An Integrated Framework for Web Services Orchestration

C. Boutrous Saab, Université Paris Dauphine, France

D. Coulibaly, Université Paris Dauphine, France

S. Haddad, LSV, ENS Cachan, CNRS, France

T. Melliti, Université d'Evry, France

P. Moreaux, Université de Savoie, France

S. Rampacek, Université de Bourgogne, France

ABSTRACT

Currently, Web services give place to active research and this is due both to industrial and theoretical factors. On one hand, Web services are essential as the design model of applications dedicated to the electronic business. On the other hand, this model aims to become one of the major formalisms for the design of distributed and cooperative applications in an open environment (the Internet). In this article, the authors will focus on two features of Web services. The first one concerns the interaction problem: given the interaction protocol of a Web service described in BPEL, how to generate the appropriate client? Their approach is based on a formal semantics for BPEL via process algebra and yields an algorithm which decides whether such a client exists and synthetize the description of this client as a (timed) automaton. The second one concerns the design process of a service. They propose a method which proceeds by two successive refinements: first the service is described via UML, then refined in a BPEL model and finally enlarged with JAVA code using JCSWL, a new language that we introduce here. Their solutions are integrated in a service development framework that will be presented in a synthetic way.

BPEL, Formal Semantics, Interoperability, Language Expressivity, Program Synthesis

INTRODUCTION

At the hour of fusions, reorganizations of companies, Information Systems (IS) must have the capacity to take into account these economic constraints. What results is a need for flexibility, adaptability, opening and even for interoperability between remote and/or heterogeneous IS, i.e. based on different technical bases. Interoperability supposes that the applications are able to be located, to be identified, to expose the functionalities (services) which they offer, and finally, to exchange data. The Web services arise today as the most suitable

DOI: 10.4018/jwsr.2009071301

solution in order to connect remote IS, eventually heterogeneous ones.

Indeed, Service Oriented Architectures Services (SOA), initially based on the components and their capacity to communicate through their interfaces, quickly showed their limits in terms of weak coupling (necessary to meet the needs for flexibility and adaptability) and of interoperability. The Web services bring these properties to the SOA which were precisely lacking with the component based architectures. The Web services lie on standards for the information exchange as well as on protocols for their transport.

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 description, and service discovery - and specifications are being developed for each one: the "Simple Object Access Protocol" (SOAP) (Gudgin, 2000), which enables communication among Web Services, the "Universal Description, Discovery and Integration" (UDDI) (Bellwood, 2002), which is a registry of Web Services descriptions and the "Web Services Description Language" (WSDL) (Christensen, 2001), which provides a formal, computer-readable description of Web services. The latter describes such software components by an interface listing the collection of operations that are network accessible through standard XML messaging. This description contains 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 also require 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 of 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 (Alves, 2007). BPEL supports two different types of business processes (Juric, 2005). (i) *Executable processes* specify the exact details of business processes. They can be executed by an orchestration engine. (ii) *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 abstract 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. On the one hand, the specification of an executable process is close to the definition of a program and naturally yields the expressivity problem: how generic, rich and concise are the constructions of the language? On the other hand, the specification of an interaction protocol mainly raises the synthesis problem: how to synthesize a client which will correctly handle the interaction with the service?

The Expressivity Problem

Whereas BPEL is appropriate for composing Web services into business processes, it does not have all the features of a programming language like Java. Therefore, different works aim at extending BPEL by combining it with Java. The

two prominent approaches are *BPELJ for Java* (BPELJ) (Blow, 2004) and the *Web Services Invocation Framework* (WSIF) (Duftler, 2001). BPELJ provides the possibility to include Java code (which is called Java snippets) in BPEL process definitions. WSIF follows another idea: use the same syntax in BPEL to invoke any resource (or service) and describe it using WSDL even if it is a Java resource that does not communicate through SOAP (the applicative communication protocol of Web services). With both approaches, the design of a service is a two-step process: first one models the service with a BPEL program, then one refines it by developing Java code for the local treatments and exchanging values between the Java and the BPEL parts. However, due to the verbose and declarative style of BPEL, when the logic of the application is complex, this process leads to a program which is almost impossible to manage. In order to design a composite service whose application logics is the main complexity factor, we propose an alternative solution: enhancing Java with BPEL features. So, our first contribution is the definition of the language Java Complex Web Service Language (JCWSL). Furthermore, starting from a JCSWL program, we automatically produce both the associated BPEL interaction protocol and a code which can be invoked through a Web server.

The Synthesis Problem

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. Thus, we have addressed this problem and proposed a solution based on a formal semantics (Melliti, 2003; Haddad, 2004).

First, we specify what an external behaviour is, i.e. we give an operational semantics to an abstract BPEL specification in terms of a timed 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, we define 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 our needs. Thus, we introduce 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, we focus on the synthesis of a client which is in an interaction relation with the transition system corresponding to the system. The client we 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 we have developed either detects the ambiguity of the Web service or generates a deterministic automaton satisfying the interaction relation. The core of our algorithm is a kind of determinisation of the transition system of the service.

This article is organized as follows. "Web Services" section points out the principles of the Web services and details the BPEL constructions. Then, in the "client synthesis" section, we propose a formal semantics of this language, we define the interoperability between a client and a service and we describe the client synthesis algorithm. We introduce JCSWL, our extension of BPEL in section "JCWSL: a design language for Web services". In the "Web service design

framework"section, we give an overview of the architecture of our environment. Finally, in "related work" section, we review related work before concluding.

WEB SERVICES

The Web services are an instantiation model of software architecture called service oriented architecture. We first describe the intrinsic characteristics of this model. Then, we detail the Web services principles while insisting on the dynamic aspects. Finally, we present BPEL and its major syntactic constructions, the design language of Web services Web (the current standard).

Service Oriented Architecture Characteristics

A service oriented architecture consists in structuring an application, an applicative block even an Information System (IS), in contracted services in order to answer the following stakes: (i) the implementation of global services between applicative blocksby an interoperability policy ; (ii) the seek of re-using inside an applicative block or an application, in particular on the service infrastructure level or the business service unit, by a re-use policy.

The re-use takes all its importance especially at two levels. (1) On the service infrastructure level, which are without a business value but that each application must inevitably implement for its own needs (safety, exchanges, etc). (2) On the fine granularity business service level, that it is of the re-use of software components within an application (for example, a software layer of common services for the "batch" chains) or about the invocation of a transverse Web Service of "adress validation" type. The re-use of such services avoids the duplication of code between the applications or the modules of an application. On the other hand, for the business service coarse grain, which exposes the value of an information system outside its borders, the stake relates more to the capacity of these services to interoperate with other IS blocks being its clients.

A service oriented architecture must guarantee technological neutrality, weak coupling, transparency with respect to their accessibility, message orientation, composition and autonomy. Today, Web services are the best technological solution adapted to answer these objectives.

From Service Oriented Applications to Web Services

Web service is a specific type of service identified by a URI and showing the following characteristics: (i) it exposes its functionalities through Internet using standards and Internet protocols ; (ii) it is implemented via a self-descriptive interface based on an Internet standard.

The concept of Web Service is currently articulated around the following three acronyms: (i) SOAP ("Simple Object Access Protocol") is an exchange protocol between independent applications of any platform, based on the XML language ; (ii) WSDL ("Web Services Description Language") gives a Web service description in a XML format while specifying the called methods, their signature and the access point (URL, port, etc) ; (iii) UDDI ("Universal Description, Discovery and Integration") standardizes a solution of distributed Web service directory, allowing simultaneously the publication and the exploration. UDDI behaves as a Web service where its methods are called via the SOAP protocol. The Web service provider publishes its service interface in WSDL format. The client searches the Web service according to a set of characteristics defined by the UDDI directory. The directory finds the service and sends the localization of the server hosting the service. The client asks the server for the suggested contract of the service. The server replies with a call format in WSDL. The client invokes the service with a SOAP message and the server replies by providing a result.

Current standardization around Web services is a vast work that aims to define a true distributed infrastructure, which is able to satisfy

the application's needs, as well as in term of exchange standardization as in term of transverse services. These standards were specified by an organization gathering the industrialists, major actors of the market: the WS-I (http://www.ws-i.org/).

BPEL

BPEL primarily lies on WSDL to deal with the information relating to the localization of the service and the data format of the messages. Here, we limit the presentation to the business process components while breaking them up into two catégories: elementary operations and constructors. We do not take into account the exact (verbose) BPEL syntax in this part. BPEL makes it possible to handle variables and types. Notice that we mask some important features of BPEL as the compensation mechanism which do not have significant impact on the interoperability.

Elementary Operations

- the receive primitive *receive* corresponds the reception of a message (coming from a client or another service). BPEL offers the possibility of synchronous or asynchronous communication. We chose to study only the asynchronous mode because it is easy to simulate a synchronous communication using this mode, whereas opposite simulation requires the creation of some auxiliary processes. Moreover, in the composite service framework, the asynchronous mode is more suitable for the long transactions and the weak coupling.
- the primitive *invoke* relates to the sending of a message (to a client or another service). Contrary to the reception of a message which suspends the service if the required message is not present in the buffer, the process continues its activity after the emission.
- the primitive *raiseproc* makes it possible for a process to give up an execution context while signaling a fault. The corresponding fault is treated by the current context or is transmitted to the global context. If the fault is not cached by any context, it causes the end of the process.
- the primitive *terminate* makes it possible for a process to stop the service. One can compare it to an untreated fault; we will study it only in the case of a process which completes normally its treatment.

The introduced constructors can be seen as processes built from sub-processes and elementary operations.

Constructors

- Process *empty* (as its name indicates it) does nothing. It is introduced into BPEL to specify the lack of treatment in some branches of a conditional execution.
- Process *sequence* executes sequentially the corresponding processes.
- Process *switch* consists of sub-processes where their execution is conditioned by a Boolean expression of internal variables (thus unknown of the client) and executes the first branch of which the condition is satisfied. In an optional way, the last branch can not be conditioned by a test.
- Process *while* carries out repeatedly a sub-process as long as a Boolean expression (similar to those appearing in *switch-proc*) is satisfied.
- Process *scope* defines an execution context of a sub-process in the following way. This process can be given up on reception of a message whose type belongs to a given set, on reaching a deadline which begins with context activation or on a fault signal. In all cases, a process is linked to each one of these events.
- Process *pick* waits for a message whose type belongs to a given set in order to execute a process corresponding to each message type. In an optional way, a delay can control this waiting. It is easy to simulate a *pickproc* process using a

contextproc process. Again we will not study it explicitly.

- Process *flow* begins a parallel execution of the corresponding processes and finishes when all these processes finish. These processes can be synchronized in the execution. Since this synchronization is not perceived by the client of the service, we will not model it. It is obviously a simplification which we will remove in a future work.

CLIENT SYNTHESIS

One of the contributions of the Web services is that potential clients discover, *at the time of invocation*, the service specification. However, that raises for the client the problem of leading, in a correct way, the interaction with the service given the flow of exchanged messages. In this section, we develop a formal approach which provides a client synthesis, *i.e.*, an implementable description of an entity interacting with the Web service defined with BPEL (named "the service" in the sequel). The method breaks up into three levels.

We initially define the formal semantics of a BPEL (abstract) process. Since time is explicit in the BPEL language, this semantics is given by a (dense) Timed Automaton (TA). Due to the BPEL language itself, we use a Process Algebra like approach to build the TA associated with every BPEL process.

We then define a suitable interaction relation between a service and its potential client. Since this relation must take into account the various timed behaviours of the processes, it should be based on the timed executions of the service and its client. So, we define the interaction relation in the context of Timed (labelled) Transition Systems (TTS), the standard semantics of TA.

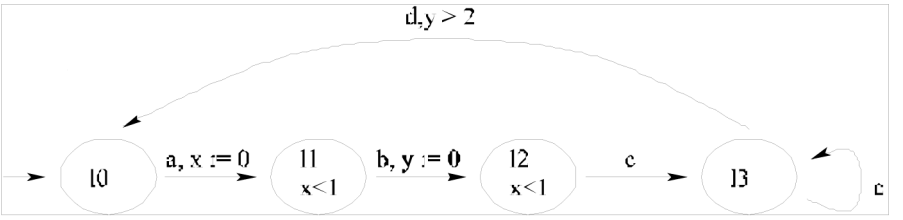
Finally, and this the most difficult step, we build when this is feasible, a deterministic client TA, by means of a specific synthesis algorithm. This TA interacts with the server automaton underlying the service process according to the interaction relation.

A Semantic Approach for Web Service Behaviour

To be able to generate and run a dynamically built composed service from Web services described with BPEL, it is first necessary to give a precise semantics to BPEL. BPEL provides a set of operators describing in a modular way the observable behaviour of an abstract process. As shown in (Staab, 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, 1984). A Process Algebra is a model of behaviours of active entities called processes. Each process may execute elementary actions belonging to a given set. Processes may also be combined by means of a set of operators, like sequencing, parallel execution, choice between several processes, etc. With a set of elementary actions and a set of operators, we can describe more complex processes, and new operators as we do in the mathematical framework of algebraic structures (groups, rings, etc.). Since time is explicitly present in some of the BPEL constructors (the operators of a Process Algebra), we must extend the standard Process Algebra semantics with time. Several models have been defined for Timed Process Algebras (see (Nicollin, 1991) for underlying problems), like (Baeten, 1991) for timed ACP, (Nicollin, 1994) for ATP, (Schneider, 1995; Leonard, 1997) for timed CSP and timed LOTOS. However, as we try to *construct* the timed model of an adapted executable client of a BPEL service, we associate directly a Timed Automaton with a BPEL process.

So, we define the semantics of a BPEL process in two steps: first we review how the constructors of the BPEL language can evolve (operational rules) when executing actions; then we construct a TA for each BPEL process.

Figure 1. A timed automaton with two clocks



Timed Automata

A Timed Automaton is an automaton extended with time specification. Time is introduced through *clocks*, invariants, guards and clock reset(see Figure 1). Let us denote by *C(X)* the set of constraints over a set X of variables, conjunctions of elementary constraints $x \trianglelefteq c$ with $x \in X$, $c \in Q_{\geq 0}$ and $\trianglelefteq \in \{\leq, <, >, \geq\}$.

Definition 3.1 (Timed Automaton (Alur, 1994)) *A Timed Automaton (TA) is a tuple* $(L, A_\tau, X, E, I, L_0, L_f)$ *where:*

- L *is the set of* locations;
- $A_\tau = A\{\tau\}$ *is the set of* actions and $\tau \notin A$ *is the* silent action;
- X *is the set of positive real-valued* clocks;
- $E \subseteq L \times C(X) \times A_\tau \times P(X) \times L$ *is the set of* edges: *an edge e is a tuple (s,g,a,R,d) with s the source location, g a guard, a an action, R a subset of clocks to reset and d the destination location*.
- $I \in C(X)^L$ *assigns an* invariant *to any location. Elementary constraint operators in invariants are restricted to* < and ≤.
- L_0 *is the set of* initial locations.

An execution of a TA is a sequence of transitions and time elapsing between states of the TA. A state is composed of a location of the automaton and a value *v(x)* per clock x (called the clock valuation). Transitions are either continuous (time passing) or discrete (actions from A_τ). A continuous transition is $(l,v) \xrightarrow{t} (l',v')$ with $l' = l$ and $v' = v + t$ (where this sum means that t is added to the valuation of every clock) provided that $\forall 0 \leq t' \leq t$, $I(l)(v + t') = true$ (l invariant remains true). A discrete transition is $(l,v) \xrightarrow{a} \{a\} (l',v')$ iff $\exists (l,g,a,R,l') \in E$ such that $g(v)=true$ (guard), $v'(x)=v(x)$ $x \notin R$, and $v'(x)=0$ for $x \in R$ (clocks reset). For instance, a possible execution of the TA given in Figure 2 is: $(l_0, 0, 0) \xrightarrow{1,2} (l_0, 1.2, 1.2) \xrightarrow{a} (l_1, 0, 1.2) \xrightarrow{0.7} (l_1, 0.7, 1.9) \xrightarrow{b} (l_2, 0.7, 0) \xrightarrow{0.1} (l_2, 0.8, 0.1) \xrightarrow{0.1} (l_2, 0.9, 0.2)$.

Let us explain how we build the TA of a BPEL process P. Locations of the TA are exactly the processes derived from P: these processes correspond to the possible evolutions of P when executing BPEL operators. Evolutions are given by a set of so called semantics rules, in the line of Structured Operational Semantics (Plotkin, 1981) for Process Algebras. A rule defines partially a set of edges in the TA. Actions of the TA model the activities of the processes. Each edge is complemented with timing annotations (guard and clocks to reset). Finally, invariants are added to locations based on the time semantics of the BPEL operators (like *scope*). We successively describe actions of the TA, rules of the BPEL language, handling of the clocks and definition of the guards and invariants of a BPEL process. As usual, we denote by 0 the "process" which does nothing, corresponding to the final state of a previously active process.

Actions

From the definition of BPEL, four kinds of actions are possible:

- Immediate actions correspond to logical actions such as selection of an alternative or throwing an exception. They are not visible to the client and are denoted by τ, the silent action. We denote by E_x the set of exceptions which appear in the semantic rules.
- Expiration of a delay is denoted by *to*.
- Reception and sending of messages are basic Web service interactions. Note that we do not model the timing of message transfers. The set of message types is noted by *M*. A sending is denoted by *!m* and a reception by *?m* with $m \in M$. We also introduce $!M = \{!m | m \in M\}$ and $?M = \{?m | m \in M\}$. Finally, the generic character * represents either ! or ?.
- In order to check that the client detects the end of the service, we introduce √, the termination event. This action simplifies the definition of some rules.

Rules

For each constructor *op* of BPEL involving processes P_i for $i \in I$, a rule describes the possible transformations of the process $P = op(P_1, P_2, ...)$ according to the actions executed by the processes P_i. A generic rule, presented in a standard form has the following structure ($\{u_i\}$ stands for $\{u_i \mid i \in I\}$):

$$op: \frac{B(\{P_i \xrightarrow{\alpha_i} P_i'\})}{P \xrightarrow{L(\{\alpha_i\})} N(P, \{P_i'\})} \text{ if } G(\{\alpha_i\})$$

The components of a rule are:

- a Boolean expression relative to the potential transitions of some components of *P*: $B(\{P_i \xrightarrow{\alpha_i} P_i'\})$;
- this condition is supplemented by a second condition, called the guard of the rule, on the labels appearing in the transitions, denoted by $G(\{\alpha_i\})$.
- If the two conditions are met, then a transition is possible for *P* whose label $L(\{\alpha_i\})$ is an expression depending on the transition labels of the subprocesses. If there is no *B* nor *G* expression, the upper part and the line in the rule are omitted.
- the new process is an expression $N(P, \{P'_i\})$ which depends of the running process and the new subprocesses.

Note that if the processes P_i and *op* do not involve immediate actions, time passing is not explicitly modelled in the rules.We present below rules corresponding to each BPEL constructor, beginning with the elementary processes *empty*, ?o{m}, !o{m} and *throw*. Observe also that since the invoke process calls a subprocess operation, corresponding message exchanges are "silent" actions from the point of view of the client; hence we do not need to model this constructor in our perspective of server-client interaction.

The empty *process*: empty is a basic element, which can only terminate; hence, it is the last action that a process can execute:

$$\text{empty} \xrightarrow{\surd} 0.$$

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} (the notification operation of WSDL), consists in sending a message of type *m*:

$$*o\{m\} \xrightarrow{*m} \text{empty } * \in \{?, !\}.$$

The throw process: the throw process r[e] simply raises an exception e which must be handled in some way (see below the scope process):

$$\forall\, e \in E,\ r[e] \xrightarrow{e} 0.$$

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 as

long as this process does not indicate its termination. In the latter case, the sequance process becomes the second process in a silent way:

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

$$\frac{P \xrightarrow{\surd} \text{and } Q \xrightarrow{a} Q'}{P;Q \xrightarrow{a} Q'}$$

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

The switch process: the process switch[$\{P_i \mid i \in I\}$] 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:

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

The while process: the process while[P] iterates an inner process as long as an *internal* condition is satisfied. Like switch, while evaluates in a silent way its condition (because it's an internal choice of the process, we do not know what appends exactly). Thus we have two rules depending on this internal evaluation.

$$\text{while}[P] \xrightarrow{\tau} P \text{ ; while}[P] \text{ and while}[P] \xrightarrow{\tau} \text{empty}$$

The flow process: the process flow[$\{P_i \mid i \in I\}$] simultaneously activates a set of processes $\{P_i\}$. In the present work, we do not model synchronization primitives associated with flow introduced in BPEL4WS and not defined in XLANG.

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.

- Individual actions:

Immediate actions of any process P_i occurs without delay, and the flow process is maintained between new subprocesses:

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

Message exchanges are proceeded by the processes P_i and the flow process is maintained between new subprocesses:

$$\frac{\forall\, m \in M_x \quad \exists\, j \in I, P_j \xrightarrow{*m} P_j' \text{ and } \forall i \neq j, \forall a \in E_x \bigcup \{\tau\}, \text{not} \exists\, i \in I, (P_i \xrightarrow{a})}{flow[\{P_i \mid i \in I\}] \xrightarrow{*m} flow[\{P_i \mid i \in I \setminus \{j\}\} \bigcup \{P_j'\}]}$$

- Common timeout

The rule about *to* describes the case where a subset J of processes execute simultaneously a *to* action:

$$\frac{\exists\, j \subseteq I, \forall j \in J, P_j \xrightarrow{to} P_j'}{flow[\{P_i \mid i \in I\}] \xrightarrow{a} flow[\{P_i \mid i \in I \setminus J\} \bigcup \{P_j' \mid j \in J\}]}$$

- Common termination

When all processes terminate, the flow process becomes the null process:

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

The scope process: scope(P,E) with

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

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 define $M_I=\{m_i \mid i \in I\}$ and $E_J=\{e_j \mid j \in J\}$.

- P actions

If P ends then scope also ends. If P may execute an action a (but not the termination nor an exception handling nor a m_i message receiving) then scope executes a:

$$\frac{P \xrightarrow{\surd}}{scope(P,E) \xrightarrow{\surd} 0}$$

and $\forall a \notin \{\pi\} \cup E_x \cup M_I$

$$\frac{P \xrightarrow{a} P'}{scope(P,E) \xrightarrow{a} scope(P',E)},$$

- Receiving a message m_i

If a m_i message is received, then scope(P,E) becomes P_i:

$$\forall i \in I, \cup \frac{\forall a \in E_x \cup \{\tau, \surd\},\ \text{not}\ (P \xrightarrow{a})}{scope(P,E) \xrightarrow{?m_i} P_i}$$

- Timeout occurrence

If P does not end, nor handle an exception nor execute a silent action, then the process scope may ends with a timeout:

$$\frac{\forall a \in \{\tau,\pi\} \cup E_x,\quad \text{not}\ (P \xrightarrow{a})}{scope(P,E) \xrightarrow{to} Q}$$

- Exception handling

Expected exceptions e_j lead to associated processes R_j whereas other exceptions are transmitted to the upper level. This last derivation allows detection of an exception e never cached at any level including the topmost one, which is an erroneous service definition:

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

and

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

The pick process: pick is a special case of the scope process: a scope with a main process P being empty:

pick[E] = scope(empty,E).

Locations of the Timed Automaton

Initially, there is only one location (the initial location) corresponding to the studied process. After the construction of a new edge from the automaton, using a semantic rule, a new process is computed. If this process does not already label a location of the automaton, a new location is created. Because of the definition of the semantic rules described above, the number of derived processes is finite (and consequently the number of the automaton locations). Each location is completed with an invariant (see below).

Clocks, Guards and Invariants of the Timed Automaton

We associate a clock with each subprocess scope of the process and a particular clock (x_{im}) to manage immediate actions. Given a process, we determine by a downward analysis which clocks are active, *i.e.* which subprocesses scope are in the course of execution.

The invariant of a location depends on the possibility of an immediate action. In such a case, the invariant is x_{im} =0; if this is not the case, the invariant is the conjunction of elementary conditions $x \leq d$ with x, an active clock and d the time defined in the subprocess corresponding to x.

For a given edge, the clocks to reset are the inactive clocks of the source process which

become active in the destination process. x_{im} is always reset.

From each location which contains active clocks, we apply the common timeout rule to every subset of active clocks which may reach their temporal bound, giving a set of edges. For such an edge, the guard specifies that the clocks of the subset reached their limit while the other active clocks did not. Edges outgoing from locations without active clocks have a *true* guard.

Construction of the Timed Automaton

The algorithm building the TA of a BPEL process can be summarized as follows:

- It maintains a set of processes to be examined and a partially built automaton. It starts with the process and an automaton reduced to only one location.
- When analyzing a process, it initially builds the edges corresponding to the semantic rules and it inserts each destination process, not yet present in the automaton, in the set of the processes to be examined.
- Then, it determines the set of active clocks of the process. From this information and already built edges, it deduces the location invariant. Finally, it generates the timeout edges.
- The Computation of the subset of clocks to reset on an edge is carried out either at the construction time of the edge if the destination process was already examined, or during the (later) examination of this process.

Example of the Timed Automaton of a BPEL Process

Let us apply our algorithm to the process

?Start;scope(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 the 0 (null) process. Our algorithm builds the TA of Figure 3.

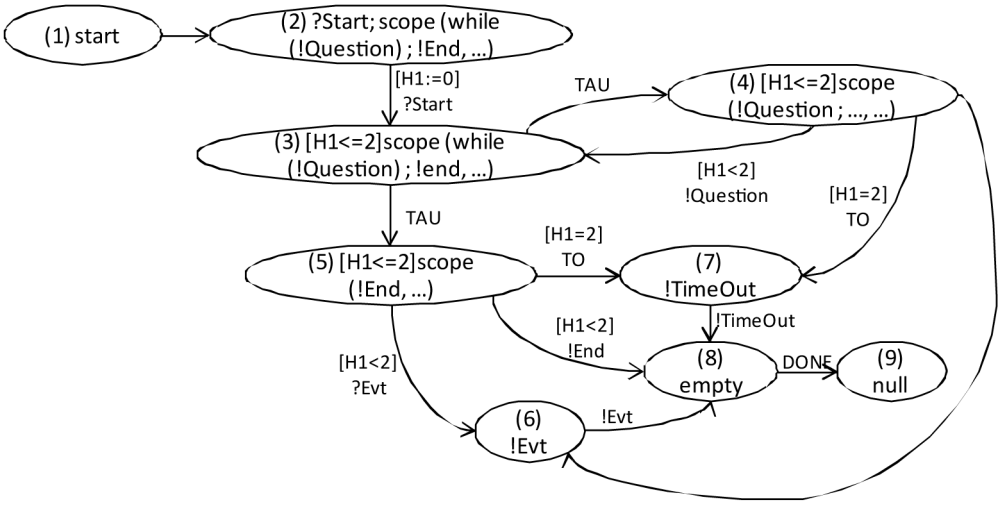
Client-Service Interaction Relation

Clearly, the built TA is a compact representation of the observable behaviour of a BPEL abstract process. However, since we try to construct a model of a client adapted to a given service we must study the interaction between these processes at the level of their timed executions. Timed executions define exactly the semantics of a TA. The formal model of the set of these executions and their relations is a Timed (labelled) Transition System (TTS). A TTS is a tuple $(S, S_0, A, \rightarrow)$ with S the set of states, $S_0 \subseteq$ S the set of initial states, A a finite set of actions and $\rightarrow \subseteq$ S $\times$ (A $\cup \mathbb{R}_{\geq 0}$) $\times$ S the set of transitions. We also write $q \xrightarrow{e}$ q' if $(q, e, q') \in \rightarrow$. A transition $q \xrightarrow{t} q'$ with $t \in \mathbb{R}_{\geq 0}$ corresponds to t units time passing. The states of a TTS associated with a TA are naturally its states (pairs (l,v)) and the transitions of the TTS are either the discrete transitions ($e \in A_\tau$) or time passing in a location.

First of all, we describe, in an informal way, what should be a correct interaction between two TTS. It is defined as a relation between states of the TTS, like the classical bisimulation relation between Labelled Transition Systems (LTS). Obviously, pairs of initial states must belong to the relation.

Moreover, a pair of related states must have a coherent vision of the forthcoming interaction. This implies that the relation must take into account mutually observable transitions only, *i.e.*, discarding τ actions. Hence, we define the observable transitions of a TTS by $s \overset{a}{\Rightarrow} s'$ iff $s \xrightarrow{\tau^* a \tau^*}$ s', $s \overset{\varepsilon}{\Rightarrow}$ s' iff $s \xrightarrow{\tau^*} s'$ and $s \overset{d}{\Rightarrow} s'$ iff $s \xrightarrow{d_1 \tau \ldots \tau d_n} s'$ with $\sum d_i = d$.

Figure 2. TA of process ?Start;scope(while[!Question];!End,[{(?Evt, !Evt)}, (H1:2, !TimeOut), {}]))



Then, we could require that (in a similar way to bisimulation), when a state *s* of a pair (*s*,*s'*) can evolve by an observable transition to a state s_1, *s'* should have a transition with the same label leading to a state s'_1 composing with s_1 another pair of consistent states.

However, we must be careful. First, if a TTS sends a message then the other TTS must be able to receive it. So, it is necessary to introduce the concept 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 be carried out by complementary actions.

But such a definition is too strong because it does not distinguish between the different nature of sending and receiving messages: sending a message is an action whereas receiving a message is a *reaction* and cannot spontaneously occur. Consequently, in a more suitable way, the interaction relation requires that, if in a state *s* of a pair (*s*,*s'*), a TTS can receive a message *m*, then (1) there is a state *s"* of the other TTS not distinguishable of *s'* from the observable transitions point of view which can send *m* and, (2) in *s'* the other TTS can send a message (possibly different from *m*). The first condition reflects the fact that the first TTS is not over-specified while the second implies that it will not indefinitely wait a message.

These considerations lead to the following definition.

Definition 2 *(Interaction relation) Let* $T_1 = (S, \{s_{01}\}, A, \rightarrow_1)$ *and* $T_2 = (S, \{s_{02}\}, A, \rightarrow_2)$ *be two TTS. Then* T_1 *and* T_2 *interact correctly iff* $\exists \sim \subseteq S_1 \times S_2$ *such that* $s_{01} \sim s_{02}$ *and* $\forall\ (s_1, s_2)$ *such that* $s_1 \sim s_2$*:*

- Let $a \notin \{?m \mid m \in M\}$:

o *if* $\exists\ s_1 \overset{\alpha}{\Rightarrow}_1 s'_1$, *then* $\exists\ s_2 \overset{\bar{\alpha}}{\Rightarrow}_2 s'_2$ *with* $s'_1 \sim s'_2$ *and*

o *if* $\exists\ s_2 \overset{\alpha}{\Rightarrow}_2 s'_2$, *then* $\exists\ s_1 \overset{\bar{\alpha}}{\Rightarrow}_1 s'_1$ *with* $s'_1 \sim s'_2$

- Let $m \in M$: *if* $s_1 \overset{?\mu}{\Rightarrow}_1 s'_1$ *then*

o $\exists\ s_2 \overset{\omega}{\Rightarrow}_2 s_2$, $\exists\ s_2 \overset{\omega}{\Rightarrow}_2 s_2^+$, $\exists s_2^+ \overset{!\mu}{\Rightarrow}_2 s'_2$ *with* $s_1 \sim s_2^+$ *and* $s'_1 \sim s'_2$ where *w* is a word on $A\backslash\{\tau\}$;

o $\exists s_2 \overset{!\mu'}{\Rightarrow}_2 s'_2$

- Let $m \in M$: *if* $s_2 \overset{?\mu}{\Rightarrow}_2 s_2'$ *then*

o $\exists\, s_{\bar{1}} \overset{\omega}{\Rightarrow}_1 s_1$, $\exists\, s_{\bar{1}} \overset{\omega}{\Rightarrow}_1 s_1^+$, $\exists s_1^+ \overset{!\mu}{\Rightarrow}_1 s_1'$ *with* $s_1^+ \sim s_2$ *and* $s_1 \sim s_2$ *with* where w is a word on $A\backslash\{\tau\}$;

o $\exists s_{\bar{1}} \overset{!\mu'}{\Rightarrow}_1 s_1'$

Synthesis Algorithm

We are now in position to present the synthesis algorithm of the client. First of all, the client must be implementable, which means that its behaviour must be deterministic. In addition, since it must take into account the *clocks* of the service, and has to interact with the service as explained above, its behaviour must be expressed with a TTS. This leads us to construct a client's model as a deterministic TA having an interaction relation with the service automaton.

Before describing this algorithm, let us notice that some BPEL processes do not admit a client able to correctly interact with them. For example, the process switch[?*o*[*m*],?*o*[*m'*]] chooses, in an internal way, to receive either a message *m* or a message *m'*. Hence a deterministic client must send either *m*, or *m'*. However, once its choice is carried out, the service waits only for one of the two messages. In other words, the corresponding two states of the TTS cannot be in an interaction relation. Note that, in contrast, the process switch[!*o*[*m*],!*o*[*m'*]] admits a deterministic interacting client which waits for either the message *m* or the message *m'*. The same problem arises with the service process *while*{!*a*} since a client does not know how many messages to receive before leaving the loop. Clocks are another cause of ambiguity (temporal ambiguity) which will be explained latter. We say that a process is *ambiguous* if it does not admit a deterministic TA which is in interaction relation with him.

The general approach of our algorithm is similar to a determinization procedure: a location of a client TA corresponds to a subset of locations of the service TA linked with edges labelled with τ (Figure 4). However, determinization of TA is known to be undecidable in the general case (Alur, 1994). So, similarly to approaches which determinize subclasses of TAb(Alur, 1999), our algorithm seeks a TA with the same clocks only as the service TA.

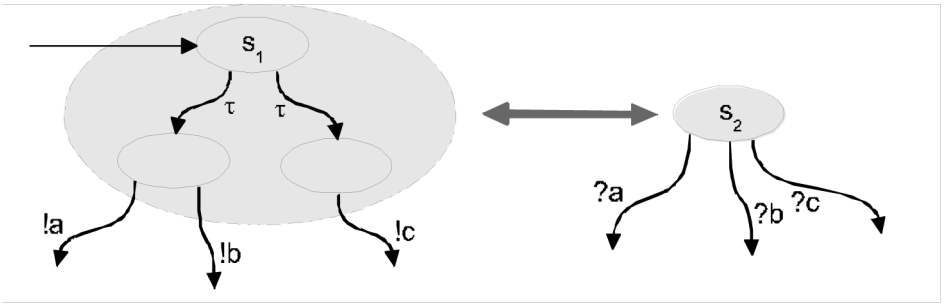
The algorithm builds the client automaton A_C, processing potential client locations it has previously defined and pushed on a stack. Building of A_C stops either when an ambiguity is detected (returning "fail") or when the stack is empty, returning the client automaton. Let us detail the algorithm for a given client potential location l_C.

A Step of the Algorithm

Let $L_S(l_C)$ be the set of service locations temporary associated with l_C (the initial location $l_{0,C}$ is associated with the initial location of the service automaton A_S. We have the following steps:

- Creation of a new client location: we compute the ε-closure of $L_S(l_C)$, *i.e.* all reachable locations from $L_S(l_C)$ by sequences of τ L_S-actions. If this subset (say $L'_S(l_C)$) is already associated with a location l'_C of the client, then edges of the client TA which have generated l_C (*i.e* with destination l_C) are redirected to l'_C and we are done with the step. Otherwise, a new client location is created.
- Check for temporal ambiguity: we compute the subset $L''_S(l_C)$ of $L'_S(l_C)$ with no τ labelled outgoing edge and we check if all locations of $L''_S(l_C)$ have the same set of clocks. If this is not the case we say that the service is temporally ambiguous and the algorithm returns fail.
- Check for interaction relation: we verify that the interaction between l_S and l_C is satisfied (see below). The algorithm returns fail if this is not true.
- Creation of new edges and new potential client locations: each outgoing edge of $L'_S(l_C)$ not labelled with τ (visible service actions) gives rise to an outgoing edge from l_C (the associated client action)

Figure 3. Subset of service locations (left) -- associated client location (right)



labelled with its complementary action. Destination locations of these edges provide new potential client locations, pushed on the stack.
- Guards and invariants definition: we copy to A_C, clock guards of the edges and clock invariants of the locations of A_S.

Interaction Verification

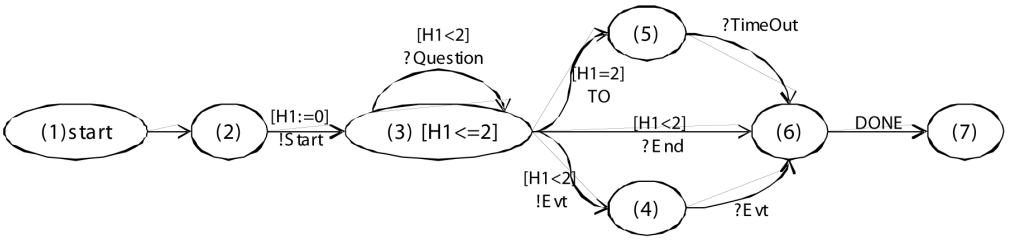
To verify the interaction relation, we compute and then analyze the Terminal Strongly Connected Components (TSCC) of the server locations set $L'_S(l_C)$ with respect to τ actions. Client and server interact correctly iff the following properties hold:

- If one of the TSCC does not send any message, then none of the server locations can send a message,
- otherwise, each TSCC must send a message. The set of messages sent by the server locations is the set of messages sent by all the TSCC.
- The set of messages received by the server locations must exactly be the same as the set of messages received by any of the TSCC.
- If any TSCC may execute √ (termination) then any TSCC must also be able to perform it, and so the server locations set can do it.

Example

Figure 5 gives the synthetis client TA of the server TA ?*Start*; scope (while [!*Question*] ; !*End*, [{(?*Evt*, !*Evt*)}, (*H*1:2, !*TimeOut*), {}]) shown in Figure 3. Observe that some locations are "merged" in the client TA. Location labelled (2) in the server is present in the client with the same label (2) but its outgoing edge receives the message *Start*, whereas the client sends this message. Location (3) in the client is more involved. The ε-closure $L'_S(3)$ of $L_S(3)$ is the set {(3),(4),(5)} of server locations. Although these

Figure 4. Client timed automaton for the server timed automaton of Figure 3



locations are important in the server behaviour, the client could not know whether the server is in location (3), (4) or (5). Hence, the algorithm "merges" these three locations in the client location (3). The different outgoing edges of $L'_s(3)$ are adapted to the client merging. Other locations are generated in nearly the same way as the client location (2).

Dense Versus Discrete Time

We have chosen a dense time model for our formal semantics of BPEL process behaviours. In a previous work (Haddad, 1994), the discrete time semantics was preferred for simplicity reasons.

The discrete time approach has the following drawbacks. First, the passing of a unit of time is modelled by an explicit transition (χ) 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.

Conversely, the derivation of the client TA from the operational rules is more intricate than in the discrete time context 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 edges. Moreover, our algorithm tries to synthetize a client TA with the same clocks as the server TA. This restriction, due to the fact that non deterministic TA are strictly more expressive than the deterministic ones may lead to false ambiguity detection. For instance, consider the process (Figure 6) switch: one branch of this switch starts with a scope process and another branch does not activate a timing constraint. Our algorithm detects this process as an ambiguous service although it may be that a client TA (with a different set of clocks) exists: in one branch there is an active clock whereas in the other one there is none and the client cannot decide which edge to follow. In a discrete time framework, the previous (complete) method produces a client. We implicitly work at a (discrete) TTS level. In the dense time framework, we work at a higher level (the TA one). This incompleteness of the algorithm is the price to pay in order to obtain a more compact representation of the client but we consider that this restriction on the clocks is reasonable because such false temporal ambiguity detections correspond to unrealistic BPEL processes.

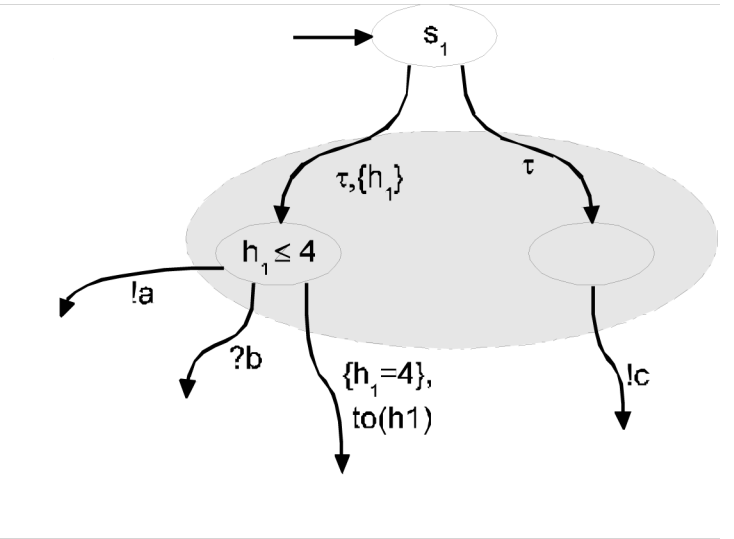
JCWSL: A DESIGN LANGUAGE FOR WEB SERVICES

JAVA Complex Web Service Language (JCWSL) is a *JAVA* script language extended with the BPEL4WS constructors. The main goal is to design the CWS using the *JAVA* language and to define, in the same time, the service behaviour using BPEL4WS constructors. The choice of *JAVA* is essentially due to its popularity in the Web services community. The BPEL constructors are defined and implemented with respect to the semantics defined by the description language.

JCWSL offers the following properties:

- **High expressivity level**: It supports *JAVA* language, and therefore offers a high level of expressivity in order to define and design a complex Web service.
- **Transparency**: Local and/or distant Web services orchestration is integrated in the language in a transparent way.
- **Flexibility**: It offers two types of operation's invocation: visible or invisible invocation. A visible invocation appears in the service behaviour, while an invisible one executes the invocation without appearing in the service behaviour.
- **Strong coupling**: JCWSL extends in an

Figure 5. False ambiguity detection for process switch(!o[c],scope(!o[a],[{(b,empty)},(4,empty),{}])



elegant and natural way the *JAVA* language. Thus, the operations and messages manipulation are visible and accessible in both parts of the program *i.e.*the implementation (*JAVA*) and the behaviour (BPEL) parts.

UML Modeling

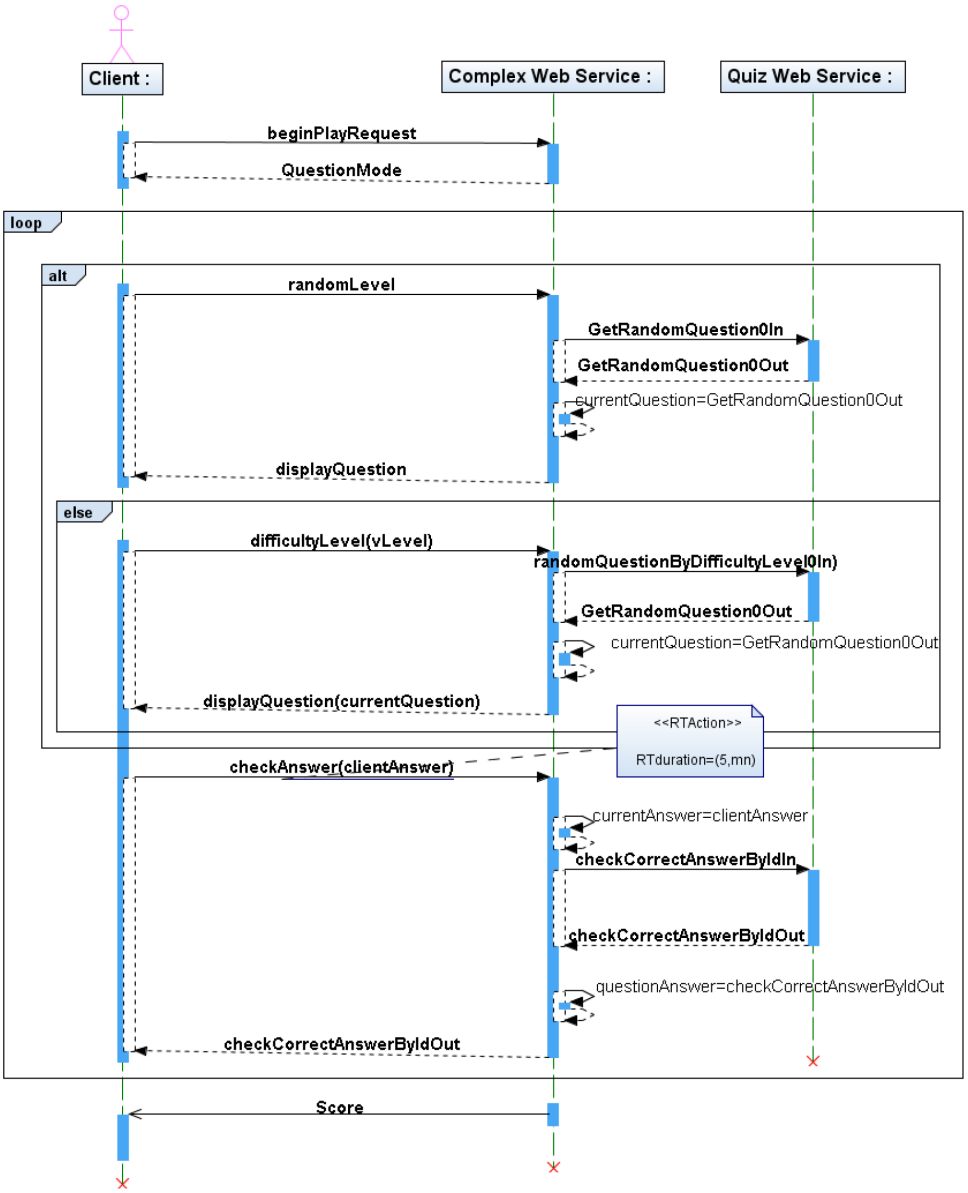
The Unified Model Language (UML) is widely used in the development of object oriented software and has also been used for business process modelling and system design. In the literature, there is different works that propose to map UML modelling to BPEL (Baresi, 2003; Skogan, 2004; Gronmo, 2006; Gardner, 2003; Cambronero 2007}. Among all this papers, Cambronero 2007 seems to be, in our sense, the most exhaustive since it proposes different stereotypes and covers the main orchestration constructors essentially those concerning time constraints. In the sequel, we use their profile in order to model our example. Due to space reasons, we restrict our UML modeling of our example to the sequence diagram.

Example

We present, in this section, the development in JCWSL of a CWS (*Example 1*) implementing an advanced quiz game based on an existing BWS. The BWS, located at www.hlrs.de/quiz/quiz.wsdl, implements a simple quiz game consisting of the invocation of a set of questions with different difficulty levels, and checking the correctness of the answer. This example presents a new game where the goal is to answer a fixed number of questions within a fixed delay. The difficulty level of any question can be chosen randomly or specified by the player. Here are the different steps of the application:

- The player requests to start the game by specifying the number of questions (*nbQuestion*).
- The player asks for a question with a given difficulty level or a random one.
- The player has to answer *nbQuestion* questions.
- The player must answer within a fixed delay otherwise the service skip to the next question.

Figure 6. The sequence diagram of the composite quiz service



- At the end, the final score, computed according to the number of correct answers and their difficulty level, is sent to the player.

The Sequence diagram of the example is presented in the Figure 7. Note that we use only The <<RTAction>> stereotype form (Cambronero 2007) to model the time out on the player answer.

The CWS uses the BWS cited earlier to implement this game. The BWS implements the following operations:

- ***randomQuestion*** returns a question of any level;
- ***randomQuestionByDifficulty*** returns a question of a specified level;
- ***checkCorrectAnswer*** checks the correctness of a user answer for a given question.

Thus, the CWS includes in the *import block*, the importation of the BWS to be able to invoke its operations. It also needs to include the input/output *JAVA* package for displaying purposes (lines 3 and 4).

After the *import block*, the *definition block* includes a declaration block and a main block. In the declaration block, operations and messages used later must be declared here. Here, five types of messages are defined (lines 9 to 30):

- *Answer*: is the answer on a given question and it includes the question and its answer;
- *Level*: defines the difficulty level of a question;
- *ChoiceMode*: represents a notification message;
- *Questions*: represents the number of questions;
- *Score*: represents the final score;

and the following operations:

- *beginPlay:* is an input output operation which allows to begin the game;
- *randomLevel*: is an input operation which allows player to choose random level;
- *checkAnswer*: is an input operation which receives the answer on a specified question;
- *difficultyLevel*: is an input operation which allows player to choose a difficulty level;
- *getScore*: is an input operation which allows to get the score;
- *finalScore*: is an output operation which sends the final score;
- *displayQuestion*: is an input operation which allows to display a question;

Several messages and operations defined earlier are instantiated. There are either defined from the imported BWS ‘’squiz” types or in the CWS (lines 35 to 45).

The implementation of the CWS begins with the main method (line 34). In addition to the instantition of messages, the main method is composed of a *behaviour block* identified by one of the BPEL constructors and/or an *implementation block* identified by the *JAVACODE* keyword. These two blocs may be nested and, all the variables declared in the *JAVACODE* section can be accessed by the BPEL constructors.

In this example, we first define the CWS behaviour using the different BPEL constructors. The *sequence* constructor specifies that the execution of the following instructions must be performed sequentially.

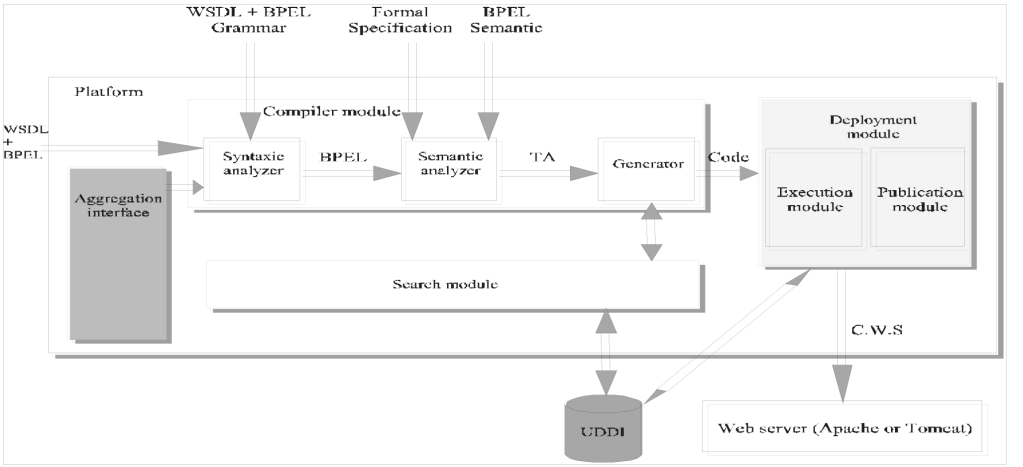
The definition of this example in JCWSL is as follows:

Example 1. Quiz game example

There are two types of function for calling an operation: execute or invoke. Both functions allow the assignment of an input, output or input/output messages. The main difference between these two functions is the external visibility. In this example, we call the invoke function which takes an input and output messages (“modeChoice” and “mode” respectively) and which is visible for the outside (*i.e.* it is part of the CWS behaviour).

The *pick* constructor is then used to wait, a given time, for a player’s answer. The CWS switches then according to the chosen mode in order to send a question of the appropriate level. After sending a question, the service will also use the pick constructor to implement a deadline after which it will go to the next question. In fact, it waits for 5 units of time

Figure 7. Platform architecture



the reception of the answer before going to the next question. At the end, the CWS returns the final score computed according to the received answers (line 95).

JCWSL Description

The development of a CWS with JCWSL is divided into two sections: an importation block **importBlock** and a definition block **defBloc**. The definition block represents the body of the complex service.

CWSLanguage::≡ (<importBloc> <defBloc>)+

Importation block: importBlock: is composed of two importation types, *JAVA* packages and Web service importations. Since our language is an extension of the *JAVA* language, JAVA packages needed by a CWS are imported exactly as in any JAVA program.

importBloc::≡ (“importBWS” <ID> = <STRING>”;”| “import” <ID> (.<ID>)* “;”)*

Web service importation allows to include in the complex Web service the required Web services. An imported Web service is identified by a unique name mapped with its URL, and which acts as a namespace for the service. Inside the definition block, a service data structure can be accessed exactly the same way as any *JAVA* package. In fact, an imported service is considered as a package composed of a set of classes representing the messages, the operations and the types declared in the WSDL file. Each service library is composed of a set of classes: a class for each complex type defined in the *types* section in the WSDL file, a class per message and a set of *stub* classes representing a client for each binding type.

Stub classes have the same name as the binding element in the WSDL file and are composed of the list of service operations accessible via the link. **importBloc** may contain zero or several import instructions.

Definition block represents the body of the CWS. It contains the definition and the behaviour of the complex Web service. It is composed of two blocs: a declaration block **declaration** and a main block **main**.

defBlock::≡ “public” “CWSDefinition” <ID> (<declaration> <main>)+

Declaration block is composed of local operations declaration and/or messages declaration. This block is optional, no messages or operations needs to be declared.

declaration::≡ (<opdeclaration> | <messDeclaration>)*

An operation is identified by a name and one or two messages depending on its type. There are three types of operations: input, output, and input/output. The operations defined in this section are appended to those defined by the imported Web services. An operation may also include a body composed of *JAVA* code.

opDeclaration (((“public defOperation” <operationName>
“(“ “input” “:” <messageType> <ID> | “output” “:” <messageType> <ID>
| “input” “:” <messageType> <ID> “,” “output” “:” <messageType> <ID>
“)” “{”<operationBody>”}”

operationBody is the main part of the operation. It is composed of either a set of JAVA instructions JAVAInstructions or the BPEL constructor execute, or both. JAVAInstructions is preceeded by the JAVACODE keyword.

operationBody ((((<java> | <execute>)*
<java> (((“JAVACODE” {(<javaInstructions>)+}

A message is identified by its name and is composed of one or several variables. Message declaration does not contain methods. Automatically, while developing the CWS, a set of management methods are created. These methods have the following form: get<variableName> and set<variableName>.

messDeclaration (((“public defMessage” <messageName>
“{”(<variableType> <variableName>”;”)+”}”

Main block The second part of defBlock is the main block. It contains the definition and the behaviour of the CWS. It is composed of the service activities and of JAVA code. Service activities are defined using BPEL constructors, whereas the service definition is written in JAVA. Those two types are not overlapped in our language, but may be nested. However, the visibility of the JAVA variables is preserved outside the JAVACode bloc and can be used in the activity blocs. In the activity bloc, only constructors imbrication and operation invocation are allowed.

main::≡ “void” “main”
“{”(<process>|<java>)* “}”

process represents CWS behaviour and is composed of the following constructors:

process::≡ <pick> | <switch> | <opCall> | <sequence> | <while> | <wait>

pick is an event handler to catch the cited events. An event is either the triggering of an alarm or a message reception. It suspends the execution until the occurrence of an event, and then executes the appropriate code.

<pick>::≡ “pick” “{” (<evt>)+ [<delayfor>] “}”
<evt>::≡ “eventhandler” “{” <invoke> “}” “{” (<process>)* “}”
<delayfor>::≡ “delayfor” “(” <integer_literal> “)” “{” (<process>)* “}”

switch is a conditional statment. It is used to perform different actions depending on the value of the associated condition.

switch::≡ Switch (“{” “JAVACODE” “{”<condExpr> “}” “}” “{ <process> “}”)

opCall allows to call an operation in a visible or invisible way. Therefore, it is composed of two constructors *invoke* and *execute*. In both cases, the operation type is either one way or request/reply. *invoke* and *execute* are executed exactly the same way, the only difference is

the external visibility of the operation. **invoke** allows to invoke an operation in a transparent way. The invocation is visible from outside the CWS and is taken into consideration while defining the CWS behaviour. Whereas, **execute** allows to invoke an operation internally, *i.e.* the invocation is not visible from the outside and is not taken into account when defining the CWS behaviour. In both cases, *inputVariable* and *outputVariable* represent messages.

opCall::≡ (<invoke> | <execute>)
invoke::≡ "invoke" "(" <operationName> [,<inputVariable> | (<inputVariable>, <outputVariable>)] ")"
execute::≡ "execute" "(" <operationName> [,<inputVariable>| (<inputVariable>, <outputVariable>)] ")"

sequence defines a block of instructions to be executed sequentially.

sequence::≡ "sequence" "{" <process> "}"

while is a loop statement. It executes the block of instructions as long as the condition is satisfied.

while::≡ "while" "{" "JAVACODE" "{"<conditionalExpr> "}" "}" "{"<process>"}"

wait suspends the process execution for the specified duration or until the expiration of the deadline.

wait::≡ "Wait" ("for"=<duration-expr> | "until"=<deadline-expr>)

WEB SERVICE DESIGN FRAMEWORK

Platform Architecture

The platform allows to build CWS from BWS. It offers different modules in order to define, compile and deploy CWS. It is composed of four modules: search module, aggregation interface, compiler, and deployment module Figure 7 illustrates the different modules of the platform as well as their different interactions.

Search Module

This module is a UDDI client which allows to explore and search the UDDI registry in order to locate basic Web services. It is used by the generator to locate and upload BWS and consequently communicates with the UDDI registry.

Aggregation Interface

It is a CWS development environment. It allows to define CWS from BWS operation's aggregation. The environment is based on JCWSL.

Compiler Module

The generator takes in entry the description of the CWS written in JCWSL and generates the corresponding code and behaviour in BPEL. It is composed of an analyzer and a generator.

- The analyzer applies syntactic and semantic analysis.
  - During syntactic analysis, the analyzer checks the file conformity with the language grammar and the validity of each BWS, constructs for each BWS its library <<**package**>>, and generates the syntactic tree of the CWS.
  - The semantic analyzer ensures that the CWS fulfills some properties. In fact, giving the semantic tree and an abstraction algorithm, it generates

the observable description of the service and its TA in order to check service ambiguity.
- The generator takes as input the TA and the syntactic tree generated earlier, and generates the CWS Java code.

Deployment Module

It generates the Web application which hosts the Web service. It also defines an implementation of the service communication model. In fact, information about the service communication is used in order to define the binding part of the service description.

Publication Module

It allows to publish, in a UDDI register, the BPEL4WS file of the service. The platform provides, in addition to the utilities necessary for the design and the development of complex services, a module named generic client, which generates a client for a complex service (see section Client Synthesis).

Client Module

It allows to generate automatically a client to interact correctly with the service if the service is not ambiguous, or an error otherwise. The generic customer is composed of two sub-modules: a synthesis module and an execution module. The synthesis module recovers, from the UDDI directory, the service specifications in BPEL, analyses and then produces the corresponding timed automaton. The execution module is a middleware which, at the service invocation time, loads the corresponding timed automaton in order to manage the client/service interaction.

An Application Life Cycle

This section describes, in detail, the various stages of a complex Web service installation using our platform: the service design, automatic generation of the service including the publication and the deployment, and finally the generation of the corresponding client.

Complex Web Service Design

The developer has a framework in order to describe its complex Web service (CWS). A module, in relation with the UDDI registry, can be used to locate existing Web services. The description of the service is made using the JCWSL language which makes it possible to describe at the same time the observable behaviour of the service with BPEL constructors and the service implementation with the *JAVA* language.

CWS Generation, Publishing and Deployment

The second step deals with the deployment of the CWS and includes the CWS generation, the behaviour generation and publishing, and the CWS deployment. From the file ".jcwsl", the compiler generates the CWS behaviour in BPEL and the Web application in *JAVA*. The publication of the complex service is carried out exactly the same way as for a basic service. Then, the deployment of a complex service consists of the creation of a JAXM servlet which plays the role of a proxy between the client and the service. When the servlet receives a message corresponding to the activation of an object (the first message in the protocol of the service), it creates an instance of the corresponding class of the complex service, initializes the correlation attributes (according to the message value), adds it in the queue and carries out its behaviour (*i.e.* starts the corresponding thread). The servlet URI corresponds to the address of the complex service. Each received message is redirected towards the instance of the corresponding class according to the object and the attributes values of the correlation instance.

Service Invocation

The last stage is the client generation starting from the service description in order to allow a correct interaction with the service. It consists of

the generation and the deployment of a JAXM servlet which acts as an input/output interface of the client instance of a given service. The servlet acts as a dispatcher which, at the reception of a SOAP message, redirects it towards the corresponding client instance. The initialization of an interaction creates a new automaton and links it to the servlet. The servlet URI address is used by all the clients.

Implementation

Parser Generation

The choice of *JAV*ACC (JAVA Compiler Compiler) to describe our language grammar is the natural choice since it extends *JAVA*. Once the description of the grammar is finished, it is written in a notation similar to BNF. In fact, *JAVACC* works with a .jj file. When the *JAVACC* is run against the .jj file, it generates a number of *JAVA* source files. One is the primary parsing code, Parser_1.*JAVA*, which you will invoke from your application when you have an expression to parse. *JAVACC* also creates six other auxiliary files that are used by the parser. Three files are specific to this particular grammar; the last four are generic helpers that are always generated no matter what the grammar looks like. Once *JAVACC* has generated these seven *JAVA* sources, they can be compiled and linked into a *JAVA* application. The new parser can be used to parse the description file for a CWS written in JCWSL. It produces a BPEL4WS describing the behaviour of the corresponding CWS.

Internal Structure

While parsing the CWS definition file, an internal representation is automatically created. This representation is a set of objects representing the different element of the service. Thus, an instance of the object called definition is created. Then, the other elements of the CWS are added *i.e.* the local messages, local operations, imported Web services. In order to do so, each time the parser meets an element of the service, it creates the corresponding element. An object called **import** containing the set of imported services is built. A CWS may declare local messages and operations. A **message** class is created for the set of local messages as well as an **operation** class is created for the set of local operations. A **portype** class is created in order to be able to invoke the declared operations. In a **portType**, it is possible to gather several operations. **Operation** class is always linked with the **message** class, since an operation handles messages (one or two). As for the behaviour block, it is represented as a set of activities which defines the behaviour orchestration of the CWS. Those activities are defined in the **definition** class. All the previously cited classes are designed using the *JAVA* API. *JAVA*, associated to XML allow a simple and rapid development of CWS. These classes are represented in XML format validated with XML Schema Language.

RELATED WORK

The composite Web service adds two dimensions by comparison to the simple ones; they are statefull and they obey to an operational behaviour (interaction protocol). This raises many theoretical and practical issues which are part of ongoing research (Nakajima, 2002). Due to the lack of 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 behavioural properties by acting directly on BPEL expressions. The main approach, followed by most of the state-of-the-art works, is to translate a service behaviour (BPEL process) into a mathematically well-founded model, considering just the semantical 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 behaviour. 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 (Breugel, 2005), 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 (Ferrara, 2004), a similar approach is given and which translates to LOTOS. In previous works, we give an operational semantic to Xlang (ancestor of BPEL) in order to verify the ambiguity (non usability) of a Web service behaviour (not deadlock-free) (Melliti, 2003; Haddad, 2004). In (Schlingloff, 2005) a similar work is done using Petri Net. In (Stahl, 2004; Ouyang, 2005; Hamadi, 2003) and (Schmidt, 2004), the authors propose a pattern-based translation of activities into Petri Nets, and then they use Petri Net properties to check properties.

Most of these works try to verify properties related to a single BPEL process (the coordinator). In this article, we begin by giving an observable operational semantic to abstract BPEL. We consider it as a grammar of timed process algebra and we define its operational semantic according to its informal definition. This step allows us to model the partners behaviour using a TTS (by applying operational rules) with regard to their interaction.

CONCLUSION

The approach developed in this article emphasizes a new interest of formal semantics. Traditionally, equipping a language of a formal semantics has two main goals: allowing the programmer to understand the language constructors in order to write correct applications and ensuring that the execution of a program is independent of the compiler and the target machine. In our case, the search of a semantic for complex Web services led us to raise an ignored problem of the experts: the service ambiguity.

Moreover, our approach (partially) answers to the service dynamicity since a client discovering a new service (or an existing but modified service) generates a timed automaton whose execution makes it possible to correctly interact with the service. Finally, this semantic decreases the cost of software development since the interface generation and the service deployment shares many components.

Our second contribution aims at unifying the design and the implementation process by providing a language that mixes BPEL with *JAVA*. In our environment, the designer starts from a UML description then transforms it in a BPEL specification and enlarges it (when necessary) in a JCSWL program that will translated in *JAVA* and deployed on the server.

Our future work will exploit this semantics in order to solve the service composition problem, namely how to guarantee that composed service does not block itself, never finishes, etc. In a complementary manner, we carry our efforts on the integration of the orientation aspect concepts in Web services. The *weaving* is obviously an activity related to the execution and thus lends itself naturally to our approach.

Alur, R., & Dill, D. L. (1994). A Theory of Timed Automata. *Theoretical Computer Science*, *126*(2), 183–235. doi:10.1016/0304-3975(94)90010-8

Alur, R., Fix, L., & Henzinger, T. A. (1999). Event-clock Automata: a Determinizable Class of Timed Automata. *Theoretical Computer Science*, *211*(1–2), 253–273. doi:10.1016/S0304-3975(97)00173-4

Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., et al. Van der Rijn, D., Yendluri, P., & Yiu, A. (2007). *Web Services Business Process Execution Language Version 2.0.* OASIS WSBPEL Technical Committee. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf.

Baeten, J. C. M., & Bergstra, J. A. (1991). Real Time Process Algebra. *Formal Aspects of Computing*, *3*(2), 142–188. doi:10.1007/BF01898401

Baresi, L., Heckel, R., Thöne, S., & Varrò, D. (2003). Modeling and Validation of Service-Oriented Architectures: Application vs. Style. *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering* (pp. 68-77). New York, NY: ACM Press.

Bellwood, T., Clément, L., & von Riegen, C. (2002). *Universal Description, Discovery and Integration.* OASIS UDDI Specification Technical Committee. http://www.oasis-open.org/cover/uddi.html.

Bergstra, J. A., & Klop, J. W. (1984). Process Algebra for Synchronous Communication. *Information and Control*, *60*(1-3), 109–137. doi:10.1016/S0019-9958(84)80025-X

Blow, M., Goland, Y., Kloppmann, M., Leymann, F., Pfau, G., Roller, D., & Rowley, M. (2004). *BPELJ: BPEL for JAVA*. BEA and IBM. http://www-128.ibm.com/developerworks/library/specification/ws-bpelj/.

Cambronero, M.E., Pardo, J.J., Díaz, G., & Valero, V. (2007). Using RT-UML for modelling web services. *ACM symposium on Applied computing.*

Christensen, E., Curbera, F., Meredith, G., & Weerawarana, S. (2001). *Web Services Description Language (WSDL) 1.1.* World Wide Web Consortium. http://www.w3.org/TR/wsdl.

Duftler, M. J., Mukhi, N. K., Slominski, A., & Weerawarana, S. (2001). *Web Services Invocation Framework (WSIF).* The Apache Software Foundation. http://ws.apache.org/wsif/.

Ferrara, A. (2004). Web Services: a Process Algebra Approach. [ACM.]. *ICSOC*, *2004*, 242–251. doi:10.1145/1035167.1035202

Gardner, T. (2003). UML Modelling of Automated Business Processes with a Mapping to BPEL4WS. *First European Workshop on Object Orientation and Web Services* (pp. 21-25), Germany.

Gronmo, R., & Jaeger, M. C. (2006). Model-Driven Methodology for building QoS-Optimised Web Service Compositions. *WEWST06*, Zurich, Switzerland.

Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J., & Nielsen, H. (2000). *Simple Object Access Protocol (SOAP) 1.1.* World Wide Web Consortium. http://www.w3.org/TR/SOAP/.

Haddad, S., Melliti, T., Moreaux, P., & Rampacek, S. (2004). A Dense Time Semantics for Web Services Specifications Languages. *Proc. of the 1st Int. Conf. on Information & Communication Technologies: from Theory to Applications (ICTTA'04)* (pp. 647–648), Damascus, Syria.

Haddad, S., Melliti, T., Moreaux, P., & Rampacek, S. (2004). Modelling Web Services Interoperability. *Proceedings of the Sixth International Conference on Entreprise Information Systems* (pp. 287-295), Porto, Portugal.

Hamadi, R., & Benatallah, B. (2003). A Petri Net-based model for web service composition. In K-D. Schewe & X. Zhou (Eds.), *Fourteenth Australasian Database Conference (ADC2003)*, v(17) of *CRPIT* (pp. 191-200). ACS.

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

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

Léonard, L., & Leduc, G. (1997). An Introduction to ET-LOTOS for the Description of Timesensitive Systems. *Computer Networks and ISDN Systems*, *29*(3), 271–292. doi:10.1016/S0169-7552(96)00078-5

Melliti, T., & Haddad, S. (2003). Synthesis of Agents for Web Services Interaction. *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*. Englewood Cliffs, NJ, USA: Prentice-Hall.

Nakajima, S. (2002). Model-Checking Verification for Reliable Web Service. *Workshop on Object-Oriented Web Services*, Seattle, Washington.

Nicollin, X., & Sifakis, J. (1991). An Overview and Synthesis on Timed Process Algebras. In J. W. de Bakker, C. Huizing, W. P. de Roever, & G. Rozenberg (Eds.), *Proc. Real-Time: Theory in Practice, REXWorkshop*, v(600) of *LNCS* (pp. 526-548). Springer-Verlag.

Nicollin, X., & Sifakis, J. (1994). The Algebra of Timed Process, ATP: Theory and Application. *Information and Computation*, *114*(1), 131–178. doi:10.1006/inco.1994.1083

Ouyang, C., Verbeek, E., van der Aalst, W., Breutel, S., Dumas, M., & ter Hofstede, A. (2005). *Formal Semantics and Analysis of Control Flow in WS-BPEL*. BPM-05-13, Business Process Management Center.

Plotkin, G. D. (1981). A Structural Approach to Operational Semantics. Technical Report FN-19, DIAMI, CSD, U. of Aarhus, Aarhus, Denmark.

REFERENCES

Schlingloff, B.-H., Martens, A., & Schmidt, K. (2005). Modeling and Model Checking Web Services. *Electronic Notes in Theoretical Computer Science*, *126*, 3–26. doi:10.1016/j.entcs.2004.11.011

Schmidt, K. &. Stahl, C. (2004). A Petri Net Semantic for BPEL4WS - Validation and Application. In E. Kindler (Ed.), *Workshop on Algorithms and Tools for Petri Nets (AWPN 04)* (pp. 1-6), Germany.

Schneider, S. A. (1995). An Operational Semantics for Timed CSP. *Information and Computation*, *116*(2), 193–213. doi:10.1006/inco.1995.1014

Skogan, D., Gronmo, R., & Solheim, I. (2004). Web Service Composition in UML. *EDOC '04: Proceedings of the Enterprise Distributed Object Computing Conference, Eighth IEEE International (EDOC'04)* (pp. 47-57), Washington, DC, USA: IEEE Computer Society.

Staab, S., van der Aalst, W., Benjamins, V. R., Sheth, A., Miller, J. A., & Bussler, C. (2003). Web Services: Been There, Done That? *IEEE Intelligent Systems*, *18*, 72–85. doi:10.1109/MIS.2003.1179197

Stahl, C. (2004). *A Petri Net Semantics for BPEL*. Technical report 10099, Humboldt-Universitäat zu Berlin.

Tidwell, D. (2000, November). Web Services - The Web's Next Revolution. *IBM developerWorks*.

van Breugel, F., & Koshinka, M. (2005). Dead-Path-Elimination in BPEL4WS. *5th International Conference on Application of Concurrency to System Design* (pp. 192-201). IEEE.

Céline Boutrous Saab is an assistant professor at LAMSADE (Laboratoire d'Analyse et Modélisation de Systèmes pour l'Aide à la Décision) located at the University of Paris-Dauphine since 2002. She obtained his PhD in computer science in 2000. Her research interests are focused on the design and the implementation of cooperative and distributed applications.

Demba Coulibay has obtained a master's degree of computer science at University of Kiev in 1992. He is currently both an engineer of University of Bamako and a PhD student under the supervision of C. Boutrous and S. Haddad. His main research interests concern Web services and software engineering.

Serge Haddad was formerly student at the Ecole Normale Supérieure de l'Enseignement Technique in Mathematics. He obtained his PhD in computer science in 1987. In 1993, he became a full professor at University Paris-Dauphine and he has recently moved to ENS Cachan. His research interests are focused on the design, the verification and the evaluation of cooperative and distributed applications.

Tarek Melliti is, since 2006, an assistant professor at IBISC (Informatique biologie intégrative et systèmes complexes) laboratory located at University of Evry Val-Essonne (France). He obtained his PhD in Computer Science in 2004. His researches deal with formal methods applied to orchestrated and choreographed Web services Composition. Recently, his researches focused on model-based Web services Diagnosis and Diagnosability.

Patrice Moreaux is "professeur agrégé de Mathématiques" and received his PhD in Computer Science in 1996. Since 2005, he is a full professor at University of Savoie. His research interests are focused on the modelling, the verification and the performance evaluation in automation, networks, distributed and parallel computer systems.

Sylvain Rampacek is an assistant professor at LE2I laboratory (UMR CNRS 5158) located at Université de Bourgogne (France). He obtained his PhD in Computer Science in 2006. His research is focused on formal methods applied to Service Oriented Architecture. The results are implemented and mainly consist to help the development of secured and adapted software in this environment.