Synthesis of Agents for Web Services Interaction

Tarak Melliti
LAMSADE, UMR CNRS 7024,
Place du Maréchal de Lattre de Tassigny, 75775
PARIS,FRANCE
melliti@lamsade.dauphine.fr

Serge Haddad
LAMSADE, UMR CNRS 7024,
Place du Maréchal de Lattre de Tassigny, 75775
PARIS,FRANCE
haddad@lamsade.dauphine.fr

ABSTRACT

With the development of the semantic Web, the specification of Web services has evolved from a "remote procedure call" style to a behavioral description including standard constructors of programming languages. Such a transformation introduces new problems since traditional clients will not be able to interact with these sophisticated services. In this work, we develop a generic agent capable to fully control the interaction process with a Web service given its XLANG behavioral description (XLANG being one of these languages). At first, we give an operational semantic to XLANG in terms of timed transition systems. Then we define a relation between two communicating systems which formalizes the concept of a correct interaction. Finally we propose an algorithm which either detects ambiguity of the Web service or generates a timed deterministic automaton which controls the agent behavior during the interaction with the service.

Keywords

process model, interaction control, client synthesis

1. INTRODUCTION

Web services are "self contained, self-describing modular applications that can be published, located, and invoked across the Web" [4]. 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: the "Simple Object Access Protocol" (SOAP)[10], which enables communication among Web Services, the "Universal Description, Discovery and Integration" (UDDI)[12], which is a registry of Web Services descriptions and the "Web Services Description Language" (WSDL)[11], 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 [3]. 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 and the port address, etc.

However simple operation invocation is not sufficient for some kind of applications. They require in addition a long-running interaction derived by an explicit process model. To deal with this lack, some specification languages have been introduced in order to describe the behavioral aspect of the Web services process such as WSFL [1], XLANG [9], WSCL [13] and more recently BPML4WS. Each of them is directly based on top of WSDL. They propose different schema to glue such services operations according to a process model. The difference between WSDL description and the behavioral description lies in considering the application state. Within WSDL, services are stateless i.e. they consider only the in-between operation states. However technologies supporting a Web services composition language must handle more complex transactions than a simple request-response model.

Such a transformation introduces new problems since traditional clients will not be able to interact with these sophisticated services. In this work, we develop a generic agent capable to fully control the interaction process with a Web service given its specification. Since time is an important parameter of a Web and XLANG includes explicit control of delays and deadlines, we have chosen it for the specification language. It should be clear to that we could easily adapt our work to similar languages.

Since our goal is to produce a client behaviour which correctly interacts with the service, we have to formally define such an interaction. But this requires beforehand to specify what is a behaviour. Thus we give an operational semantic to an XLANG 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 behavior of some components, a rule specifies a possible transition of a service built via these constructor by these components. Since XLANG does specify neither how the service is implemented nor the context of its execution, 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. Let us briefly explain why the two main ones - the language equivalence and the bisimulation equivalence - do not match our needs. The language equivalence is unable to express the different branching capabilities of the systems (e.g. an immediate choice versus a delayed one) since it does not require an equivalence relation between the intermediate states of the two systems. The bisimulation equivalence does not take into account the different nature of the event: in an asynchronous communicating system the sending of a message is an action whereas the reception is a reaction.Thus the interaction relation that we introduce can be viewed as a bisimulation relation modified in order to capture the nature of the events.

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 appears that some XLANG specifications do not admit such a client i.e. they are inherently ambiguous. Thus the algorithm we develop either detects ambiguity of the Web service or generates a deterministic automaton satisfying the interaction relation. The core of our algorithm is a kind of determinization of the transition system of the service. Thus the size of the generated automaton may be exponential w.r.t. the size of the transition system. This exponential blow-up is unavoidable since the problem of determinizing an automaton reduces to our problem.

The balance of the paper is the following one. In the next section, we introduce the main construction of XLANG with a simplified syntax and we give a formal semantic for this language. The third section is devoted to the interaction relation between two communicating transition systems and to the synthesis algorithm. At last, we conclude and give some perspectives to this work.

2. XLANG: SYNTAX AND SEMANTIC

2.1 Informal description of XLANG with a simplified syntax

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 subprocesses. 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 primitive 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. An XLANG process is recursively defined in the following subsections.

2.1.1 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 $r[e]$ simply raises an exception e which must be handled in some way (see below the context process).

2.1.2 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 ";".

2.1.3 The switch process, the while process and the all process

The process $switch(\{c_i, P_i\}_{i\in I})$ chooses to behave as one process among the set $\{P_i\}$.

Each branch of its execution is guarded by an **internal** condition denoted by a qualified name (c_i). 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. The main consequence is that from the point of view of the client, this choice is non deterministic.

The process $while(c, P)$ iterates an inner process while an **internal** condition c is satisfied. The remarks about the choice constructor are also valid for the while constructor.

The process $all(\{P_i\}_{i\in I})$ simultaneously activates a set of processes $\{P_i\}$. 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.

2.1.4 The pick process and the context process

The pick process $pick[\{m_i, P_i\}_{i\in I}, \{d, Q\}, \{e_j, R_j\}_{j\in J}]$ 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 occurence 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 $[P, E]$ has different roles but here we only describe the handling of exceptions. Each context contains an (optional) exception process E which is 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 catched. The "time" action of the exception block is a watchdog for the context execution delay. Similarly, cancelling or aborting messages can be handled by this construction.

2.2 A formal semantic for XLANG

XLANG provides a set of operators describing in a modular way the observable behavior of a Web service. In fact, this approach is close to the process algebra paradigm illustrated for instance by CCS [7], CSP [5] and ACP [2]. The main objective of the process algebra approach is to cope with the complexity of the conception of parallel systems. In order to achieve this goal, the theoretical developments related to a process algebra generally consists in four steps[6]. At first one defines a set of operators and syntactic rules for constructing processes (e.g. what we have done in the previous subsection). Then one associates to each operator a set of semantic rules which assign to a process a behavioural interpretation; In order to compare different processes, one introduces some equivalence relations and congruences which express that two processes (or components) have a similar behaviour w.r.t. to different criteria. At last one develops algorithms which decide the equivalence of two processes, working at the syntactical level (e.g via a set of algebraic laws) or at the semantical level (e.g. with techniques like model-checking).

Since time is an important issue in such systems, the process algebra model has been enlarged by introducing (discrete or dense) time passing as a special transition [8]. Thus it appears that the syntactic features of XLANG make it a good candidate to be an algebra of timed processes. We have chosen to represent time passing by units for the following reasons. The time constraints of a Web service are generally "soft" thus the discretization of time is a valid abstraction. In the sequel, we will complete the XLANG algebra with an operational semantic as the first step for the development of our generic agent. Beforehand, we give the elements necessary to this semantic.

A labelled transition system is an oriented graph where the nodes represent the possible states of the system (with an initial state) and the arcs represent the state transitions. Each arc is labelled by the action whose occurence has triggered this transition. Depending on the process algebra language, some labels have a special meaning. We will detail our alphabet later.

Definition 1. A labelled transition system LTS is defined a tuple $LTS = (S, L, \rightarrow, s_0)$ where:

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

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

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

We describe now the events of a LTS associated to an XLANG specification:

- The set of types of messages will be denoted M. There are two events associated to a message m: the emission denoted by $!m$ and the reception denoted by $?m$. We also denote $!M = \{!m \,|\, m \in M\}$ and $?M = \{?m \,|\, m \in M\}$ and the joker character $*$ may be substituted by ! or ?.
- Since the service may evolve in an unobservable way (e.g. the evaluation of a condition) we introduce τ, the internal action.
- Since XLANG takes into account the time, χ denotes one unit time passing.
- The exception event set of XLANG is denoted by E.
- In order to control that the client correctly detects the end of the service, we introduce $\surd$ the termination event. This action will also simplify the definition of the operational semantic.

Since we consider that due to internal or external conditions, any basic action of a process can be delayed, the behaviour of the basic processes is specified by the following rules:

$$*o[m] \xrightarrow{\chi} *o[m] \text{ and } *o[m] \xrightarrow{*o[m]} empty \text{ where } * \in \{!, ?\}$$
$$r[e] \xrightarrow{\chi} r[e] \text{ and } r[e] \xrightarrow{e} empty$$

The *empty* process is different from the null process 0. After some units of time it indicates its termination and then becomes 0.

$$empty \xrightarrow{\chi} empty \text{ and } empty \xrightarrow{\surd} 0$$

The *sequence* process acts at its first subprocess while this process does not indicate its termination. In the latter case, the *sequence* process becomes the second process in a silent way.

$$\frac{P \xrightarrow{a} P' \wedge \neg P \xrightarrow{\surd} P''}{P;Q \xrightarrow{a} P';Q} \text{ where } a \neq \surd \text{ and } \frac{P \xrightarrow{\surd} P'}{P;Q \xrightarrow{\tau} Q}$$

The *switch* process becomes one of its sub-processes 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 sub-processes.

$$\forall i \in I \; switch(\{c_i, P_i\}_{i\in I}) \xrightarrow{\tau} P_i$$

Like the *switch* process, the *while* process evaluates in a silent way its condition. Thus we have two rules depending on this internal evaluation.

$$while(c,P) \xrightarrow{\tau} P; while(c,P) \text{ and } while(c,P) \xrightarrow{\tau} empty$$

The subprocesses of a *all* process act independently except for two actions. They simultaneously let pass a unit of time and they simultaneously indicate their termination. In the latter case, the *all* process becomes the null process.

$$\frac{\exists j \in I \; P_j \xrightarrow{a} P'}{all(\{P_i\}_{i\in I}) \xrightarrow{a} all(\{P_i\}_{i\in I\setminus\{j\}} \cup P')} \text{ where } a \notin \{\chi, \surd\}$$

$$\frac{\forall i \in I \; P_i \xrightarrow{\chi} P_i'}{all(\{P_i\}_{i\in I}) \xrightarrow{a} all(\{P_i'\}_{i\in I})} \text{ and } \frac{\forall i \in I \; P_i \xrightarrow{\surd} P_i'}{all(\{P_i\}_{i\in I}) \xrightarrow{\surd} 0}$$

Due to space considerations, we only give a semantic to the *context* process and then we will informally explain the semantic of a *pick* process when it is not an exception process. In the following rules, E the *exception* process is an abbreviation for the process $pick(\{m_i, P_i\}_{i\in I}, \{d, Q\}, \{e_j, R_j\}_{j\in J})$.

This rule expresses that the exception block may be triggered by the reception of the expected messages

$$[P,E] \xrightarrow{?m_i} P_i$$

The next rules specify that the time elapses under the control of the watchdog.

$$\frac{P \xrightarrow{\chi} P'}{[P,E] \xrightarrow{\chi} [P', pick(\{m_i, P_i\}_{i\in I}, \{d-1, Q\}, \{e_j, R_j\}_{j\in J})]} \text{ where } d > 1$$

$$\frac{P \xrightarrow{\chi} P'}{[P,E] \xrightarrow{\chi} Q} \text{ where } d = 1$$

Let us note that we can adapt the three previous rules in order to determine the behaviour of a *pick* process which is not an *exception* process. The adaptation consists in deleting the conditions related to P. The following two rules handle the case of a raised exception depending on whether the *context* process has a sub-process to handle this exception. If it is not the case the exception is "transmitted" to the including context block.

$$\frac{P \xrightarrow{e_j} P'}{[P,E] \xrightarrow{\tau} R_j} \text{ and } \frac{P \xrightarrow{e} P'}{[P,E] \xrightarrow{e} empty} \text{ where } e \in E\setminus\{e_j\}_{j\in J}$$

The last rules describe the actions of P inside the *context*.

$$\frac{P \xrightarrow{\surd} P'}{[P,E] \xrightarrow{\surd} 0} \text{ and } \frac{P \xrightarrow{a} P'}{[P,E] \xrightarrow{a} [P',E]} \text{ where } a \notin E \cup \{?m_i\}_{i\in I} \cup \{\chi, \surd\}$$

3. AGENT SYNTHESIS

Using the previous rules, starting from a XLANG specification we develop the LTS related to its behaviour. Although we will not prove it here, this LTS has a finite number of states.

We want to specify the behaviour of an agent able to correctly interact with the service. Obviously we choose the formalism of the labelled transition systems for the representation of this behaviour. We remark that this LTS must be **deterministic** in order to be implementable.

Now we proceed in two steps. At first, we need to formally define what is a correct interaction between two LTS. Once this relation is defined, we develop an algorithm producing the LTS of the client behaviour if such a behaviour exists or detecting the ambiguity of the Web service.

3.1 An interaction relation for Web services

As usual in the LTS formalism, we define an observable transition relation between states given by $s \stackrel{a}{\Rightarrow} s'$ iff $s \stackrel{\tau^* a \tau^*}{\longrightarrow} s'$ and $s \stackrel{\epsilon}{\Rightarrow} s'$ iff $s \stackrel{\tau^*}{\longrightarrow} s'$. Moreover we suppose that the exception events are not observable in the LTS of the service. If it is the case, it means that the service does not catch an exception and then must be modified.

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

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

We introduce the following notation $?m^c = !m$, $!m^c = ?m$ and $\forall a \notin \{!m\}_{m\in M} \cup \{?m\}_{m\in M} \; a^c = a$.

Definition 2. Let $LTS_1 = (S_1, L_1, \rightarrow_1, s_{01})$ and $LTS_2 = (S_2, L_2, \rightarrow_2, s_{02})$ be two labelled transition systems. Then S_1 and S_2 correctly interact iff $\exists \sim \subseteq S_1 \times S_2$ such that:

- $s_{01} \sim s_{02}$
- $\forall s_1, s_2$ such that $s_1 \sim s_2$
  - Let $a \notin \{?m\}_{m\in M}$, if $\exists s_1 \stackrel{a}{\Longrightarrow}_1 s_1'$ then $\exists s_2 \stackrel{a^c}{\Longrightarrow}_2 s_2'$ with $s_1' \sim s_2'$ and if $\exists s_2 \stackrel{a}{\Longrightarrow}_2 s_2'$ then $\exists s_1 \stackrel{a^c}{\Longrightarrow}_1 s_1'$ with $s_1' \sim s_2'$
  - Let $m \in M$, if $s_1 \stackrel{?m}{\Longrightarrow}_1 s_1'$ then

* $\exists s_2^- \overset{w}{\Longrightarrow}_2 s_2$, $\exists s_2^- \overset{w}{\Longrightarrow}_2 s_2^+$, $\exists s_2^+ \overset{!m}{\Longrightarrow}_2 s_2'$ with $s_1 \sim s_2^+$ and $s_1' \sim s_2'$ where w is a word
* $\exists s_2 \overset{!m'}{\Longrightarrow}_2 s_2'$

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

3.2 The synthesis algorithm

The algorithm builds the deterministic LTS following a kind of determinization of the LTS of the service. Each state of the potential client is associated to a subset of states of the service. There is a stack of couples (s_1, S_2') to be processed. Let us describe one step of the algorithm.

- At first, one completes S_2' with $\overset{\epsilon}{\Longrightarrow}_2$ transitions.
- If a state of the client s_1' is already associated to S_2' then one redirects all the input edges of s_1 to s_1' and one deletes s_1.
- Otherwise for each $s_2 \overset{a}{\Longrightarrow}_2 s_2'$ with a and $s_2 \in S_2'$, one builds a new vertex s_1' and a new edge $s_1 \overset{a^c}{\Longrightarrow}_1 s_1'$ and one stacks (s_1', S_2^*) where $S_2^* = \{s_2' | \exists s_2 \in S_2', \exists s_2 \overset{a}{\Longrightarrow}_2 s_2'\}$
- Let $a \notin \{?m\}_{m \in M}$ such that $s_1 \overset{a}{\Longrightarrow}_1 s_1'$, if there is a $s_2 \in S_2'$ with $\nexists s_2 \overset{a^c}{\Longrightarrow}_2 s_2'$ then stop and return "service ambiguous".
- Let $s_1 \overset{?m}{\Longrightarrow}_1 s_1'$, if there is a $s_2 \in S_2'$ with $\nexists s_2 \overset{!m'}{\Longrightarrow}_2 s_2'$ then stop and return "service ambiguous".

The algorithm starts with the couple (s_{01}, s_{02}) in the stack and stops either when the stack is empty (i.e. the client has been built) or when it has detected the ambiguity of the service. Again due to space considerations, we do not include the correctness proof of the algorithm.

3.3 An example

in the following we present an XLANG service of selling tickets planes. The WSDL service is composed by the following operations:

- $?con[req], !conRes[res]$: It represents an operation for requesting the agency database, the request (req) contain all the information concerning the quest such as the dates, destination, number of persons etc.. $!cons[rep]$ is an output operation which send the result.
- $?res[req]$: It represents a reservation operation. The message (req) contain the identifier of the choosing proposition (given by the $con[res]$).
- $!resC[rep]$: It represents the service confirmation to the client reservation.
- $!cnd[rep]$: It represents a possible response to the reservation request informing the client that the choice is any more available.
- $!timeOut[res]$: It's a timeout message over the user choice process.

the service behavior corresponding to the following process model described using our notation of the XLANG primitives:
$service ::\equiv ?cons[req]; [while(possible, !con[rep]; pick(\{?res[req], switch(\{disp, !conf[rep]\}\{n_d isp, !cna[rep]; ?cons[res]\})\}, 2, !timeOut[res]))!end[rep]$

the services is instantiated after a client request. it return the response and wait the user 2 time unit to make his choice. when receiving the user choice the service check for the user choice availability; if the choice is still available then the service send a confirmation message and ends the interaction else it informs the user and invite him to request the database. in the timeout case the service sends a timeout message and re-sends the last request response to the user to remake his choice. after n failure attempts the service cancel the while process and send en end message to user. The figure1 represents the service TIOTS generated according the operators semantics while figure2 represents it's correct client according the algorithm.

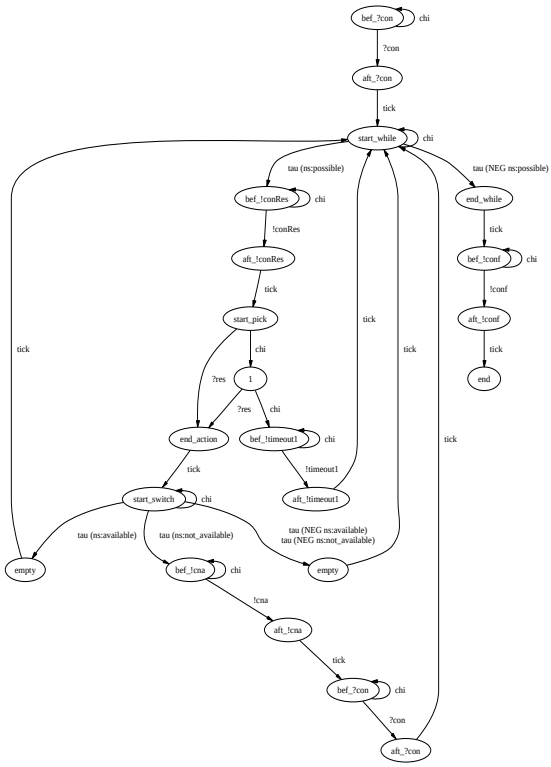

Figure 1: the service TIOTS

4. CONCLUSION

In this work, we have developed a generic agent capable to fully control the interaction process with a Web service given its XLANG behavioral description. In order to achieve this goal, we have given an operational semantic to XLANG in terms of timed transition systems, then we have defined a relation between two communicating systems formalizing the

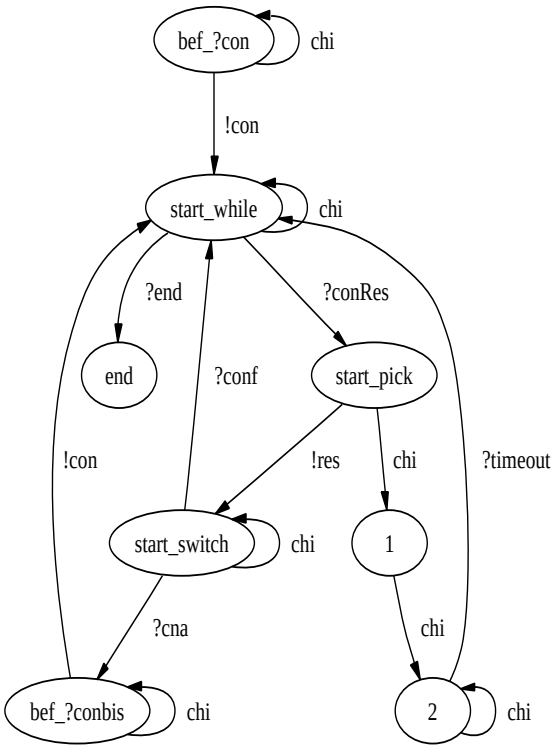

Figure 2: the generated client TIOTS

concept of a correct interaction and finally we have proposed an algorithm which either detects ambiguity of the Web service or generates a timed deterministic automaton which controls the agent behavior during the interaction with the service.

We are currently implementing this generic client. From a theoretical point of view, we are investigating two issues. On the one hand, we have implicitly supposed a perfect communication between the client and the service; thus we want to generalize our work by taking into account the properties of the medium. On the other hand, a client of a service may be a service itself; thus we are looking for a dynamic service specification which could be automatically modified depending on the publication of new services.

5. REFERENCES

[1] S. Bechhofer and al. Web services flow language (wsfl 1.0). Technical report, IBM Corporation, may 2001.

[2] J. bergstra and J. Klop. Algebre of communicating processes. Technical report, Center of Mathimatics and Computer Scienc Departement OF Computer Science, 1985.

[3] P. CaudWell and al. *Service Web XML Professionnel*. wrax, Paris, dec 2001.

[4] T. Doug. Web services - the web's next revolution. *IBM developerWorks*, nov 2000.

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

[6] I. Lee and al. A process algebraic approach to the specification and analysis of resource-bound real-time systems. Technical report, DARPA/NSF, 1993.

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

[8] X. Nicollin and J. Sifakis. The algebra of timed process, atp: Theory and application. Technical report, Information and Computation, 1994.

[9] S. Thatte. Xlang: Web services for business process design. World Wide Web page, may 2001.

[10] Simple object access protocol (soap) 1.1. Technical report, World Wide Web Consortium, may 2000.

[11] Web services description language (wsdl) 1.1. Technical report, World Wide Web Consortium, mar 2001.

[12] Universal description, discovery and integration. Technical report, OASIS UDDI Specification Technical Committee, mar 2002.

[13] Web services conversation language (wscl) 1.0. Technical report, World Wide Web Consortium, mar 2002.