[11] C. Hoare. *Communicating sequential processes*. Prentice Hall, 1985.

[12] T. Melliti and S. Haddad. Synthesis of agents for web services interaction. *International Confrence Electronic Commerce, workshop*, Sept 2003.

[13] L. Meredith and S. Bjorg. Contracts and types. In *Communications of the ACM*, volume 40(10), pages 41–47, 2003.

[14] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[15] S. Nakajima. Model-checking verification for reliable web service. In *Workshop on Object-Oriented Web Services,*, Seattle, Washington, 2002.

[16] X. Nicollin and J. Sifakis. The algebra of timed process, ATP: Theory and application. Technical report, Technical Report RT-C26, Institut National Polytechnique De Grenoble, 1991.

[17] B.-H. Schlingloff, A. Martens, and K. Schmidt. Modeling and model checking web services. *Electr. Notes Theor. Comput. Sci.*, 126:3–26, 2005.

[18] K. Schmidt and C. Stahl. A Petri Net semantic for BPEL4WS - validation and application. In E. K. (ed.), editor, *Workshop on Algorithms and Tools for Petri Nets (AWPN 04)*, pages 1–6, Germany, 2004.

[19] C. Stahl. A Petri Net semantics for bpel. technical repport 10099, Humboldt-Universitat zu Berlin, 2004.

[20] van Breugel Franck and K. Mariya. Dead-path-elimination in BPEL4WS. In *5th International Conference on Application of Concurrency to System Design*, St Malo, 2005. IEEE.

[21] W. van der Aalst, M. Dumas, C. Ouyang, A. Rozinat, , and H. Verbeek. Choreography conformance checking: An approach based on BPEL and Petri Nets, may 2005.