CLIENT SYNTHESIS FOR WEB SERVICES BY WAY OF A TIMED SEMANTICS

Serge Haddad
LAMSADE, Université Paris Dauphine
Place du Maréchal de Lattre de Tassigny, 75775 Paris Cedex 16, FRANCE
haddad@lamsade.dauphine.fr

Patrice Moreaux
LISTIC-ESIA, Université de Savoie
Domaine universitaire d'Annecy le vieux, BP 806, 74016 Annecy Cedex, FRANCE
patrice.moreaux@univ-savoie.fr

Sylvain Rampacek
CReSTIC, Université de Reims Champagne-Ardenne
UFR Sciences de Reims, BP 1039, 51687 Reims Cedex 02, FRANCE
sylvain.rampacek@univ-reims.fr

Keywords: Web Services, BPEL4WS, Algebra of timed processes, Timed Automata.

Abstract: A complex Web service described with languages like BPEL4WS, consists of an executable process and its observable behaviour (called an abstract process) based on the messages exchanged with the client. The abstract process behaviour is non deterministic due to the internal choices during the service execution. Furthermore the specification often includes timing constraints which must be taken into account by the client. Thus given a service specification, we identify the synthesis of a client as a key issue for the development of Web services. To this end, we propose an approach based on (dense) timed automata to first describe the observable service behaviour and then to build correct interacting clients when possible. The present work extends a previous discrete time approach and overcomes its limitations.

1 Introduction

From elementary Web services to complex ones Web services are "self contained, self-describing modular applications that can be published, located, and invoked across the Web" (Tidwell, 2000). They are based on a set of independent open platform standards to reach a high level of acceptance. Web services framework is divided into three areas: communication protocol, service discovery and service description. The "Web Services Description Language" (WSDL) (WSDL, 2001) provides a formal, computer-readable description of Web services. Such a description specifies the software component interfaces listing the collection of operations that are network accessible through standard XML messaging. It includes all information that an application needs to invoke such as the message structure, the response structure and some binding information like the transport protocol, the port address, etc.

However simple operation invocation is not sufficient for some kind of composite services. They require in addition a long-running interaction derived by an explicit process model. This kind of services may often be encountered in two cases. First when a Web service is developed as an agent, it is composed by a set of accessible operations and a process model which schedules the invocation to a correct use of the service. Secondly, facing to the capability limits of Web services, composite services may be obtained by aggregating existing Web services in order to create more sophisticated services (and this in a recursive way).

In order to deal with the behavioural aspects of complex services, some industrial and academic specifications languages have been introduced. Among them, "Business Process Execution Language for Web Services" (BPEL4WS or more succinctly BPEL) has been proposed by leading actors of industry (BEA, IBM, and Microsoft) and has quickly become a standard (Andrews et al., 2003).

The two facets of complex Web services BPEL supports two different types of business processes (see for instance (Juric, 2005), (Juric et al., 2005)):

- Executable processes specify the exact details of business processes. They can be executed by an orchestration engine.

- Abstract business protocols specify the public message exchange between the client and the service. They do not include the internal details of process flows but are required in order for the client to correctly interact with the service.

Given the description of an executable process, its associated interaction protocol is obtained by an abstraction mechanism (which masks all the internal operations of the service). However the issues raised by these two types of processes are very different. A specification of an executable process is close to the definition of a program whereas the specification of interaction protocol mainly raises an difficult problem: how to synthetize a client which will correctly handle the interaction with the service.

The synthesis problem Indeed by construction, the external behaviour of a service is non deterministic due to its internal choices. It is then *a priori* unclear whether a client, i.e. a deterministic program, can be designed to interact with it. Furthermore the specification often includes timing constraints (e.g. implicit detection of the withdrawal of an interaction by the client) implying that these timing constraints must also be taken into account by the client. However since no semantics of the interaction process is given for BPEL (not to be confused with the semantics of the service execution), this problem could not be formally stated.

A discrete time solution (Melliti and Haddad, 2003; Haddad et al., 2004b) In this work, the authors have specified what is an external behaviour, i.e. they have given an operational semantics to an abstract BPEL specification in terms of a discrete time transition system. The semantics is obtained by a set of rules in a modular way. Given a constructor of the language and the behaviour of some components, a rule specifies a possible transition of a service built via this constructor applied on these components. As previously discussed, the transition system is generally non deterministic.

Then they have defined a relation between two communicating systems which formalizes the concept of a correct interaction. There are standard relations between dynamic systems like the language equivalence and the bisimulation equivalence but none of them matches their needs. Thus they have introduced the interaction relation which can be viewed as a bisimulation relation modified in order to capture the nature of the events (i.e. the sending of a message is an action whereas the reception is a reaction).

Afterwards they have focused on the synthesis of a client which is in an interaction relation with the transition system corresponding to the system. The client they look for must be implementable, in other words it should be a deterministic automaton. It has appeared that some BPEL specifications do not admit such a client i.e. they are inherently ambiguous. Thus the algorithm they have developed either detects the ambiguity of the Web service or generates a deterministic automaton satisfying the interaction relation. The core of this algorithm is a kind of determinisation of the transition system of the service.

Our present contribution In the previous solution, the discrete time semantics is preferred for simplicity reasons. However such a solution has the following drawbacks. First, the passing of a unit of time is modelled by an explicit transition in the transition system which means that the compact representation of timing constraints by values is now hidden in the model by their combination with logical transitions. In other words, whereas handling correctly the interaction with the service, the client automaton is hardly understandable by a user. Moreover if two timing constraints are not of the same order, the time unit must be chosen w.r.t. the shorter one leading to a combinatory explosion of the automaton due to the "translation" of the longer one.

Here we develop a dense time semantics for a BPEL specification as a timed automaton (Alur and Dill, 1994). A preliminary version of this semantics for XLANG (Thatte, 2001) was given in (Haddad et al., 2004a). Switching from XLANG to BPEL is relatively easy thanks to the way the operational semantics is defined. The construction of this automaton is based on modular rules which works similarly to the discrete case. However they are more intricate since, on the one hand, the values of the timing constraints are handled symbolically with the help of clocks and, on the other hand, given some expression we must determine which clocks are active and how they govern the guards of the transitions. Next, we revisit the interaction relation. Contrary to the previous relation, we manage explicitly the time since now an execution is a discrete event sequence where the events are stamped by the time of their occurrence. The last part of the work is the most difficult one. Let us recall that the core of the former synthesis algorithm for the client automaton is similar to an automaton determinisation. It is well-known that the non deterministic timed automata are strictly more expressive than the deterministic ones. Thus we have designed an original procedure which decides whether a specification is (potentially) ambiguous and in the negative case produces such a deterministic automaton. Similarly to the approaches which determinise subclasses of timed automata (Alur et al., 1999), we require that the deterministic automaton has the same clocks as those of the original one.

Related work The platform WSAT (Fu et al., 2004a; Fu et al., 2004b) enables designers of a Web service composition to model check properties expressed by LTL formulas with SPIN tool. The formal semantics is obtained by gluing patterns for each BPEL construction. One pattern is connected from its final state to the initial state of next pattern according to the BPEL description with local transitions. This work does not cover the time features and it focuses only on message exchanges: the conversation is obtained by a *virtual watcher* that is supposed to record all messages sequences sent by each peer enrolled in the composition.

Another research of Web services formal semantics is based on a BPEL to Finite State Processes (FSP) translation (Foster et al., 2003). This work lies on message sequence charts and the core of the verification mechanism consists to check trace equivalence. Again, the time features of the specification are not taken into account.

(Turner, 2005) uses the notation CRESS (Chisel Representation Employing Systematic Specification) to formalise Web services. This model presents two main advantages: automatic translation into formal languages for analysis as well as into implementation languages for deployment. Then the CRESS specification is translated into LOTOS and analysed with tools like TOPO, LOLA and CADP. Again, the temporal aspects are not present.

These different contributions share with our approach the design of a formal semantics for Web services. However they study the BPEL execution process and not the interaction protocol, they do not include the time features of BPEL and they perform component verification whereas we perform component synthesis.

Organization of the paper In section 2, we describe how to associate a formal semantics with a business process leading to the building of a timed automaton. Section 3 presents the client-service interaction relation. The synthesis of a client automaton is described in section 4. We summarize our results in section 5 and we present some directions for future works. The appendix, to be omitted in the final version, gives a detailed presentation of the synthesis algorithm.

2 A formal semantics for BPEL abstract processes

BPEL provides a set of operators describing in a modular way the observable behaviour of an abstract process. As shown in (Staab et al., 2003), this kind of process description is close to the process algebra paradigm illustrated for instance by CCS (Milner, 1989), CSP (Hoare, 1985) and ACP (Bergstra and Klop, 1984). However, time is explicitly present in some of the BPEL constructors and thus the standard process algebra semantics are inappropriate for the semantics of such a process.

Thus our semantics associates a timed automaton (TA) with an abstract process. Let us briefly describe what is a TA. A TA is a (non deterministic) finite automaton enhanced with a finite set of clocks. An execution of a TA consists of (possibly null) time steps interleaved with discrete transitions. A TA configuration is composed of a state and a value per clock (called a clock valuation). The discrete transitions correspond to the edges between the states. The clocks control the behaviour of the TA as follows. A boolean expression relative to the clocks called an invariant is attached to each state. Time can elapse in a state if the invariant associated with the state is satisfied during all the successive configurations. In addition to its label, the specification of an edge includes a similar boolean expression, here called a guard, and a subset of clocks to be reset. In order to follow this edge from a configuration, the corresponding guard must be true and the clock valuation after the reset operation must satisfy the invariant of the new state.

Definition 1 (Timed automaton (Alur and Dill, 1994)) *A Timed Automaton (TA) is a tuple* $T = (L, C, A, E, l_0)$ *where* L *is the set of* locations *or control states,* C *is the set of* clocks*,* A *is the set of* actions*,* $E \subseteq L \times C \times A \times C \times L$ *is the set of* edges*. An edge* e *is* (s, g, a, r, d) *with* s *the source location,* g *the guard of* e*,* a *the action,* r *the subset of clocks reset by* e *and* d *the destination location.* l_0 *is the initial location.*

2.1 The alphabet of the TA

The first step for the defining a semantics consists in specifying the action alphabet for a BPEL process. We have four kinds of actions:

- Silent actions, denoted by τ cannot be observed by the client. They correspond to decisions taken by the server (evaluation of a condition for switch, while, etc.). The exception events set of BPEL4WS is denoted by E_x.
- Timeout expirations which are denoted by to.
- Sending and receiving messages: the set of types of messages will be denoted by M. The emission is denoted by $!m$ and the reception is denoted by $?m$. We also set $!M = \{!m \,|\, m \in M\}$ and $?M = \{?m \,|\, m \in M\}$ and the wildcard $*$ may be substituted for ! or ?.

- 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.

2.2 The states of the TA

Each state will be associated with a BPEL process obtained by successive transformations from the initial process. Two states have different associated processes. At the beginning of the construction, there is a single state (the initial one) corresponding to this process. Each time an edge is defined, a new process is computed and if this process does not label an existing state then such a state is created. Due to the semantic rules given in the next subsection, it can be proved that the number of derived processes is finite (and thus the number of states is also finite).

2.3 The edges of the TA

The edges starting from a state are obtained by a top down analysis of the process expression labelling this state. This analysis is usually defined with the help of operational semantic rules. The definition of a semantic rule $[op_x]$ for a generic process $P = op_x(P_1, P_2, \ldots)$ is given as for Algebra of Timed Processes (Nicollin and Sifakis, 1994). Components of a rule are:

- 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 by $guard(\{\alpha_i\})$.
- 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 subprocesses transition and
- the new process is an expression $Nexp(P, \{P'_{o(i)}\})$ depending on the original process and the new subprocesses.

So, a generic rule, presented with the usual style has the following structure:

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

The guard and reset associated with an edge will be defined in the next subsection. For sake of readability, we do not follow the (verbose) XML syntax of a BPEL process. Instead we have chosen a simplified syntax close to the one used for process algebra whose meaning should be immediate for who knows BPEL. As usual, we begin the definition of rules by giving the ones corresponding to the basic processes of BPEL. These basic processes are `empty`, $?o[m]$, $!o[m]$ and `throw`$[e]$.

The `empty` process `empty` can only terminate (the notation 0 is the null process).

$$\texttt{empty} \xrightarrow{\surd} 0$$

The $?o[m]$ and $!o[m]$ processes The process $?o[m]$ (which corresponds to the input operation of WSDL) consists in receiving a message of type m. The process $!o[m]$ (which corresponds to the notification operation of WSDL) consists in sending a message of type m. We consider only these two types of WSDL operations. The two other types can be built with the sequence constructor (see below).

$$*o[m] \xrightarrow{*m} \texttt{empty} \quad \text{with} * \in \{?, !\}$$

The `throw` process The process `throw`$[e]$ raises an exception e which must be catched in some `scope` process.

$$\forall\, e \in E_x,\ \texttt{throw}[e] \xrightarrow{e} 0$$

The `sequence` process (;) The process $P\ ;\ Q$ executes the process P then the process Q. Since the operator ";" is associative, we safely restrict the number of operands to two processes. The `sequence` process acts as its first subprocess while this process does not indicate its termination. In the latter case, the `sequence` process acts as the second process can do.

$$\forall\, a \neq \surd,\ \frac{P \xrightarrow{a} P'}{P\ ;\ Q \xrightarrow{a} P'\ ;\ Q}$$

$$\forall\, a,\ \frac{P \xrightarrow{\surd} \text{ and } Q \xrightarrow{a} Q'}{P\ ;\ Q \xrightarrow{a} Q'} \text{ where } a \in \{!m, ?m, \surd, \tau\}$$

Note that if there is an action $a \neq \surd$ such that $P \xrightarrow{a} P'$, then $P \xrightarrow{\surd}$ cannot occur.

The `switch` process The process `switch`$[\{P_i\}_{i \in I}]$ chooses to behave as one process among the set $\{P_i\}$. Each branch of its execution is guarded by an *internal* condition. Conditions are evaluated w.r.t. the order of their appearance in the description. However since the client has no way to predict the choice of the service, this order is irrelevant. The main consequence is that from the point of view of the client, *this choice is non deterministic*. The `switch` process becomes one of its subprocesses in a silent way. Let us note that we have implicitly supposed that at least one condition is fulfilled. In the other case, it is enough to add the process `empty` as one of the subprocesses.

$$\forall\, i \in I,\ \texttt{switch}[\{P_i \mid i \in I\}] \xrightarrow{\tau} P_i$$

The `while` **process** The process $\texttt{while}[P]$ iterates an inner process as long as an *internal* condition is satisfied. Like `switch`, `while` evaluates in a silent way its condition. Thus we have two rules depending on this internal evaluation.

$$\texttt{while}[P] \xrightarrow{\tau} P \,;\, \texttt{while}[P]$$

$$\texttt{while}[P] \xrightarrow{\tau} \texttt{empty}$$

The `flow` **process** The process $\texttt{flow}[\{P_i\}_{i\in I}]$ simultaneously activates a set of processes $\{P_i\}$. For the moment considering that the synchronization primitives of BPEL are internal ones we have not yet implemented this synchronization. Thus this parallel execution is similar to a "fork-join" in the sense that the combined process ends its interaction when all subprocesses have completed their execution. Subprocesses of a `flow` process act independently except for one action: they simultaneously indicate their termination. In the latter case, the `flow` process becomes the null process. Furthermore internal actions are considered as immediate and consequently the occurrence of such an action in a subprocess prevents the occurrence of a delayed action (sending or reception of a message) in another subprocess.

• Individual actions:

1.
$$\frac{\forall\, a \in E_x \bigcup \{\tau\},\quad \exists\, j \in I,\ P_j \xrightarrow{a} P'}{\texttt{flow}[\{P_i \mid i \in I\}] \xrightarrow{a} \texttt{flow}[\{P_i \mid i \in I \backslash \{j\}\} \bigcup \{P'\}]}$$

2.
$$\frac{\forall\, m \in M,\quad \exists\, j \in I,\ P_j \xrightarrow{*m} P' \text{ and } \forall\, i \neq j,\ \forall\, a \in E_x \bigcup \{\tau\},\ \text{not } \exists k \in I,\ (P_i \xrightarrow{a})}{\texttt{flow}[\{P_i \mid i \in I\}] \xrightarrow{*m} \texttt{flow}[\{P_i \mid i \in I \backslash \{j\}\} \bigcup \{P'\}]}$$

• Termination:

$$\frac{\forall\, i \in I,\ P_i \xrightarrow{\surd} P'_i}{\texttt{flow}[\{P_i \mid i \in I\}] \xrightarrow{\surd} 0}$$

The `scope` **process** $\texttt{scope}(P, E)$ with

$$E \stackrel{def}{=} [\{(m_i, P_i) \mid i \in I\}, (d, Q), \{(e_j, R_j) \mid j \in J\}]$$

may evolve due to P evolution, reception of a message m_i, expiration of the timeout with duration d or occurrence of an exception e_j. We note $M_I = \{m_i \mid i \in I\}$ and $E_J = \{e_j \mid j \in J\}$.

• P actions: The termination exits the scope whereas another action does not.

$$\frac{P \xrightarrow{\surd}}{\texttt{scope}(P, E) \xrightarrow{\surd} 0} \qquad \frac{P \xrightarrow{a} P'}{\texttt{scope}(P, E) \xrightarrow{a} \texttt{scope}(P', E)}$$

• Receiving a message m_i:

$$\forall\, i \in I,\ \frac{\forall\, a \in E_x \bigcup \{\tau, \surd\},\ \neg(P \xrightarrow{a})}{\texttt{scope}(P, E) \xrightarrow{?m_i} P_i}$$

• Exception handling: which depends whether the raised exception is catched in this scope.

$$\forall\, j \in J,\ \frac{P \xrightarrow{e_j}}{\texttt{scope}(P, E) \xrightarrow{\tau} R_j}$$

$$\forall\, e \notin E_J,\ \frac{P \xrightarrow{e}}{\texttt{scope}(P, E) \xrightarrow{e} 0}$$

If an exception e is never catched at any level then the process is an erroneous one which can straightforwardly checked by examining whether an exception labels an edge of the TA.

The `pick` **process** can be viewed as a particular case of the `scope` process.

2.4 The clocks of the TA: invariants, guards and reset operations

We associate a clock with each `scope` process and a special clock (x_{im}) for handling the immediate actions. Given a process, we determine by a top down analysis which clocks are active, i.e. which `scope` subprocesses are activated. The invariant associated with a state depends whether an immediate action is possible. If it is the case, the invariant is $x_{im} = 0$ else the invariant is a conjunction over the active clocks of elementary conditions $x \leq d$ where d is the value defined in the scope corresponding to x.

The clocks to be reset when following an edge are simply the clocks which were inactive in the source process and become active in the target process. x_{im} is always reset.

There is no guard for the transitions defined by the operational rules. However, we add to each state which owns active clocks, a set of edges labelled by to (one per subset of active clocks which can simultaneously reach their bound). For each such edge the guard specifies that these active clocks have reached their bound while the other ones have not.

2.5 The overall TA construction

The computation of the timed automaton of the service can now be summarized as follows.

- It manages a set of processes to be examined and a current version of the TA. It starts with the initial process and a automaton reduced to a single state.

- When examining a process, it first builds the edges corresponding to the operational rules and for each target process not already present in the TA, it adds it to the set of processes to be examined.
- Then it determines the set of active clocks of the current process. Based on this information and the previous edges, it determines the invariant of the state. Afterwards, it generates the "time-out" edges.
- The information updating about the clock reset may take place at two different moments. If an edge points to an already encountered state then (based on the active clocks of the two states) this information is immediately updated. Otherwise, it will be updated when the target state will be examined.

3 Interaction relation

It should be clear that the TA is a compact description of the observable behaviour of the BPEL process. However, as briefly sketched above, TA have themselves a formal semantics defined in terms of a Timed Transition System (TTS). A TTS over the set of actions A is a tuple $(S, s_0, A, \rightarrow)$ where S is a set of states, $s_0 \in S$ is the initial state, A is a finite set of actions disjoint from time passing, and $\rightarrow \subseteq S \times (A \cup \mathbb{R}_{\geq 0}) \times S$ is a set of edges. If $(q, e, q') \in \rightarrow$, we also write $q \xrightarrow{e} q'$. $q \xrightarrow{d} q'$ with $d \in \mathbb{R}_{\geq 0}$ corresponds to d units of time passing. The states of the TTS associated with a TA are simply the reachable configurations and its edges are either its discrete transitions or time passing in a location.

We first informally what should be a correct interaction between two TTS. As for the bisimulation relation, we require a relation between pairs of states of the two systems. Obviously the pair consisting of the initial states should belong to this relation.

Furthermore, the states of a pair should have a coherent view of the next interaction steps to occur. At first, this implies that the relation must take into account the mutually observable steps. Thus we introduce the observable transition relation of a TTS by $s \overset{a}{\Rightarrow} s'$ iff $s \overset{\tau^* a \tau^*}{\rightarrow} s'$, $s \overset{\epsilon}{\Rightarrow} s'$ iff $s \overset{\tau^*}{\rightarrow} s'$ and $s \overset{d}{\Rightarrow} s'$ iff $s \overset{d_1 \tau \ldots \tau d_n}{\rightarrow} s'$ with $\sum d_i = d$.

Once it is done, we could require (like for bisimulation) that if a state s of the pair (s, s') may evolve by an observable transition of its TTS to some new state s_1, s' should have a similar observable transition leading to a state s'_1 which would compose with s_1, a new pair of consistent views.

However we need to be careful. First, if a TTS sends a message the other one must be able to receive the message. So it is necessary to introduce the notion of complementary actions $\overline{?m} = !m$, $\overline{!m} = ?m$ and $\forall a \notin \{!m\}_{m \in M} \cup \{?m\}_{m \in M}\ \overline{a} = a$ and to require that the synchronized evolution is obtained via complementary actions.

But this requirement is too strong as it does not capture the different nature of the sending and reception of a message. A sending is an action whereas a reception is a reaction and will not spontaneously occur. Therefore a more appropriate relation will first require that if, in s belonging to the pair (s, s'), a TTS may receive a message m, then there is a third state s'' of the other TTS indistinguishable from s' w.r.t. the observable transitions which can send m and second that in s' the other TTS can send a message (not necessarily m). The first condition expresses that the former TTS is not over specified and the second one that it will not wait indefinitely for a message.

These considerations yield the following formal definition.

Definition 2 (Interaction relation) *Let* $A_1 = (S, s_{01}, A, \rightarrow_1)$ *and* $A_2 = (S, s_{02}, A, \rightarrow_2)$ *be two TTS. Then* A_1 *and* A_2 correctly interact *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* $a \notin \{?m \mid m \in M\}$*; if* $\exists\, s_1 \overset{a}{\Longrightarrow}_1 s'_1$*, then* $\exists\, s_2 \overset{\overline{a}}{\Longrightarrow}_2 s'_2$ *with* $s'_1 \sim s'_2$ *and if* $\exists\, s_2 \overset{a}{\Longrightarrow}_2 s'_2$ *then* $\exists\, s_1 \overset{\overline{a}}{\Longrightarrow}_1 s'_1$ *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*
    - $\exists\, s_1 \overset{!m'}{\Longrightarrow}_1 s'_1$

4 Client timed automaton synthesis

We are now in position to present the client synthesis algorithm. First as the client must be implementable, we require that it is *deterministic*. Second as it must handle clocks to manage on its side the timeout of the service, we need to express its behaviour by a TTS. These consideration lead to choose as model for our client a deterministic timed automaton. Last, we must produce a TA which is in interaction relation with the TA of the BPEL process.

Before developing we emphasize that there exist BPEL process which do not admit clients. For

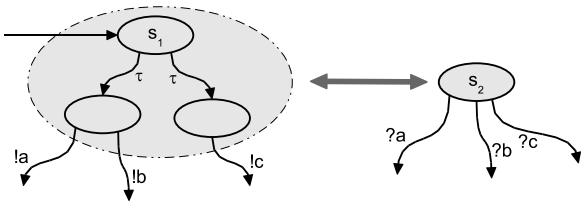

Figure 1: Service subset of states (left) - client state (right)

instance, process $\texttt{switch}[?o[m], ?o[m']]$ internally chooses to receive either a message m or m' and thus no deterministic TA can correctly interact with it since it would imply that, in its initial state, the client should send either m or m' while the server would wait the other message. Note also the difference with process $\texttt{switch}[!o[m], !o[m']]$ where a client can be easily designed: just wait for either m or m'. We say that a process is *ambiguous* if it does not admit a deterministic TA which is in interaction relation with it.

4.1 The synthesis algorithm

Our algorithm does not look for any deterministic TA but restricts its search for a TA which has the same clocks as the TA of the BPEL process. Thus when the algorithm outputs "ambiguity", it just means that no TA with this constraint exists. In other words, our procedure is not complete. However this restriction seems to be reasonable (see the next subsection about incompleteness of our algorithm).

The general principle of our algorithm (see figure 2) is similar to a determinisation procedure: a state of the TA client will correspond to a subset of states of the TA of the service (see fig. 1).

More precisely, each potential state s of the TA client is associated with a subset of states $S_2(s)$ of the TA service which are related to s via the interaction relation. During the construction, there is a stack of client states to be processed. At the beginning of the algorithm, the stack contains an initial client state s_{01} such that $S(s_{01}) = \{s_{02}\}$, s_{02} being the initial state of the service. It stops either when the stack is empty (i.e. the client has been built) or when it has detected the ambiguity of the service.

First, we compute the ϵ-closure by τ-transitions. If this subset (call it S') of service states is already associated with a state s of the client, then the edge of the client TA which has generated the subset is redirected to s. Otherwise, one creates a new client state (say $snew$).

We compute a subset of S' (say S'') that contains only states who has not output τ-immediate-transition.

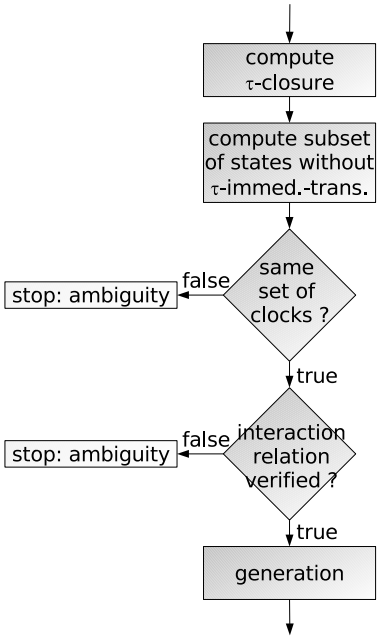

Figure 2: A step of the client synthesis algorithm

Next, we compute the set of clocks for each state of S''. If this clock set is not unique, then there is an *ambiguity* (temporal ambiguity case) and we stop the construction. Afterwards, we check the interaction relation for discrete transitions. If it is not fulfilled then we also stop the construction.

The construction of the clocks guard has two steps. The first one consists to copy clock guards of the edges and clock invariants of the vertice of the TA server and, next, the complementary transitions are constructed.

A detailed description of the algorithm is given in appendix.

4.2 Incompleteness of the algorithm

As already discussed, our algorithm is incomplete. We give now an example of a false detection (see figure 3): a process starting by a `switch` such that one branch of this `switch` starts with a `scope` process and another branch does not activate a timing constraint is detected as an ambiguous service. Indeed in one branch there is an active clock whereas in the other there is none.

In a discrete time framework, the previous (complete) method (Haddad et al., 2004b) produces a client. Indeed the time elapsing is symbolized by an action (χ) and we implicitly work at a (discrete) TTS level. Here we work at a higher level (the TA one). Thus incompleteness is the price to pay in order to obtain a more compact representation of the client.

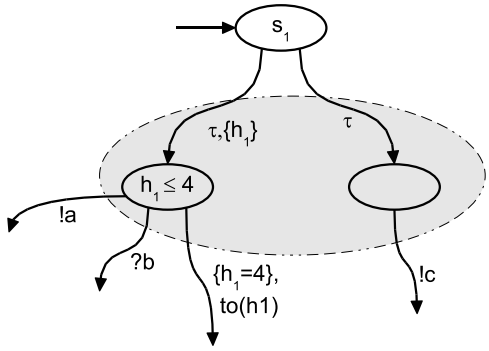

Figure 3: A false detection of ambiguity for the process $\texttt{switch}(!o[c], \texttt{scope}(!o[a], [\{(b, \texttt{empty})\}, (4, \texttt{empty}), \{\}])$

5 Conclusion

We have shown that the interaction with a Web service requires a theoretical development relative to its semantics. Extending a previous work in discrete time, we have proposed in this paper a dense time semantics for BPEL: from the definition of a service - an abstract process - we build a timed automaton corresponding to its formal semantics. Then we have defined an interaction relation between client and service considering them as timed transition systems. We have also designed an algorithm synthetising a deterministic timed automaton (the client) when this is possible and detecting ambiguous service otherwise.

This approach is implemented in our framework ICIS and will be soon reachable from the net. For sake of simplicity, we have considered a perfect communication channel (no loss and no delay). We are currently working on the generalization of our approach by including the specification of the channel characteristics.

A A detailed description of the algorithm

The main types used in algorithm 1 are:

- *graph*: a graph and its associated methods (addVertex,addEdge,g(s)etInitialState,getStates)
- *clientState*: a client state and its associated methods (g(s)etSet)
- *stack*: a stack and associated methods (isEmpty, pop, push)

The function $interacRelationVerified(graph, set\ of\ serverStates)$, used in line 26, verifies if a client state can be created such that it is in interaction relation with every state in set (see in section 3). It returns a boolean (true if interaction relation is verified,

Algorithm 1 CLIENTSYNTHESIS

Parameters
GServer: *graph* (input)
GClient: *graph* (input and output)
Local variables
cs, cs': *clientState*
ss, ss': *serverState*
set, set', front, oldset: *set of serverState*
sa: *set of clocks*
st: *set of transitions*
t: *transition*
Stk: *stack*

1: new(cs)
2: cs.setSet({GServer.getInitialState})
3: Gclient.addVertex(cs)
4: Gclient.setInitialState(cs)
5: Stk.push(cs)
6: **while** not Stk.isEmpty() **do**
7: cs ⇐ Stk.pop()
8: set ⇐ s.getSet()
9: *// compute τ-closure*
10: front ⇐ set
11: **while** not front.isEmpty() **do**
12: oldset ⇐ set
13: **for each** ss ∈ front **do**
14: set ⇐ set ∪ {ss' | ∃ ss $\xrightarrow{\tau}$ ss'}
15: **end for**
16: front ⇐ ss \ oldss
17: **end while**
18: **if** ∃ s' ∈ GClient.getStates | set = s'.getSet() **then**
19: *// there is an equivalent state, merge them*
20: redirect all the input edges of s to s' in GClient
21: **else**
22: cs.setSet(set)
23: set' ⇐ {ss' ∈ set | ¬ ss' $\xrightarrow{\tau}$}
24: sa ⇐ set'.getfirst().getActiveClocks()
25: **if** ∀ ss' ∈ set', ss'.getActiveClocks() == sa **then**
26: **if** interacRelationVerified(GServer, ss) **then**
27: *// create new edges*
28: st ⇐ {t | ∃ s ∈ ss, ∃ s $\xrightarrow{t}$ s', t ≠ τ}
29: **for each** t ∈ st **do**
30: set' ⇐ {ss' | ∃ ss ∈ set, ss $\xrightarrow{t}$ ss'}
31: new(cs')
32: cs'.setSet(set')
33: Gclient.addVertex(cs')
34: Stk.push(cs')
35: GClient.addEdge(cs, $\bar{t}$, cs')
36: **end for**
37: **else**
38: return ambiguity *//interaction ambiguity*
39: **end if**
40: **else**
41: return ambiguity *//time ambiguity*
42: **end if**
43: **end if**
44: **end while**

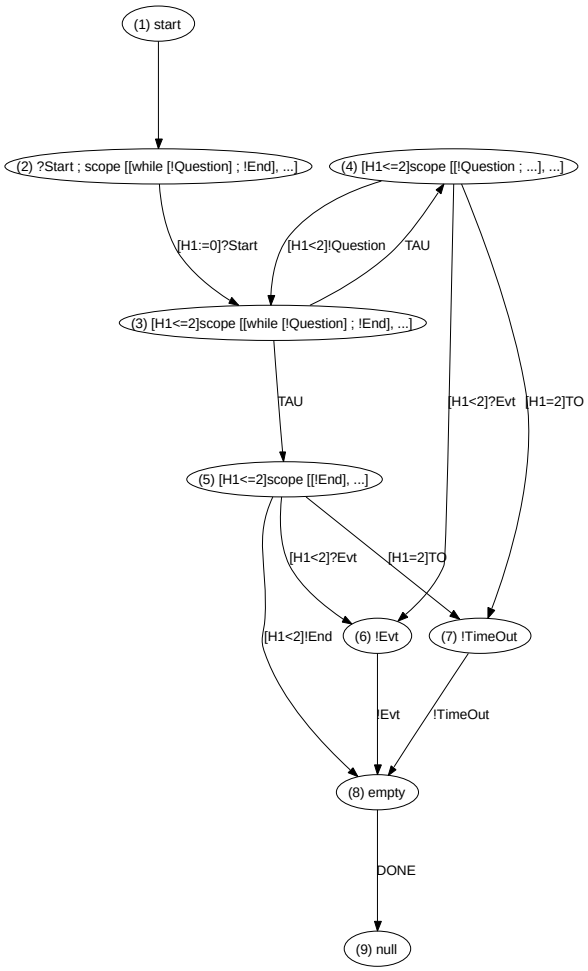

Figure 4: Timed automaton of a service (process : $?Start; \texttt{scope}\ [[\texttt{while}\ [!Question]; !End],\ \{(?Evt,\ !Evt)\}\ (H1 : 2, !TimeOut)\ \{\}]$)

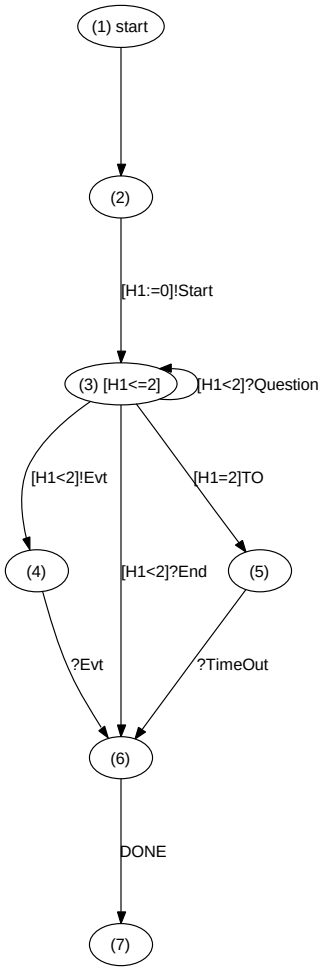

Figure 5: Timed automaton of the client corresponding to the process on figure 4

false otherwise).

B Server and client timed automata example

Server timed automaton

We present, in this section, an example of a client generation from the server process : $?Start; \texttt{scope}\ [[\texttt{while}\ [!Question]; !End],\ \{(?Evt,\ !Evt)\}\ (H1 : 2,\ !TimeOut)\ \{\}]$. This server process receives a $Start$ message, then it starts a `scope` process associated with a $TimeOut$ event (this $TimeOut$ can occur after 2 units of time) and also associated with a reception event called here Evt. The core of this `scope` process is a loop `while` that can send $Question$ zero, one or more times. When the loop terminates, the server sends a message End and derives on $null$ process.

According to our formal semantics (section 2), we can compute the corresponding timed automaton of this process (see figure 4).

Client timed automaton

Figure 5 gives the representation of the client timed automaton generated by our client synthesis algorithm (see Algorithm 1) corresponding to the previous server process.

We can see that some states are merged in the client timed automaton. In fact, the state labelled (2) in the server is present in the client with the same label (2). The only change is the outgoing transition: in the server, it receives $Start$, and in the client, it sends $Start$.

States labelled (3) in service is "more complex". The $\tau - closure$ of the synthesis algorithm returns

the $\{(3),(4),(5)\}$ states set. This states set is important for the server, but the client can not know the current state of the server: there is no message exchange going from state (3) to (4) or (5) but instead only internal transitions. So, this server states set is merged into one state in client: the state (3). The different outgoing transitions of the sets are adapted to the client merging.

Others states follow nearly the same procedure than the server state (2).

REFERENCES

Alur, R. and Dill, D. L. (1994). A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235.

Alur, R., Fix, L., and Henzinger, T. A. (1999). Event-clock automata: a determinizable class of timed automata. *Theoretical Computer Science*, 211(1–2):253–273.

Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., and Weerawarana, S. (2003). Business process execution language for web services.

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

Foster, H., Uchitel, S., J.Magee, , and J.Kramer (2003). Model-based verification of web service compositions. In *Proc. of. the 18th Int. Conf. on Automated Software Eng.*

Fu, X., Bultan, T., and Su, J. (2004a). Analysis of interacting bpel web services. In *Proc. of the 13th International World Wide Web Conference (WWW'04)*, USA. ACM Press.

Fu, X., Bultan, T., and Su, J. (2004b). Wsat: A tool for formal analysis of web services. In *Proc. of the 16th International Conference on Computer Aided Verification (CAV'04)*.

Haddad, S., Melliti, T., Moreaux, P., and Rampacek, S. (2004a). A dense time semantics for Web services specifications languages. In *Proc. of the 1st Int. Conf. on Information & Communication Technologies: from Theory to Applications (ICTTA'04)*, pages 647–648, Damascus, Syria. IEEE France.

Haddad, S., Melliti, T., Moreaux, P., and Rampacek, S. (2004b). Modelling web services interoperability. In *Proc. of the 6th Int. Conf. on Enterprise Information Systems (ICEIS04)*, Porto, Portugal.

Hoare, C. (1985). *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, NJ, USA.

Juric, M. (2005). BPEL and Java. *On line journal theserverside.com*. `http://www.theserverside.com/articles/article.tss?l=BPELJava`.

Juric, M., Sarang, P., and Mathew, B. (2005). *Business Process Execution Language for Web Services*. Packt Publishing.

Melliti, T. and Haddad, S. (2003). Synthesis of agents for web services interaction. In *Workshop Semantic Web Services for Enterprise Application Integration and E-Commerce of the Fifth International Conference on Electronic Commerce*, Pittsburgh, USA.

Milner, R. (1989). *Communication and Concurrency*. Prentice-Hall, Englewood Cliffs, NJ, USA.

Nicollin, X. and Sifakis, J. (1994). The algebra of timed processes, atp: theory and application. *Inf. Comput.*, 114(1):131–178.

Staab, S., van der Aalst, W., Benjamins, V., Sheth, A., Miller, J., Bussler, C., Maedche, A., Fensel, D., and Gannon, D. (2003). Web services: Been there, done that? *IEEE Intelligent Systems*, 18:72–85.

Thatte, S. (2001). Xlang: Web services for business process design. World Wide Web page. http://www.gotdotnet.com/team/xml/wsspecs/xlang-c/default.htm.

Tidwell, D. (2000). Web services - the web's next revolution. *IBM developerWorks*.

Turner, K. J. (2005). Formalising web services. In *Proc. of Formal Techniques for Networked and Distributed Systems (FORTE 2005)*, volume LNCS 3731, pages 473–488, Taipei, Taiwan. Springer.

WSDL (2001). Web services description language (wsdl) 1.1. Technical report, World Wide Web Consortium. http://www.w3.org/TR/wsdl.