A dense time semantics for Web services specifications languages

S. Haddad[1], T. Melliti[1], P. Moreaux[1,2], S. Rampacek[2]
(1) LAMSADE, Université Paris Dauphine, Paris, France
(2) LERI-RESYCOM, Université de Reims-Champagne-Ardenne, France
{haddad,melliti,moreaux}@lamsade.dauphine.fr, sylvain.rampacek@univ-reims.fr

Abstract

The specification of a Web service, described with languages like WSFL, XLANG, BPEL4WS, is mainly the description of its observable behaviour based on the messages exchanged with the client. This behaviour is non deterministic due to the internal choices of the server. Furthermore the specification often includes timing constraints. Building applications using Web service requires controlling interaction with them. To this end, we propose a formal dense time semantical approach based on timed automata to first describe service specifications and then to build correct interacting clients when possible. The present work extends our previous discrete time approach and overcome its limitations.

1. Introduction

Web services are ``self contained, self-describing modular applications that can be published, located, and invoked across the Web" [TID00]. 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) [SOAP00], which enables communication among Web services, the "Universal Description, Discovery and Integration" (UDDI) [UDDI02], which is a registry of Web services descriptions and the "Web Services Description Language" (WSDL) [WSDL01], which provides a formal, computer-readable description of Web services. However the need for users to deal with complex applications controlled by an explicit process model led to extended services definition; these specifications are embedded by means of specification languages tags in service description languages.

Most of these specifications languages - WSFL [WSFL01], XLANG [THA01], and more recently BPEL4WS and DAML-S [ANK01] - propose a set of operators in order to describe the service in a modular way. The basic services are messages driven and the operators are related to the orchestration of activities. Despite the fact that such operators seem like the constructors of a programming language, a specification of a Web service is mainly the description of its observable behavior based on the messages exchanged with the client.

This kind of services introduces a new problem: the clients must handle, in addition to the technical aspect, an explicit management of their behavior in order to support a long running interaction with the service whose specification is discovered at run time. Thus by construction, this behavior is non deterministic due to the internal choices of the server. Furthermore the specification often includes timing constraints (e.g. detection of the withdrawal of an interaction by the client).

The overall goal of our work is to design a generic agent able to use correctly any service only based on its formal description. In a previous work [MH03, HMMR04]; we have developed such an agent starting from a discrete time semantics for Web services. In the present work, we develop an alternative approach which overcomes a limitation of our preceding method. The paper is organized as follows: we first recall briefly XLANG and our previous formalization since we follow a similar approach. Then we present the basic operational rules translating the XLANG operators. Afterwards we explain how to build the formal model of a XLANG service as a timed automaton. We present in the next section the service-client interaction relation and we indicate how to build the client automaton of a given service. Finally, the conclusion summarizes our results and gives information on work in progress.

2. A discrete time semantical approach for handling XLANG client-service interactions

2.1. Presentation of XLANG

XLANG is an XML block-structured specification which offers a set of flow control primitives in order to define the process model of the Web service. The flow control primitives organize the operation execution exactly like the different primitives that we meet in programming languages. An XLANG description is always built on one or more WSDL description which supplies a set of operations. It uses their operations as the basic elements in order to construct the processes. An XLANG process is built by applying control primitives on operations and XLANG sub processes.

Every flow control primitive represents a specific execution order model to the XLANG processes and the WSDL operations according to a specific semantic. In

addition to flow control primitives, XLANG offers a set of primitives to structure the processes organization by defining an execution context for a set of processes or transactions. We have chosen to deal with the main constructors of this language. The forgotten ones either has an unclear semantic or are not finalized. Rather than following the XML syntax of XLANG, we have chosen to delete the syntactic sugar in order to manage compact expressions. This leads to the following syntax.

The basic processes

The process *?o[m]* (which corresponds to the input operation of WSDL) consists in receiving a message of type *m*. The process *!o[m]* (which corresponds to the notification operation of WSDL) consists in sending a message of type *m*. We consider only these two types of WSDL operations. The two other types can be built with the sequence constructor (see below). The raise process *raise[e]* simply raises an exception *e* which must be handled in some way (see below the context process).

The sequence process and the empty process

The process *P;Q* executes the process *P* followed by the process *Q*. Since the operator ``;'' is associative (see the formal semantics), we safely restrict the number of operands to two processes. The empty process *empty* does nothing; it is similar to the *skip* instruction of some languages. It can also be interpreted as the neutral element for the operator ``;''.

The switch process, the while process and the all process

The process *switch[{P_i | $i \in I$}]* chooses to behave as one process among the set *{P_i | $i \in I$}*. Each branch of its execution is guarded by an internal condition denoted by a qualified name. The conditions are evaluated w.r.t. the order of their appearance in the description. However since the client has no mean to predict the choice of the service, this order is irrelevant as well as the meaning of these conditions. So we omit them. The main consequence is that from the point of view of the client, this choice is non deterministic.

The process *while[P]* iterates an inner process while an internal condition is satisfied. The remarks about the *switch* constructor are also valid for the *while* constructor.

The process *all[{P_i | $i \in I$}]* simultaneously activates a set of processes *{P_i | $i \in I$}*. XLANG does not include synchronization primitives since it considers synchronization as an internal action unobservable by the client. This parallel execution is similar to a ``fork join'' in the sense that the combined process finishes its interaction when all the sub-processes have achieved their execution.

The pick process and the context process

The pick process *pick[E]* where *$E=\{(m_i,P_i) \mid i \in I\} \cup (d,Q) \cup \{(e_j,R_j) \mid j \in J\}$* manages a condition race between sub-processes based on timing or triggers. It contains one or more event handler sub-blocks. Each event handler associates a specific service behavior to an occurrence of the corresponding event. The possible kinds of event are the reception of an expected message m_i, the triggering of a time-out whose duration is expressed w.r.t. to some time unit by an integer *d* (delay actions) or the raising of some exception e_j. When some event happens, the service behaves as the associated process (P_i, Q or R_j). The ``time'' event introduces a watchdog for reception of messages. There is at most one such event in the construction. The specification of catching processes is authorized only if the pick process is the exception part of a context process.

The context process *context(P,E)* has different roles but here we only describe the handling of exceptions. Each context contains an (optional) exception process *E* which is defined as the single parameter of a pick process. The exception process has catching sub-processes which intercept the raised exceptions during the current context execution or during a nested one if the raised event has not been previously caught. The ``time'' action of the exception block is a watchdog for the context execution delay. Similarly, canceling or aborting messages can be handled by this construction.

2.2. A discrete-time approach for XLANG

In our previous work [MH03, HMMR04] we have defined an approach to synthesize a correct client of a given Web service described in XLANG. The different steps of such an approach are the following ones:

- The selection of a timed formalism in order to describe the processes and their interactions. In the discrete case, the Timed Input/Output Transition Systems (TIOTS) is the standard choice. They allow us to express evolution of processes by actions corresponding to XLANG actions and a unit time passing.
- The design of an algorithm which translates XLANG processes into TIOTS. This transformation is based on a set of operational rules associated to each XLANG operator. For a given service, we obtain a TIOTS which is in fact its formal semantics.
- The definition of an interaction relation between two TIOTS. The standard relations like bisimulation relations or language equivalences are inappropriate for our purposes. The message nature of Web service-client interactions led us to define a non standard relation derived from the bisimulation and reflecting the fact that

sending a message is an action while receiving a message is a reaction.

- The design of an algorithm which builds the TIOTS of a client correctly (i.e. with respect to our previously defined relation) interacting with the service.

The last step of the approach is the most difficult one. It is based on a kind of determinization of the service TIOTS. Furthermore since for a service there may be no interacting client what is called ambiguity, we check during the building of the potential client the ambiguity of the service.

We have implemented this discrete time approach in our experimental platform. However there are at least two drawbacks to this solution. First, since the transitions corresponding to the time passing are mixed with the other transitions, the client transition system is hardly understandable by a user. For instance the timers associated to the *pick* and the *context* constructor have no equivalent items in the client automaton. Secondly and more importantly, the discrete time semantics causes an exponential increase of the TIOTS size w.r.t. the size of the XLANG specification. This is especially significant when several time scale activities arise in the service: we need to use the shortest time unit to describe all the delays. To overcome this important limitation we now propose a dense time approach whose steps are similar to the discrete case. However, we shall see that this requires more complex algorithms even to define the formal semantics of a XLANG process.

3. A dense time formal semantics for XLANG processes

In the dense time context, timed automata (TA) [AD94] are one of the most widely used formalism for specifying (and verifying) systems. We chose TA to model XLANG services and we synthesize a correct client as a deterministic TA interacting with the service TA.

3.1. Timed automata

A Timed Automaton (TA) is a tuple $T=(L,C,B,A,E,s_0,F)$, where L is the set of locations (or control states), C is the set of clocks, B is the set of boolean expressions built over atomic boolean propositions (see below), A is the set of actions, E is the set of edges; an edge $e = (s,g,a,r,d)$ of $L\times B\times A\times B\times L$ is defined by the source location s, the guard g, the action a, the subset r of clocks reset by e and the destination location d. s_0 and F are the initial and final locations of the TA. Atomic boolean propositions are expressed as "x *op* n" and "x-y *op* n" with x,y belonging to C and *op* a comparison operator (<, >, =); n is an natural constant. These propositions are combined with usual operators (*not*, *and*, *or*) to build boolean expressions. A variant of TA associates $I(s)$ a boolean proposition (called invariant) with each location.

The semantics of TA is defined by timed executions. At any time, each clock has a value $v(c)$; v is called a valuation of the clocks. A configuration or state of T is a pair (s,v) with $s \in L$ and $I(s)(v)=tt$ (true). The initial state is $(s_0,0)$, i.e. at time $t=0$, all clocks values are null. We denote by $v+t$ the valuation defined by $(v+t)(c)=v(c)+t$. There are two kinds of transitions in T.

- A transition $(s,v) \xrightarrow{(g,a,r)} (s',v')$ corresponds to an edge $e=(s,g,a,r,s')$ of T: This transition is possible iff $g(v)=tt$ and $I(s')(v')=tt$ where v' is defined by $v'(c)=0$ for $c \in r$ and $v'(c)=v(c)$ otherwise.

- A transition $(s,v) \xrightarrow{d(t)} (s,v+t)$ corresponds to some time passing in location s and may occur iff $\forall t' \leq t$ $I(s)(v+t')=tt$.

An execution of T is a sequence of states and transitions from the initial state to a state (s, t) where $s \in F$.

3.2. XLANG as a dense timed process algebra

In order to formalize XLANG as dense timed process algebra, we proceed in the following way:

- A process of XLANG is a state of a TA whose "untimed transitions" are deduced by a set of operational rules.
- To each *context* and *pick* operator occurring in a process of XLANG and involving a delay is associated a clock of the automaton.
- However in a process, some of these clocks are inactive. For instance in $P;Q$, a clock associated to a *context* operator occurring in Q is inactive. Thus we analyze the process in order to determine which clocks are active.
- By combining the untimed transitions and the set of active clocks, we obtain the invariant and the automaton transitions starting from the process.
- The clocks reset in a transition depend both on starting process and the destination process. Thus this information is obtained later if the destination process has not yet been analyzed.

Let us now introduce the actions of the process algebra. In contrast to discrete time, time passing in a given process is no more explicitly described by operational rules. The possible actions are message receiving ($?m$) and sending ($!m$), internal actions (τ), raise of exception ($e \in E$), expiration of timeout (to) and the termination of the process ($\surd$). We distinguish three kinds of actions: the immediate actions corresponding to a logical transition ($\tau,e,\surd$), the asynchronous actions where an unknown amount of time elapses before the occurrence of the actions ($?m,!m$) and the synchronous actions (to) which occur after a fixed delay.

3.2. The operational rules of XLANG

Here we use the standard notations with axioms and inference rules starting from the behavior of the components.

The *empty* process can only terminate.

$$empty \xrightarrow{\sqrt{}} 0$$

Here 0 is the null process disabling any action. The message transmission or reception leads to the empty process

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

Raising an exception aborts the process.

$$raise[e] \xrightarrow{e} 0$$

As will be seen in the rules associated to the context operator, the exceptions may be caught. When two processes P and Q are sequentially executed, the process P runs until termination and then Q starts:

$$\forall a \neq \sqrt{} \frac{P \xrightarrow{a} P'}{P;Q \xrightarrow{a} P';Q} and \forall a \frac{P \xrightarrow{\sqrt{}} and Q \xrightarrow{a} Q'}{P;Q \xrightarrow{a} Q'}$$

In the conditional processes *switch* and *while*, the condition is internal. Thus the switch process may evolve as any of the possible processes and the *while* process may iterate the loop or exits the loop.

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

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

The *all* process expresses parallelism. It terminates when all its components have terminated.

$$\frac{\forall i \in I, P_i \xrightarrow{\sqrt{}}}{all[\{P_i | i \in I\}] \xrightarrow{\sqrt{}} 0}$$

If some process may execute an immediate action, the *all* process evolves accordingly.

$$\forall a \in E \cup \{\tau\} \frac{\exists j \in I, P_j \xrightarrow{a} P'}{all[\{P_i | i \in I\}] \xrightarrow{a} all[\{P_i | i \in I \setminus \{j\}\} \cup \{P'\}]}$$

If no process can execute an immediate action and a process may execute an asynchronous action, then the *all* process evolves accordingly.

$$\frac{\exists j \in I, P_j \xrightarrow{*m} P' and \forall a \in E \cup \{\tau\} not \exists k \in I, P_k \xrightarrow{a}}{all[\{P_i | i \in I\}] \xrightarrow{*m} all[\{P_i | i \in I \setminus \{j\}\} \cup \{P'\}]}$$

The rules of the context process are the more complex ones. We abbreviate the *context* process by *context(P,E)* where $E=\{(m_i,P_i) \mid i \in I\} \cup (d,Q) \cup \{(e_j,R_j) \mid j \in J\}$ and m_i is a message, d a delay, e_j an exception and P_i, Q, R_j are processes.

When the main process terminates the context process terminates.

$$\frac{P \xrightarrow{\sqrt{}}}{context(P,E) \xrightarrow{\sqrt{}} 0}$$

When the main process raises an expected exception, the context process catches it (silently).

$$\frac{P \xrightarrow{e_j}}{context(P,E) \xrightarrow{\tau} R_j}$$

When the main process raises an unexpected exception, the context process propagates it.

$$\forall e \in E \setminus \{e_j | j \in J\} \frac{P \xrightarrow{e}}{context(P,E) \xrightarrow{e} 0}$$

When the main process may execute an internal action, the context process acts accordingly.

$$\frac{P \xrightarrow{\tau} P'}{context(P,E) \xrightarrow{\tau} context(P',E)}$$

The next rule requires that no immediate actions are possible in the main process. Upon reception of a canceling message, the appropriate process is activated.

$$\frac{\forall a \in E \cup \{\tau, \sqrt{}\}, not(P \xrightarrow{a})}{context(P,E) \xrightarrow{?m_i} P_i}$$

The other asynchronous actions of the main process are executed by the context process.

$$\frac{P \xrightarrow{*m} P'}{context(P,E) \xrightarrow{*m} context(P',E)}$$

The *pick* process is a special case of the context process in which the main process does nothing. So we do not explicitly give the rules which can be deduced from the previous ones.

3.3. Search of active clocks

We recall that to each context operator is associated a clock. However not every clock is active in a process. The algorithm given below analyses the syntactical tree of the process in order to search these active clock.

```
begin activeclocks
switch root(T) of
    case empty: return
    case *o[m]: return
    case raise: return
    case switch: return
    case while: return
    case sequence (;):
        activeclocks(leftsubtree(T))
        return
    case all:
        foreach subtree(T) do
            activeclocks(subtree(T))
        endforeach
        return
    case context:
        If a delay occurs in the context
        then
              mark the clock associated
              to this context as active
        endif
        activeclocks(leftsubtree(T))
        return
endswitch
end activeclocks
```

We explain the main cases. In the sequence operator, we look for active clocks in the left term which is the current active process. In the *all* process, the active clocks are searched in all the sub processes. At last, if the *context* process owns a delay construction then the associated clock is active.

3.4. Building the edges of the TA

Let us recall that an edge of the automaton will be a tuple *e=(P, g, a, r, P′)* where *P* and *P′* are the source and destination processes, g is the guard of *e* and *r* is the subset of clocks to be reset. In order to build the edges out of *P*, we first determine the possible actions from *P*, noting that we have two kinds of actions: effective transitions of *P* and timeout actions. Then we compute the possible edges taking into account the operational rules and the time semantics of XLANG to define *g* for actions depending on their kind. The set *r* of clocks to be reset is computed during the study of *P′* and is exactly the set of clocks active in *P′* and not active in *P*.

If P may execute an immediate action *a* and becomes *P′* (via the operational rules) then the corresponding edge is $P \xrightarrow{(tt,a,r)} P'$.

If *P* cannot execute immediate actions and may execute an asynchronous action *a* (either *?m* or *!m*) and become *P′* then the corresponding edge $P \xrightarrow{(g,a,r)} P'$ has for guard the conjunction of conditions $(c < d_c)$ for all active clocks *c* with d_c being the value of the delay.

For timeout expirations, the definition of the transitions and particularly the associated guard is more involved. Indeed some timeouts may be taken into account simultaneously (like two timeouts in sub processes of an *all* process) whereas some timeout inhibits other ones (like two nested context where in case of simultaneous expirations, only the nesting context has to be taken into account). Thus while searching the active clocks of a process, we build a forest whose nodes are the active clocks and which expresses a partial order between clocks. We then consider each non ordered (w.r.t. the forest structure) subset *C′* of active clocks as triggering a potential simultaneous expiration to be taken into account. Let *Succ(C')* be the active clocks which are greater (w.r.t. the partial order) than *C′*. Then we have an edge $P \xrightarrow{(g,to(C'),r)} P'$ with *g* being the conjunction of the three terms: a conjunction of the conditions $(c = d_c)$ for $c \in C'$, a conjunction of the conditions $(c \leq d_c)$ for $c \in Succ(C')$, and a conjunction of the conditions $(c < d_c)$ for *c* being any other active clock. This guard expresses that "dominated" clocks may expire at the same time as the clocks of *C′*. We do not detail the building of *P′* in this case due to lack of space.

4. Interaction relation and client timed automaton synthesis

When the TA of a service has been built, we try to obtain a correct client for this service. Since this client program communicates by means of message exchanges and must manage timeouts, we represent it as a TA. As for the discrete case, this requires first to define the interaction relation between TA. We then derive an algorithm to determine if a service TA accepts a deterministic interacting TA or is ambiguous. In the present paper, we only give an intuitive presentation of the interaction relation and the construction of the client TA.

4.1. Client-service interaction relation

During the interaction, the client may send a message from the set of messages expected by the service; conversely, every expected message by the client corresponds to a message which may be sent by the server, based on previously exchanged messages; finally the client must be able to detect service termination. If such a client program does not exist, we say that the service is ambiguous. For instance, the process *P=while[!o[a]]* is ambiguous since the client is not able to determine how many messages it will receive before the service termination.

Classical comparison relations between timed systems are language equality and bisimulation. Equality between observable languages does not take intermediate states into account and the nature of the different actions does not matter with bisimulation. We have then defined a relation similar to the bisimulation relation but taking care of the type of actions. Note that since the semantics of TA is defined by their timed transition systems, the interaction relation has to be defined between these timed transitions systems. The interaction relation definition is obtained in three steps.

At first, since the relation abstracts from the internal actions, we define the derivations abstracting from internal actions: if *a* is an action, then $s \Rightarrow_a s'$ means that the process evolves from *s* to *s'* by first executing (possibly several or zero) invisible (τ) actions, then *a* and again τ action(s). For time passing action *d(t)*, $s \Rightarrow_{d(t)} s'$ means that the process evolves from *s* to *s'* with internal actions interleaved with time passing, and with a total time passing of *t*.

Second, for each possible a action we define its "complementary" action ${}^c a$ which will be used in the definition of "equivalent" states. We have ${}^c!m=?m$, ${}^c?m=!m$ and ${}^c a=a$ otherwise.

Finally, TA *Q* and *R* interact if to each state of a TA *P* able to do action *a*, except for receiving a message, its "equivalent state" of the interacting TA *Q* is able to do ${}^c a$. The reception action is a special case specific to our interaction relation: suppose that one state of *R* expects the reception of *m*, does it mean that *Q* is able to send this message? The answer is not necessary since *Q* may evolve in an indistinguishable way from one state to two states, one where it is able to send *m* and the other one where it is not. However we require that in the other state, it is able to send a message in order to avoid an infinite waiting of *P*.

4.2. Construction of the client timed automaton

The core of the construction of the interacting TA of a service is based, as for the discrete time, on a kind of automaton determinization of the service. However it is well-known, in contrast with discrete time transition

systems, that the non deterministic timed automata are strictly more expressive than the deterministic ones. Thus we have designed an original procedure which decides whether a specification is (potentially) ambiguous and in the negative case produces such a deterministic automaton. A node of the deterministic automaton is a set of nodes of the original automaton. Similarly to the approaches which determinize subclasses of timed automata [AFH94], we require that the deterministic automaton has the same clocks as those of the original automaton. Then, we inspect edges of the service automaton and we define edges and guards of the deterministic automaton, when possible, based on the guards of the starting automaton. Such an approach has been successful due to the nature of the timed automata that we generate in the first stage of the method.

5. Conclusion and future work

The interactions with Web service processes require a formal specification framework. Extending a previous work in discrete time, we have proposed in this paper dense time semantics for specification languages of Web services like XLANG: from the definition of a service - a process- we construct a timed automaton corresponding to its formal semantics in dense time. Then we have explained how to define an interaction relation between client and service at the timed automata level. We have also described the guidelines of the algorithm synthesizing a deterministic client timed automaton when this is possible and detecting ambiguous service otherwise.

We are currently implementing these algorithms in our framework ICIS and they will be soon reachable from the net. Moreover with a slight adaptation of this work, we plane to build a platform for defining composite services from elementary ones which could be directly executed without any new implementation.

References

[AD94] R. Alur and D. L. Dill, "A Theory of Timed Automata", *Theoretical Computer Science,* 126, pp. 193-235, 1994.

[AFH94] R. Alur, L. Fix, T. and A. Henzinger "*Event-Clock Automata: A Determinizable Class of Timed Automata*", Lecture Notes in Computer Science 818, pp. 1-13, 1994.

[ANK01] Ankolekar, A. and al, "Daml-s : Semantic markup for web services", *Proceedings of the International Semantic Web Working Symposium (SWWS)*, Standford, USA, 2001.

[WSFL01] Bechhofer, S. and al, "Web services flow language (wsfl 1.0) ", *Technical report, IBM Corporation*, 2001 (http://www4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf).

[HMMR04] S. Haddad, T. Melliti, P. Moreaux and S. Rampacek, "Modelling WEB services interoperability", *Proc. of the 6th Int. Conf. on Enterprise Information Systems (ICEIS04)*, Porto, Portugal, Apr. 14-17, 2004 (to appear).

[MH03] T. Melliti and S. Haddad, "Synthesis of Agents for Web Services Interaction", *International Conference Electronic Commerce, Workshop on Semantic Web Services for Enterprise Application Integration and E-Commerce*, Pittsburgh, Sept. 2003.

[UDDI02] OASIS, "Universal description, discovery and integration", *Technical report*, OASIS UDDI Specification Technical Committee, 2002 (http://www.oasis-open.org/cover/uddi.html).

[THA01] Thatte, S, "XLANG: Web services for business process design", *World Wide Web page*, 2001 (http://www.gotdotnet.com/team/xml/wsspecs/xlangc/default.htm.

[TID00] Tidwell, D., "Web services - the web's next revolution", *IBM developerWorks*, 2000.

[SOAP00] W3C, "Simple object access protocol (soap) 1.1", *Technical report*, World Wide Web Consortium (http://www.w3.org/TR/SOAP).

[WSDL01] W3C, "Web services description language (wsdl) 1.1", *Technical report*, World Wide Web Consortium, 2001, (http://www.w3.org/TR/wsdl).