# Formal Modeling and Discrete-Time Analysis of BPEL Web Services

**Radu Mateescu**
INRIA / VASY project-team, Centre de Recherche Grenoble – Rhône-Alpes
655, avenue de l'Europe, Montbonnot, F-38334 Saint Ismier Cedex, France
E-mail:Radu.Mateescu@inria.fr

**Sylvain Rampacek**
LE2I, Faculté des Sciences Mirande, Aile de l'Ingénieur,
Université de Bourgogne, BP 47870, F-21078 Dijon Cedex, France
E-mail:Sylvain.Rampacek@u-bourgogne.fr

**Abstract:** Web services are increasingly used for building enterprise information systems according to the Service Oriented Architecture (SOA) paradigm. We propose in this paper a tool-equipped methodology allowing the formal modeling and analysis of Web services described in the BPEL language. The discrete-time transition systems modeling the behavior of BPEL descriptions are obtained by an exhaustive simulation based on a formalization of BPEL semantics using the Algebra of Timed Processes (ATP). These models are then analyzed by model checking value-based temporal logic properties using the CADP toolbox. The approach is illustrated with the design of a Web service for GPS navigation.

**Keywords:** Web services, formal specification, model checking, exhaustive simulation, process algebra.

## 1 Introduction

Information systems present in companies and organizations are complex software artifacts involving concurrency, communication, and coordination among various applications that exchange data and participate to business processes. *Service Oriented Architecture* (SOA) [18] is a state-of-the-art methodology for developing information systems by structuring them in terms of services, which can be distributed and composed over a network infrastructure to form complex business processes. Web services are a useful basis for implementing business processes, either by wrapping existing software or by creating new functionalities as combinations of simpler ones. BPEL (*Business Process Execution Language*) [17] is a standardized language of wide industrial usage for describing abstract business processes and detailed Web services. It allows to capture both the behavioral aspects (concurrency and communication) and the timing aspects (duration of activities) of Web services.

The BPEL language allows to create Web services either from scratch, or as the composition of existing sub-services, which can be invoked sequentially (one at a time) or concurrently (several ones at the same time). Each Web service described in BPEL can be used as a sub-service by other Web services (described in BPEL or not), thus enabling a hierarchical construction of complex Web services. A BPEL business process is defined by a workflow consisting of various steps, which correspond internally to algorithmic computations (possibly with time constraints) and externally to message-passing interactions with a client. Business processes are typically built upon existing Web services (although this is not mandatory), each one being specialized for carrying out a particular task. These sub-services are invoked every time a specific information is needed during a step of the workflow; therefore, a business process is not simply the set of sub-services it is built upon, but acts as an orchestrator of these sub-services in order to provide newly added functionalities.

The conjunction of concurrency and timing constraints makes business processes complex and requires a careful design in order to avoid information losses and to obtain a satisfactory quality of service. In this context, formal modeling and analysis techniques available from the domain of concurrent systems allow to improve the quality of the design process and to reduce the development costs by detecting errors as soon as possible during the business process life cycle. These techniques can operate successfully on languages equipped with a formal semantics definition, from which suitable models can be constructed and analyzed automatically.

In this paper, we propose a tool-supported approach for the formal modeling and analysis of business processes and Web services described in Bpel. Our approach consists of the following ingredients: the definition of a formal semantics of Bpel in terms of process algebraic rules, taking into account the discrete-timing aspects [13, 14]; the automated generation of models (state/transition graphs) from the Bpel specifications using an exhaustive simulation based on the formal semantics rules, implemented in the WSMod tool; and the analysis of the resulting models by using standard verification tools for concurrent systems, such as Cadp [11]. We illustrate the application of this approach to the design and discrete-time analysis of a Web service for Gps navigation.

**Related work.** The modeling and analysis of Web services benefits from a considerable attention in the research community. The Wsat tool proposed in [7, 8] gives to Web service designers the possibility of verifying Ltl properties on Bpel business processes by applying the Spin model checker. Each Bpel process is transformed into a Promela model (via a pattern) and connected to other processes in the description. This work covers only the untimed aspects of Bpel.

Another approach, proposed in [32], uses the Cress (*Chisel Representation Employing Systematic Specification*) notation for specifying the untimed behavior of Web services. Cress descriptions are translated into the formal description technique Lotos [16] and analyzed with dedicated tools, such as Topo, Lola or Cadp. A direct translation from Bpel to Lotos is given in [6], enabling the use of the aforementioned tools for analyzing the untimed behavior of Web services. Bpel was also used as target language for producing executable Web services from Lotos specifications [2, 28]; this allows to combine the advantages of the formal verification using Cadp and of the deployment and execution features of Bpel.

Compared to existing work, our approach differs in the following respects: it is based on a translation of Bpel directly into state/transition graphs, without using an intermediate language such as Promela or Lotos, thus being potentially more efficient; and it handles not only the behavioral, but also the discrete-time aspects of Bpel descriptions.

**Paper outline.** Section 2 presents our methodology and software platform for modeling and analyzing Bpel descriptions. Section 3 describes the Gps Web service case-study and its analysis using the platform. Finally, Section 4 gives some concluding remarks and directions for future work.

## 2 Modeling and Analysis Approach

Web services can be seen as complex distributed systems that communicate by message-passing. Therefore, their design methodology can be naturally supported by the formal modeling and analysis techniques stemming from the domain of concurrent systems. To apply these techniques, it is necessary to represent the dynamic behavior of Web services in a formal, non-ambiguous manner.

The approach we propose for the modeling and analysis of Web services described in Bpel is illustrated in Figure 1. Our software platform consists roughly of two parts, described in the sequel: the Bpel descriptions are first translated into discrete-time Ltss using the WSMod tool, and are subsequently analyzed using the Cadp verification toolbox.

### 2.1 Translation from BPEL to discrete-time LTSs

The behavior of a Web service comprises not only the concurrency and communication between its various constituent activities, but also the delay of response of the service. These aspects can be modeled using dtLtss (*discrete-time Labeled Transition Systems*), i.e., state/-transition graphs in which every transition is labeled by an action performed by the Web service. The actions are of the following kinds: emissions and receptions of messages, prefixed by '!' and '?', respectively; elapsing of time, represented by the symbol $\chi$ (discrete-time tick, also noted *time*); the internal action $\tau$ (or *tau*) denoting unobservable activity of the service; and the terminating action $\surd$ (or *done*), which is the last internal action that a service can do.

The global behavior of the Web service (and therefore, the actions it can perform) is obtained by an exhaustive simulation of the Bpel description, performed by the WSMod tool (see Figures 2, 3), which is able to handle both discrete [13] and continuous [14] time representations. WSMod takes two different inputs (see Figure 1):

- A Web service description in Bpel [17], a standardized language allowing to specify the behavior of business processes. Bpel supports two different types of business processes: *executable* processes specify the behavior of business processes in full detail, such that they can be executed by an orchestration engine; and *abstract* business protocols specify the public message exchanges between the service and a client (i.e., excluding the message exchanges which take place internally, e.g., during invocations of sub-services).

- A formal representation of the Bpel semantics, based on the Algebra of Timed Processes (Atp) [27], which specifies using operational rules how the model of the business process behavior is generated. Depending on the time representation chosen, the resulting model is either a dtLts, or a timed automaton (Ta) [1]. The Atp rules formalizing the Bpel semantics in discrete-time is shown in Table 1. For example, the process "time" can only elapse time (represented by the $\chi$ action), and the process "receive" or "reply" can send or receive a message (first rule) or elapse time too (second rule).

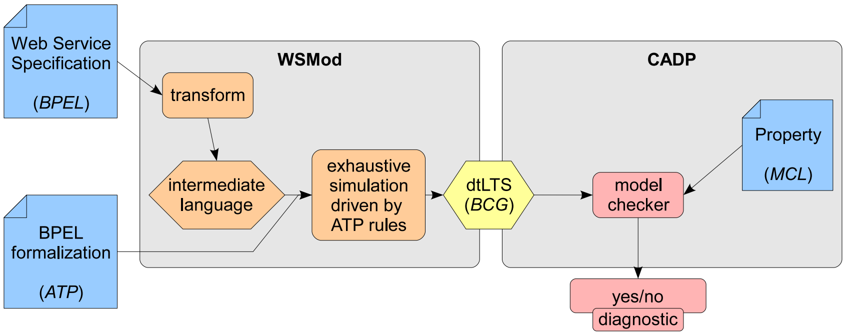



Figure 1: Platform for Web service modeling and analysis

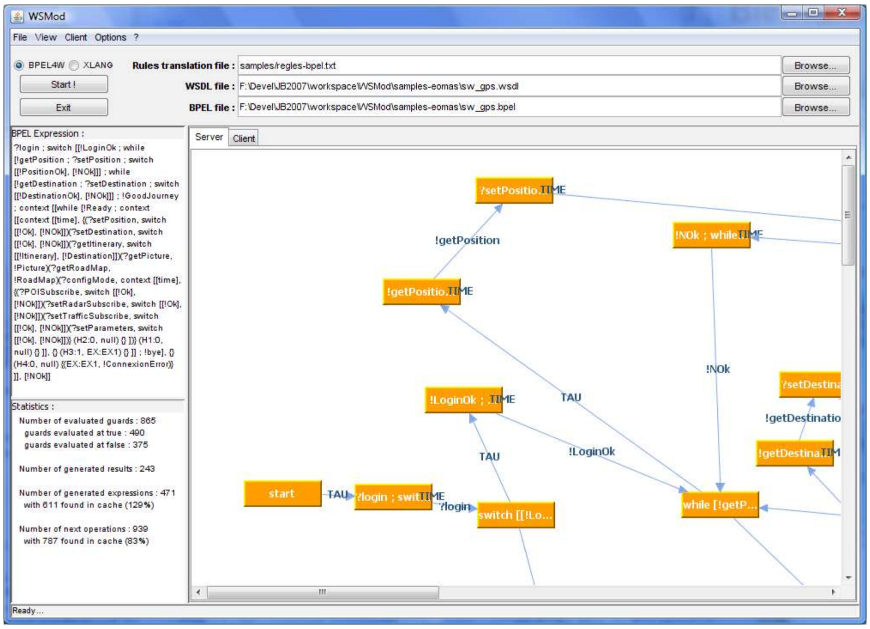



Figure 2: Screenshot of the WSMod tool

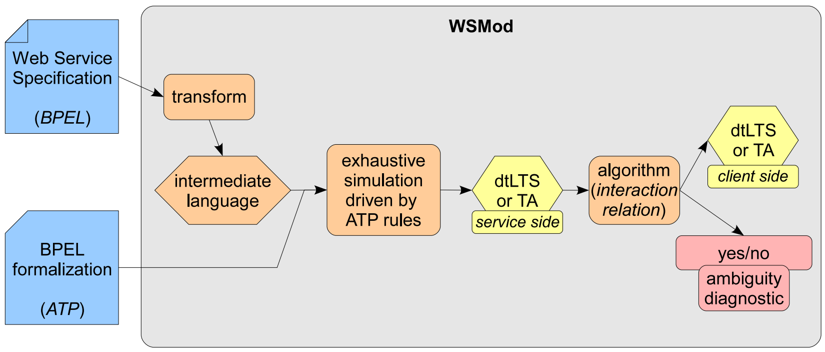



Figure 3: Main functionality of WSMod tool (dtLts: discrete-time Lts, Ta: timed automaton)

To generate the model representing the behavior of the input Bpel description, WSMod performs an exhaustive simulation guided by the operational Atp rules. The tool is also able to synthesize automatically the model of an adapted client interacting with the Web service, whose behavior complies with that of the service as regards emission and reception of messages, time elapsing, etc (see Figure 3). In this study, we focus only on the Web service model generation feature of WSMod in discrete time.

Table 1: The process algebra formalizing BPEL, in discrete-time

| BPEL | ATP |
|---|---|
| empty | $\text{empty} \xrightarrow{\surd} 0$ |
| time | $\text{time} \xrightarrow{\chi} \text{time}$ |
| throw | $\forall e \in E_X,\ \text{throw}[e] \xrightarrow{e} 0$<br>with $E_X$ set of exceptions that can be thrown. |
| receive / reply | $*o[m] \xrightarrow{*m} \text{empty}$ with $* \in \{?, !\}$<br>$*o[m] \xrightarrow{\chi} *o[m]$ |
| sequence ( ; ) | $\forall a \neq \surd,\ \dfrac{P \xrightarrow{a} P'}{P\ ;\ Q \xrightarrow{a} P'\ ;\ Q}$<br>$\forall a,\ \dfrac{P \xrightarrow{\surd} P' \quad \wedge \quad Q \xrightarrow{a} Q'}{P\ ;\ Q \xrightarrow{a} Q'}$ |
| switch | $\text{switch}[\{P_i\}_{i \in I}] - \forall i \in I,\ \text{switch}[\{P_i \mid i \in I\}] \xrightarrow{\tau} P_i$ |
| while | $\text{while}[P] \xrightarrow{\tau} P\ ;\ \text{while}[P]$<br>$\text{while}[P] \xrightarrow{\tau} \text{empty}$ |
| scope | Let $M_I = \{m_i \mid i \in I\}$ a set of messages and<br>let $E_J = \{e_j \mid j \in J\}$ a set of exceptions.<br>$\text{scope}(P, E)$ with $E = [\{(m_i, P_i) \mid i \in I\}, (d, Q), \{(e_j, R_j) \mid j \in J\}]$<br>$\dfrac{P \xrightarrow{\surd}}{\text{scope}(P,E) \xrightarrow{\surd} 0}$<br>$\forall a \notin \{\chi, \surd\} \cup E_X \cup M_I \quad \dfrac{P \xrightarrow{a} P'}{\text{scope}(P,E) \xrightarrow{a} \text{scope}(P',E)}$<br>$d > 1,\ \dfrac{P \xrightarrow{\chi} P' \ \text{ and } \ \forall a \in E_X \cup \{\tau, \surd\},\ \neg(P \xrightarrow{a})}{\text{scope}(P,E^d) \xrightarrow{\chi} \text{scope}(P,E^{d-1})}$<br>$\dfrac{P \xrightarrow{\chi} P' \ \text{ and } \ \forall a \in E_X \cup \{\tau, \surd\},\ \neg(P \xrightarrow{a})}{\text{scope}(P,E^1) \xrightarrow{\chi} Q}$<br>$\forall i \in I,\ \dfrac{\forall a \in E_X \cup \{\tau, \surd\},\ \neg(P \xrightarrow{a})}{\text{scope}(P,E) \xrightarrow{?m_i} P_i}$<br>$\forall j \in E_J,\ \dfrac{P \xrightarrow{e_j}}{\text{scope}(P,E) \xrightarrow{\tau} R_j}$<br>$\forall e \notin E_J,\ \dfrac{P \xrightarrow{e}}{\text{scope}(P,E) \xrightarrow{e} 0}$ |
| pick | $\text{pick}[E] = \text{scope}(\text{time}, E)$<br>with $E = [\{(m_i, P_i) \mid i \in I\}, (d, Q), \{(e_j, R_j) \mid j \in J\}$ |
| flow | $\text{flow}[\{P_i\}_{i \in I}]$ executes simultaneously the set of processes $\{P_i\}$.<br>$\forall a \in E_X \cup \{\tau\},\ \dfrac{\exists j \in I,\ P_j \xrightarrow{a} P'}{\text{flow}[\{P_i \mid i \in I\}] \xrightarrow{a} \text{flow}[\{P_i \mid i \in I \setminus \{j\}\} \cup \{P'\}]}$<br>$\forall m \in M,\ \dfrac{\exists j \in I,\ P_j \xrightarrow{*m} P' \ \text{ and } \ \forall i \neq j, \forall a \in E_X \cup \{\tau\},\ \neg\ \exists k \in I,\ (P_k \xrightarrow{a})}{\text{flow}[\{P_i \mid i \in I\}] \xrightarrow{*m} \text{flow}[\{P_i \mid i \in I \setminus \{j\}\} \cup \{P'\}]}$<br>$\dfrac{\forall i \in I,\ P_i \xrightarrow{\surd} 0}{\text{flow}[\{P_i \mid i \in I\}] \xrightarrow{\surd} 0}$<br>$\dfrac{\exists J \neq \emptyset,\ J \subseteq I,\ \forall i \in J,\ P_i \xrightarrow{\chi} P'_i \ \text{ and } \ \forall i \in I \setminus J,\ P_i \xrightarrow{\surd}}{\text{flow}[\{P_i\}_{i \in I}] \xrightarrow{\chi} \text{flow}[\{P'_i\}_{i \in J} \cup \{P_i\}_{i \in I \setminus J}]}$ |

#### 2.1.1 BPEL Process to ATP

This section describes shortly each algebra rule presented in Table 1. In fact, each process described here corresponds to a process in BPEL language. For readability, we do not follow the (verbose) XML syntax of a BPEL process.

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

**The** time **process** can only elapse time (represented by the $\chi$ action).

**The** throw **process** can raise an exception. Generally, in a correct specification, the exception must the catched in some scope process.

**The** receive **or** reply **process** can send ($!o[m]$) or receive ($?o[m]$) a message (first rule) or elapse time too (second rule). They correspond to input/output WSDL operations.

**The** sequence $P;Q$ **process** executes the process $P$ followed by 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.

**The** switch$[\{P_i\}_{i \in I}]$ **process** chooses to execute one of $P_i$ process. In BPEL, the choice is done internally by the Web service: the client doesn't know the result of the condition evaluation. So, we translate this choice by the internal action ($\tau$).

**The** while$[P]$ **process** executes the process $P$ as long as an internal condition is evaluated at *true*. Again, the client does not know the result of the condition evaluation.

**The** scope$(P, E)$ **process** executes the process $P$ according to some events and guards (timing and exception) defined by the Web services and represented here by the expression $E = [\{(m_i, P_i) \mid i \in I\}, (d, Q), \{(e_j, R_j) \mid j \in J\}]$, where:

- $m_i$ are the events and $P_i$ the associated processes triggered;
- $d$ the maximum execution time unit for process $P$ (otherwise, the process $Q$ is executed);
- $e_j$ the exceptions that can be catched and $R_j$ the associated processes triggered.

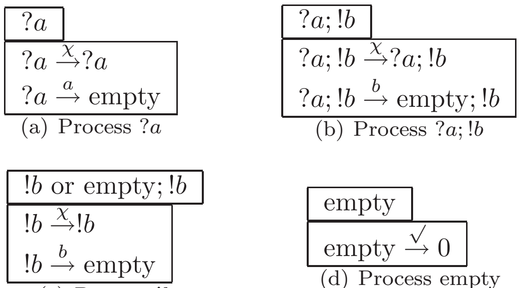
**The** pick **process** is a particular case of scope in which there is no main process (and more precisely, only one that consist to elapse time).

**The** flow$[\{P_i\}_{i \in I}]$ **process** executes simultaneously the set of processes $P_i$, according to the rules concerning time, exceptions, etc. In BPEL, there is a mechanism that enables the Web service to do some synchronization dependencies (*links*) that is not present in the rules, but taken into account in the WSMOD tool.

#### 2.1.2 Transition system synthesis

Transition system synthesis begins from the algebra rules (describe in Table 1) and the intermediate language (see Figure 3). This intermediate language is based on a syntactical tree. We can remark that the root of the syntactical tree is the global business process, which denotes the global behavior of the Web service.

By analysing its root according to the algebra rules, we can infer a set of actions that can be executed: these actions will be represented by transitions in the DTLTS. Due to the structure of the global expression representing the business process, it can be useful to repeat the analysis recursively to refine the set of actions.




(a) Process $?a$ (b) Process $?a; !b$ (c) Process $!b$ (d) Process empty

Figure 4: Actions set synthesis example for the process $?a; !b$, in discrete time.

For example, we can take the process "$?a; !b$" corresponding to the reception of an "$a$" message, followed sequentially by the reply of a "$b$" message. The set of actions that a process sequence can do, depends on the first process in it. So, we must analyze, first, the process "$?a$". This process can do two actions: (i) the time elapsing or (ii) receiving the message "$a$" (see Figure 4(a)). Next, we must verify the compatibility of the process "$?a$" actions with each guard of sequence process, and then, determine the action set that this one can realize. Here, the two sets are identical.

The "empty$; !b$" process has been created during the previous step. We can simplify this process, according to the sequence algebra rules, for two reasons: (i) the "empty" process can only terminate ($\surd$ action), and (ii) if the first sub-process of a sequence can terminate, then the sequence process executes the second sub-process. So, "empty$; !b$" process can become (or is equivalent to) "$!b$" process.

The analysis of "empty$; !b$" and empty processes are explained respectively in Figure 4(c) and Figure 4(d).

Each step enables us to determine the global DTLTS of "$?a; !b$" process (see details in Figure 4). The initial state is "$?a; !b$" and the final state is "0". Of course, each analysis step is not hard-coded, but the tool analyses the algebra rules to determine each derivate action to be executed by a given process.

### 2.2 Analysis of discrete-time LTSs

Once the DTLTS model of the BPEL specification under design has been obtained, it can be analyzed by using standard tool environments available for concurrent systems. For our purpose, we use the CADP (*Construction and Analysis of Distributed Processes*) toolbox [11] dedicated to the formal specification and verification of concurrent asynchronous systems. CADP accepts as input specifications written in process algebraic languages, such as LOTOS [16], FSP [21, 29] or CHP [22, 30], as well as networks of communicating automata given in the EXP language [19]. These formal specifications are translated by specialized compilers into labeled transition systems (LTSs), i.e., state spaces modeling exhaustively the dynamic behavior of the specified systems. LTSs are the formal model underlying the analysis functionalities offered by CADP, which aim at assisting the user throughout the whole design process: code generation and rapid prototyping, random execution, interactive and guided simulation, model checking and equivalence checking, test case generation, and performance evaluation.

An LTS can be represented within CADP in two complementary ways: either *explicitly*, by its list of states and transitions encoded as a file in the BCG (*Binary Coded Graphs*) format equipped with specialized compression algorithms, or *implicitly*, by its successor function given as a C program complying to the interface defined by the OPEN/CÆSAR [9] environment for graph manipulation. The explicit representation is suitable for *global* verification algorithms, which explore transitions forward and backward, whereas the implicit representation is suitable for *local* (or *on-the-fly*) verification algorithms, which explore transitions forward, thus enabling an incremental construction of the LTS during verification. To deal with large systems, CADP provides several advanced analysis techniques: on-the-fly verification, partial order reductions, compositional verification, and massively parallel verification using clusters of machines.

CADP contains currently over 40 tools and libraries for LTS manipulation, which can be invoked either in interactive mode via the EUCALYPTUS graphical interface, or in batch mode via the SVL [10] scripting language dedicated to the description of complex verification scenarios. The toolbox was used for the validation of more than 100 industrial case-studies[1].

#### 2.2.1 A fragment of the MCL language

Since we focus on model checking discrete-time properties on DTLTSs, we could apply in principle existing tools operating on LTSs, such as the EVALUATOR 3.5 [23] on-the-fly model checker of CADP, which takes as input temporal formulas expressed in regular alternation-free $\mu$-calculus, an extension of alternation-free $\mu$-calculus [5] with regular expressions over transition sequences. However, the evaluation of discrete-time properties requires the counting of *time* actions in the DTLTS; this can be encoded in standard modal $\mu$-calculus [31] using fixed point operators (one operator for each counter value), but may lead to prohibitively large temporal formulas, as noticed in the framework of temporal CCS [26]. Discrete-time properties can be succinctly and naturally formulated using data-handling extensions of the modal $\mu$-calculus, such as the MCL language underlying the EVALUATOR 4.0 tool recently integrated into CADP.

MCL (*Model Checking Language*) [24] is an extension of the alternation-free $\mu$-calculus [5] with regular expressions and data-handling operators. We describe below the syntax and semantics of an MCL fragment containing modal operators equipped with extended regular expressions over transition sequences, a more detailed description of the language being available in [24]. The MCL fragment considered consists of action formulas (noted $\alpha$), regular formulas (noted $\rho$), and state formulas (noted $\phi$), which characterize actions, transition sequences, and states of the DTLTS, respectively. Action formulas have the following syntax:

$$\begin{array}{rcl} \alpha & ::= & a \\ & | & \mathsf{true} \\ & | & \mathsf{false} \\ & | & \neg\alpha \\ & | & \alpha_1 \vee \alpha_2 \\ & | & \alpha_1 \wedge \alpha_2 \end{array}$$

An action formula $\alpha$ denotes a set of action names (transition labels) of the DTLTS. The action name $a$ denotes the singleton set containing the action $a$. The boolean connectors have their usual interpretation: $\mathsf{true}$ and $\mathsf{false}$ denote the set of all actions of the DTLTS and the empty set; $\neg\alpha$ denotes the complement of the action set denoted by $\alpha$; $\alpha_1 \vee \alpha_2$ and $\alpha_1 \wedge \alpha_2$ denote the union and the intersection of the action sets denoted by $\alpha_1$ and $\alpha_2$. Action formulas provide a simple, yet useful means of characterizing subsets of actions. This kind of formulas were originally introduced in action-based logics such as ACTL (*Action-based Computation Tree Logic*) [4].

Regular formulas have the following syntax:

$$\begin{array}{rcl} \rho & ::= & \alpha \\ & | & \mathsf{nil} \\ & | & \rho_1.\rho_2 \\ & | & \rho_1|\rho_2 \\ & | & \rho^* \\ & | & \rho^+ \\ & | & \rho\{m\} \\ & | & \rho\{m...n\} \\ & | & \rho\{m...\} \end{array}$$

A regular formula $\rho$ denotes a binary relation containing the couples of states which are source and target of a transition sequence in the DTLTS such that the word obtained by concatenating all actions labeling the transitions of the sequence belongs to the regular language defined by $\rho$. The regular formula $\alpha$ denotes all one-step sequences consisting of a single transition labeled by an action satisfying the action formula $\alpha$. The $\mathsf{nil}$ operator denotes the empty sequence, containing zero transitions. The concatenation $\rho_1.\rho_2$ denotes the sequences containing two adjacent subsequences satisfying $\rho_1$ and $\rho_2$, respectively. The choice $\rho_1|\rho_2$ denotes the sequences satisfying either $\rho_1$, or $\rho_2$. The transitive reflexive closure $\rho^*$ (resp. the transitive closure $\rho^+$) denotes the sequences consisting of the concatenation of zero or more (resp. one or more) subsequences satisfying $\rho$. The bounded iterations $\rho\{m\}$, $\rho\{m...n\}$, and $\rho\{m...\}$ denote the sequences consisting of the concatenation of exactly $m$, between $m$ and $n$, and at least $m$ subsequences satisfying $\rho$, respectively. These extended regular operators are similar to the operators implemented by string searching tools such as the `egrep` utility available on

[1] See the online catalog at
`http://www.inrialpes.fr/vasy/cadp/case-studies`

Unix systems; in practice they turn out to be as useful for describing transition sequences in DTLTSs as their `egrep` counterparts are for describing character strings.

State formulas have the following syntax:

$$\begin{array}{rcl} \phi & ::= & \mathsf{true} \\ & | & \mathsf{false} \\ & | & \neg\phi \\ & | & \phi_1 \vee \phi_2 \\ & | & \phi_1 \wedge \phi_2 \\ & | & \langle\rho\rangle\,\phi \\ & | & [\rho]\phi \\ & | & X \\ & | & \mu X.\phi \\ & | & \nu X.\phi \end{array}$$

A state formula $\phi$ denotes a set of states of the DTLTS. The boolean connectors have the usual interpretation over the set of DTLTS states. The possibility modality $\langle\rho\rangle\,\phi$ denotes the states from which there is (at least) an outgoing transition sequence satisfying $\rho$ and leading to a state satisfying $\phi$. The necessity modality $[\rho]\phi$ denotes the states from which all outgoing transition sequences satisfying $\rho$ must lead to states satisfying $\phi$. The minimal fixed point operator $\mu X.\phi$ (resp. the maximal fixed point operator $\nu X.\phi$) denote the minimal (resp. maximal) solution of the equation $X = \phi$ interpreted over the powerset lattice of the state set. Propositional variables $X$ denote sets of states; they are bound by the fixed point operators in a way similar to quantifiers in first-order logic. The state formulas of the MCL fragment above are similar to those of the modal $\mu$-calculus, except for the two modalities containing regular formulas, which are inspired from dynamic logics such as PDL (*Propositional Dynamic Logic*) [15].

The usage of this MCL fragment for specifying discrete-time properties of BPEL descriptions will be illustrated in Section 3.3.

## 3 Case Study: A Web Service for GPS Navigation

We illustrate in this section the application of our approach to the modeling and analysis of a Web service dedicated to GPS navigation. Given the relative complexity of this Web service, we do not detail here its textual BPEL and WSDL descriptions, but present its workflow graphically using the BPMN [12] notation.

### 3.1 System description

The purpose of the GPS Web service is to compute itineraries from a position to a destination fixed by a user (client of the service). In addition to the requested itinerary, the user can also obtain: pictures of the travel (taken from the air), the global map of the itinerary, and various kinds of information (about traffic, radar stations, point of interest (POI), etc). At last, the user can configure the subscription to the various kinds of information, as well as some parameters of the travel (e.g., to take motorway or not, to deviate toward a POI, etc.). The relationships between these functionalities are represented in Figure 5.

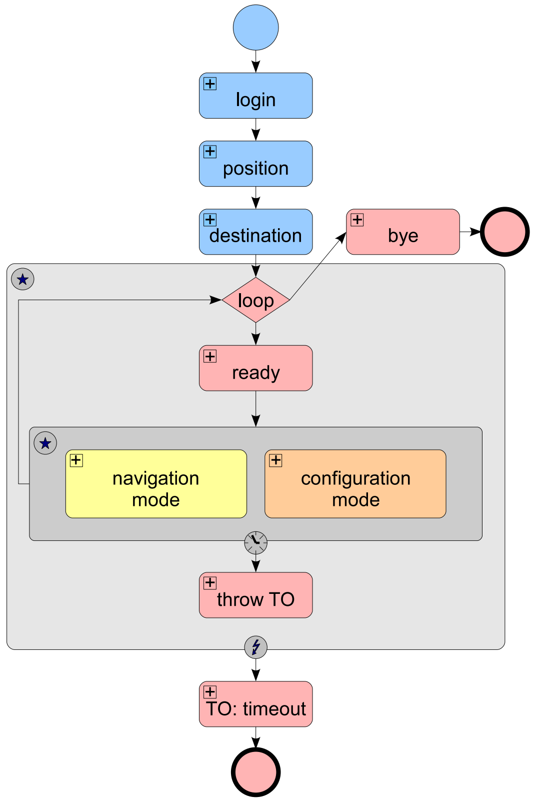


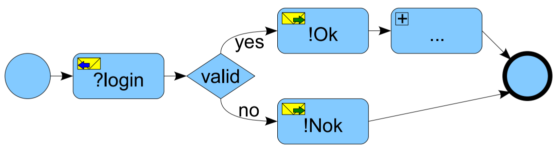

Figure 5: Functionality workflow of the GPS Web service

The behavior of the GPS Web service consists of two main phases, described in the sequel: the initialization phase (login, setting of the initial position and destination) and the main loop phase (management of the itinerary, modification of the parameters, etc.).

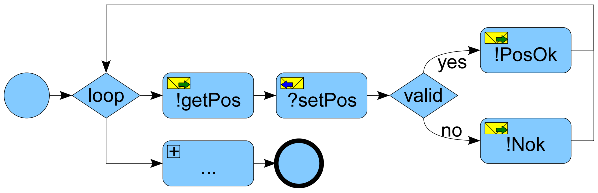
#### 3.1.1 Initialization phase

The initialization phase comprises three activities: login, position and destination.

**Login activity.** The access to the Web service is restricted to authenticated users only. To identify itself, the user must send a couple login/password, to which the service responds by a message "Ok" or "NOk" depending whether the couple is valid or not.




**Position activity.** After authentication, in order to use the main functionalities of the Web service, the user must indicate where the start location of the travel is. This is done by sending a message with information about the street, city, and country where the navigation session must be started; the message must be resent until the start location is accepted by the service (message "Ok").




**Destination activity.** Finally, before the service may attempt to calculate an itinerary, the user must enter a destination. This is similar to the position activity above: the user must retry until the end location is accepted by the service.

#### 3.1.2 Main loop phase

After the initialization phase, the service can compute an itinerary, send information about traffic, Poi, etc. To maintain the connection with the user, the service requires that the time elapsed between certain consecutive user actions does not exceed a given timeout value (a kind of "ping alive"). In Section 3.2 we will consider for analysis configurations of the system with a timeout value ranging from 1 to 60 seconds.

From the Web service point of view (see Figure 5), this timeout is managed by a "scope" process: when the timeout is reached, this process generates an exception that will be caught by another process. The main activity of this "scope" process is a "pick" process. This kind of choice enables the user to select a desired action; if we used the "switch" Bpel construct instead, the choice would be made by the Web service and not by the user. Furthermore, the "scope" is encapsulated into a "while", enabling the user to do more than one operation during the session (notice that the first action carried out by the service when entering the "while" is the emission of a *ready* message to the user). Finally, the "while" is the main activity of a second "scope", that catches the first exception thrown when the timeout is reached.

The activities executed by the main loop are partitioned in two modes, described in the sequel: the navigation mode (obtaining the itinerary, modifying the current position or destination, getting a picture or a roadmap), and the configuration mode (subscribing to a Poi, getting information on radars or traffic, setting of parameters).

**Navigation mode.** In navigation mode, the user can change the current position and the destination (using the same procedure as for the initialization phase). Next, the user can ask for the itinerary, a picture, the roadmap, or enter in configuration mode. There are two types of answer for itinerary requests: either a complete itinerary leading from the current position to the destination, with various information (about street, radar, Poi, etc.) depending on the user subscriptions, or simply a *destination* message indicating that the current position is (near) to the destination. The requests for picture and roadmap allow the user to obtain an air-picture of the area (in Png format) or a veritable roadmap (in Svg format).

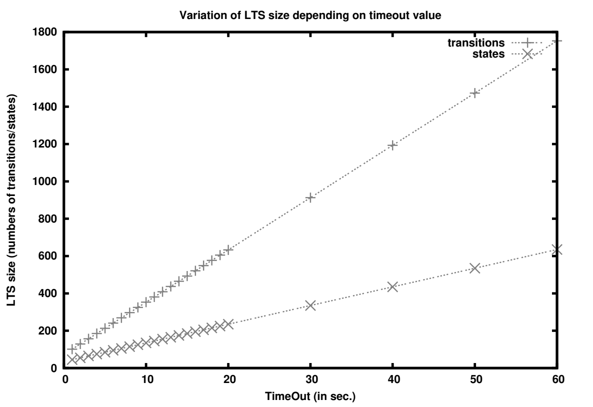
**Configuration mode.** In configuration mode, the user can subscribe or cancel his subscription to information about Poi, radar or traffic. This information is added to the itinerary if necessary. Additionally, the user can set various parameters, such as the kind of the itinerary (on motorway or not), etc.

### 3.2 Discrete-time LTS synthesis

Starting from the Bpel description of the Gps Web service, we apply the WSMod tool in order to obtain a dtLts on which the verification tools of Cadp will operate. We show below the dtLts model obtained for a timeout of 1 second, then we study its variation in size as the timeout value increases, and finally we discuss the behavior of the Web service w.r.t. the ambiguity detection feature implemented in WSMod.

**Discrete-timed labeled transition system.** dtLts models represent the observable behavior of Web services. The actions labeling the dtLts transitions denote the messages exchanged (emissions and receptions are prefixed by '!' and '?', respectively), the elapse of a discrete-time unit $\chi$ (or *time*), the internal action $\tau$ (or *tau*), and the termination action $\surd$ (or *done*). The global behavior of the Web service is obtained by an exhaustive simulation of the Bpel description driven by the Atp rules given in Table 1. The dtLts obtained in this manner for the Gps Web service with a timeout value of 1 second is shown in Figure 6.

**Variation of DTLTS size with the timeout value.** The size of the dtLts (number of states and transitions) depends on several aspects of the Bpel description: the number of Bpel processes, their complexity and nesting, the amount of communications, and the values of the timeouts. For the sake of readability, we have shown in Figure 6 the dtLtss for a timeout of 1 second (corresponding to one $\chi$ in discrete-time), but we carried out verification also for larger values of the timeout.



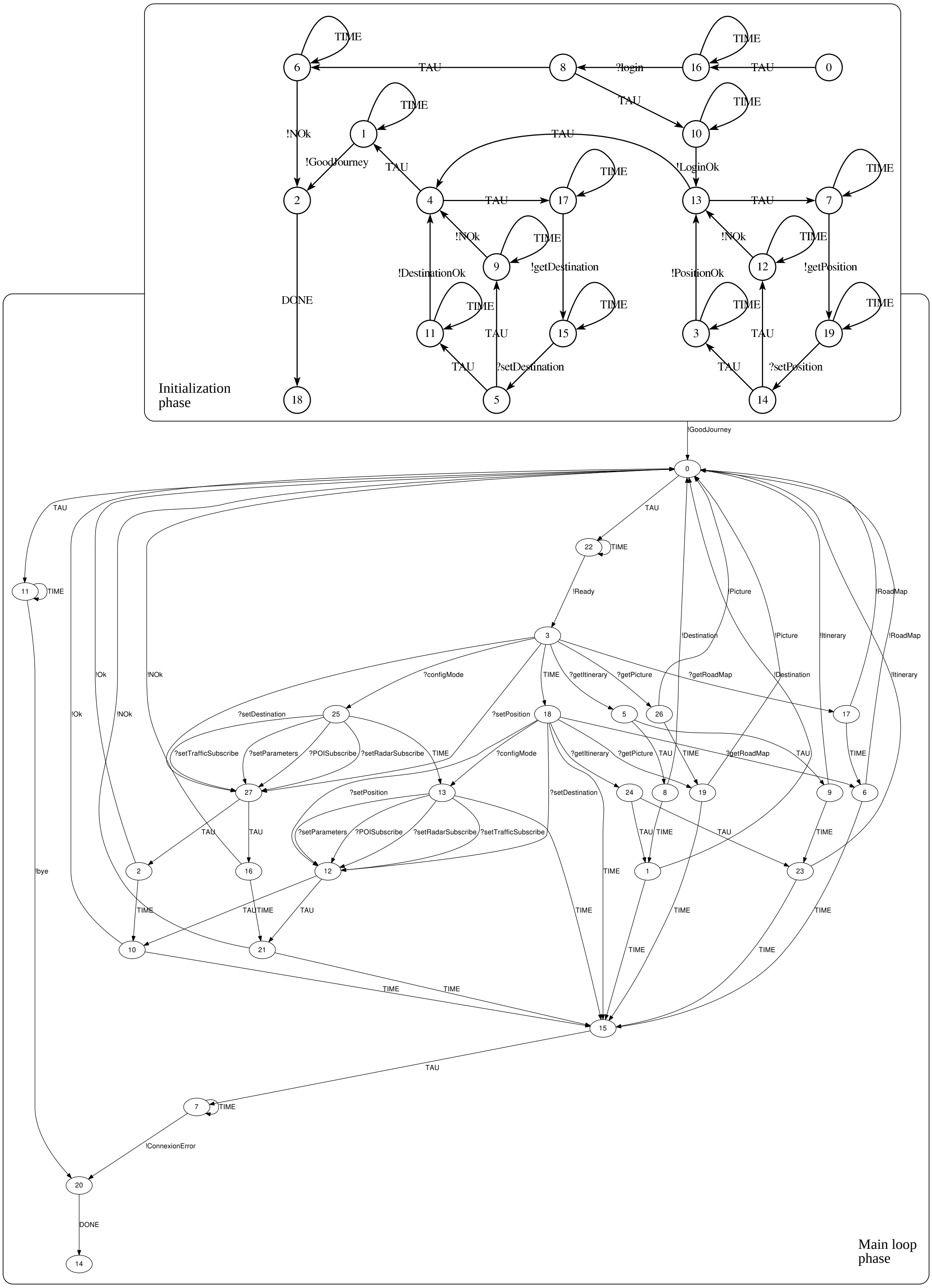


Figure 6: dtLts model of the Gps Web service, with zoom on the initialization phase. The action *!GoodJourney* makes the link between the initialization phase and the main loop phase.

The figure above shows the variation of the DTLTS size for timeout values ranging from 1 to 60. We observe a linear increase of both the number of states and transitions; this is a consequence of the fact that the BPEL description contains a single timeout (according to the ATP rules). In the presence of multiple, overlapped timeouts, the size of the DTLTS may increase much more quickly.

**Non ambiguous Web service.** In this study, we focus on the verification of the Web service behavior. However, the WSMOD tool can also synthesize automatically a DTLTS modeling the behavior of an adapted client interacting with the Web service, provided that the model of the service respects certain properties (concerning non ambiguity in message exchanges, time elapsed, etc.) detailed in [13]. Here, the GPS Web service is identified as *non ambiguous* by WSMOD, meaning that the tool can synthesize an adapted client that can know, on each message exchange, the exact choice made on the service side, and therefore the client and the service can evolve without any deadlocks or mismatches.

### 3.3 Verification of discrete-time properties

We analyze below the behavior of the GPS Web service (considering a timeout of 50 seconds) by means of discrete-time model checking using the EVALUATOR 4.0 [24] tool of CADP. Table 2 illustrates the formulation in MCL of several safety and liveness properties, of both untimed and timed nature. The colored parts of the formulas indicate discrete-time properties, which involve the counting of *time* actions. All properties were successfully verified on the corresponding DTLTS of the system, which has 535 states and 1473 transitions.

**Safety properties:** they specify informally that "something bad never happens" during the execution of the system. In the MCL language, these properties can be expressed in a concise manner by identifying the undesirable execution sequences, characterizing them using extended regular expressions, and forbidding their existence in the DTLTS model using necessity modalities.

Properties $S_1$ and $S_2$ concern the ordering of actions during the initialization phase: $S_1$ specifies that the user cannot set the position or the destination before logging in successfully, and $S_2$ states that after requesting the position or the destination, the Web service cannot begin the main loop before receiving an appropriate answer from the user. Properties $S_3$ and $S_4$ deal with the main loop phase: $S_3$ forbids the user to make another request before the current one (here, an itinerary demand) has been handled by the service, and $S_4$ states that a demand cannot be fulfilled anymore by the service after the timeout has expired.

**Liveness properties:** they specify informally that "something good eventually happens" during the execution of the system. In MCL, these properties contain diamond modalities and minimal fixed point operators for encoding the existence of certain desirable execution sequences (*potentiality*) or trees (*inevitability*) in the DTLTS.

Properties $L_1$ and $L_2$ concern the initialization phase: $L_1$ specifies that after the user has logged in, the Web service will eventually ask for the position, the destination, or end the initialization, and $L_2$ states that after the initialization was finished the service will end up in the main loop or decide to terminate the session. Properties $L_3$ and $L_4$ deal with the main loop phase: $L_3$ indicates that as long as the timeout has not expired, the service can still prompt for a user request, and $L_4$ states that an expiration of the timeout eventually interrupts the connection. The $\mathsf{AF}\ p$ operator of CTL [3] expressing the inevitable reachability of a state $p$ is defined in $\mu$-calculus as $\mu X.p \vee (\langle \mathsf{true} \rangle\, \mathsf{true} \wedge [\mathsf{true}] X)$.

## 4 Conclusion and Future Work

The design of complex business processes according to the SOA approach requires to carefully take into account the presence of concurrency, communication, and timing constraints induced by the interaction of Web services. To facilitate the design process, we propose here a tool-equipped methodology for modeling and analyzing Web services described in BPEL. We focus on the behavioral and discrete-time aspects of Web services, and rely upon the model-based verification technologies stemming from the concurrency domain. The state/transition models of BPEL Web services are produced automatically by the WSMOD tool, which implements an exhaustive simulation algorithm based on a formalization of BPEL semantics by means of process algebraic rules. The tool is able to handle both discrete and continuous time constraints; for the moment we handle only discrete-time models, which can be analyzed using the EVALUATOR 4.0 model checker [24] of the CADP toolbox [11]. Discrete-time safety and liveness properties can be concisely expressed using the data-handling facilities of the MCL language accepted as input by EVALUATOR 4.0, and particularly the extended regular expressions over transition sequences, which allow to count tick actions occurring in the model. We illustrated the verification of discrete-time properties on the example of a GPS Web service; however, most of them can be easily adapted for other business processes described in BPEL. Our methodology enables the Web service designers to carry out formal analysis on complex Web services before publishing them, and thus to improve the quality of the design process.

We plan to continue our work along several directions. Firstly, we can improve the connection between WSMOD and CADP by producing implicit DTLTSs according to the interface defined by OPEN/CÆSAR [9]. This would enable on-the-fly verification, which allows to detect errors in large systems without constructing the complete DTLTS model beforehand but exploring it in a demand-driven way. Secondly, using discrete-time models allows to directly reuse

Table 2: Safety and liveness properties of the Gps Web service (timeout of 50 sec.)

| Prop. | Mcl formula |
|---|---|
| $S_1$ | $[\ (\neg !LoginOk)^*.?setPosition \vee ?setDestination\ ]\ \mathsf{false}$ |
| $S_2$ | $[\ (\mathsf{true}^*.$<br>$((!getPosition.(\neg ?setPosition)^*) \mid (!getDestination.(\neg ?setDestination)^*))\ .$<br>$!GoodJourney)\ ]\ \mathsf{false}$ |
| $S_3$ | $[\ \mathsf{true}^*.?getItinerary.(\neg(!Itinerary \vee !Destination))^*.$<br>$(?getPicture \vee ?getRoadMap \vee ?configMode \vee$<br>$?setPosition \vee ?setDestination \vee ?getItinerary)\ ]\ \mathsf{false}$ |
| $S_4$ | $[\ \mathsf{true}^*.?getItinerary.(\neg(!Itinerary \vee !Destination))^*.$<br>$(time.(\neg(!Itinerary \vee !Destination))^*)\{51\}\ .$<br>$(!Itinerary \vee !Destination)\ ]\ \mathsf{false}$ |
| $L_1$ | $[\ \mathsf{true}^*.!LoginOk\ ]\ \mathsf{AF}\ \langle !getPosition \vee !getDestination \vee !GoodJourney \rangle\ \mathsf{true}$ |
| $L_2$ | $[\ \mathsf{true}^*.!GoodJourney.(\tau \vee time)^*\ ]\ \langle (\tau \vee time)^*.\ !Ready \vee !bye \rangle\ \mathsf{true})$ |
| $L_3$ | $[\ \mathsf{true}^*.!Ready.\ time\{\ 0\ ...\ 50\ \}\ ]$<br>$\langle \mathsf{true}^*.!Picture \vee !RoadMap \vee !Itinerary \vee !Destination \rangle\ \mathsf{true}$ |
| $L_4$ | $[\ \mathsf{true}^*.!Ready.$<br>$((\neg(!Itinerary \vee !Destination \vee !Picture \vee !RoadMap))^*.time)\{51\}\ ]$<br>$\mathsf{AF}\ \langle !ConnectionError \rangle\ \mathsf{true}$ |

the tools available for data-based temporal logics, such as Evaluator 4.0; however, this may lead to state explosion in the presence of numerous timeouts. An alternative solution would be to use continuous time models; this can be achieved by connecting the time automata produced by WSMod with the Uppaal [20] tool dedicated to the verification of continuous time models. Finally, we will extend the methodology to handle compositions of multiple Web services, following our previous work on automated client synthesis [25], but focusing on the verification of composition. For this purpose, the compositional verification techniques available in tools such as Exp.Open [19] will be certainly useful.

## REFERENCES


[1] R. Alur and D. L. Dill. A theory of timed automata. *Th. Comp. Sci.*, 126(2):183–235, 1994.

[2] A. Chirichiello and G. Salaün. Encoding abstract descriptions into executable web services: Towards a formal development. In *WI '05: Proc. of the 2005 IEEE/WIC/ACM International Conference on Web Intelligence*, pages 457–463, Washington, DC, USA, 2005. IEEE Computer Society.

[3] E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 2000.

[4] R. De Nicola and F. W. Vaandrager. Action versus State Based Logics for Transition Systems In *Lecture Notes in Computer Science vol. 469, pp. 407–419, Springer Verlag, 1990*

[5] E. A. Emerson and C-L. Lei. Efficient Model Checking for Fragments of the Propositional Mu-Calculus. In *Proc. of the 1st International Symposium on Logic in Computer Science LICS'86*, 1986.

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

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

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

[9] H. Garavel. OPEN/CÆSAR: An open software architecture for verification, simulation, and testing. *Proc. of TACAS'98*, LNCS vol. 1384, pp. 68–84, March 1998. Springer Verlag. Full version available as INRIA Research Report RR-3352.

[10] H. Garavel and F. Lang. SVL: a scripting language for compositional verification. In M. Kim, B. Chin, S. Kang, and D. Lee, editors, *Proc. of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems FORTE'2001 (Cheju Island, Korea)*, pages 377–392. IFIP, Kluwer Academic Publishers, August 2001. Full version available as INRIA Research Report RR-4223.

[11] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2006: A toolbox for the construction and anal-

ysis of distributed processes. In W. Damm and H. Hermanns, editors, *Proc. of the 19th International Conference on Computer Aided Verification CAV'2007 (Berlin, Germany)*, volume 4590 of *Lecture Notes in Computer Science*, pages 158–163. Springer Verlag, July 2007.

[12] Object Management Group. Business process modeling notation (BPMN) specification, may 2006.

[13] S. Haddad, T. Melliti, P. Moreaux, and S. Rampacek. Modelling web services interoperability. In *Proc. of the 6th Int. Conf. on Enterprise Information Systems (ICEIS04)*, Porto, Portugal, April 14–17 2004.

[14] S. Haddad, P. Moreaux, and S. Rampacek. A formal semantics and a client synthesis for a BPEL service. In *Lecture Notes in Business Information Processing, ICEIS06 Revised Selected Paper*, volume 3. Springer, 2008.

[15] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[16] ISO/IEC. LOTOS — a formal description technique based on the temporal ordering of observational behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1989.

[17] D. Jordan and J. Evdemon. Web Services Business Process Execution Language Version 2.0 - Oasis Standard, 11 april 2007.

[18] N. Josuttis. *SOA in Practice – The Art of Distributed System Design*. O'Reilly Media, City, 2007.

[19] F. Lang. EXP.OPEN 2.0: A flexible tool integrating partial order, compositional, and on-the-fly verification methods. In J. van de Pol, J. Romijn, and G. Smith, editors, *Proc. of the 5th International Conference on Integrated Formal Methods IFM'2005 (Eindhoven, The Netherlands)*, volume 3771 of *Lecture Notes in Computer Science*. Springer Verlag, 2005.

[20] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[21] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. Wiley, 1999.

[22] A. J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.

[23] R. Mateescu and M. Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *Science of Computer Programming*, 46(3):255–281, March 2003.

[24] R. Mateescu and D. Thivolle. A model checking language for concurrent value-passing systems. In J. Cuellar and T. Maibaum, editors, *Proc. of the 15th International Symposium on Formal Methods FM'2008 (Turku, Finland)*, volume 5014 of *Lecture Notes in Computer Science*. Springer Verlag, 2008.

[25] T. Melliti, C. Boutrous-Saab, and S. Rampacek. Verifying correctness of web services choreography. In *Proc. Forth IEEE European Conference on Web Services (ECOWS06)*, Zurich, Switzerland, December 4–6 2006. IEEE Computer Society Press.

[26] M. J. Morley. Safety-level communication in railway interlockings. *Science of Computer Programming*, 29(1-2):147–170, 1997.

[27] X. Nicollin and J. Sifakis. The algebra of timed processes ATP: Theory and application, 1994.

[28] G. Salaün, A. Ferrara, and A. Chirichiello. Negotiation among web services using LOTOS/CADP. In L.-J. Zhang, editor, *ECOWS*, volume 3250 of *Lecture Notes in Computer Science*, pages 198–212. Springer, 2004.

[29] G. Salaün, J. Kramer, F. Lang, and J. Magee. Translating FSP into LOTOS and networks of automata. In J. Davies, W. Schulte, and J. Song Dong, editors, *Proc. of the 6th International Conference on Integrated Formal Methods IFM'2007 (Oxford, United Kingdom)*, volume 4591 of *Lecture Notes in Computer Science*, pages 558–578. Springer Verlag, July 2007.

[30] G. Salaün and W. Serwe. Translating hardware process algebras into standard process algebras — illustration with CHP and LOTOS. In J. van de Pol, J. Romijn, and G. Smith, editors, *Proc. of the 5th International Conference on Integrated Formal Methods IFM'2005 (Eindhoven, The Netherlands)*, Lecture Notes in Computer Science. Springer Verlag, 2005.

[31] C. Stirling. *Modal and Temporal Properties of Processes*. Springer Verlag, 2001

[32] K. J. Turner. Representing and analysing composed web services using CRESS. *J. Netw. Comput. Appl.*, 30(2):541–562, 2007.