Client Synthesis for Aspect Oriented Web Services

Mehdi Ben Hmida[1], Serge Haddad[2]

[1] LAMSADE, CNRS & Université Paris-Dauphine, France
[2] LSV, CNRS & ENS Cachan, France
serge.haddad@lsv.ens-cachan.fr,
mehdi.benhmida@lamsade.dauphine.fr

Abstract. Client synthesis for complex Web services is a critical and still open topic as it will enable more flexibility in the deployment of such services. In previous works, our team has developed a theoretical framework based on process algebra that has led to algorithms and tools for the client interaction. Here, we show how to generalise our approach for aspect oriented Web services.

1 Introduction

From elementary Web services to complex ones. Web services are "self contained, self-describing modular applications that can be published, located, and invoked across the Web" [23]. They are based on a set of independent open platform standards to reach a high level of acceptance. Web services framework is divided into three areas: communication protocol, service discovery and service description. The "Web Services Description Language" (WSDL) [27] provides a formal, computer-readable description of Web services. Such a description specifies the software component interfaces listing the collection of operations that are network accessible through standard XML messaging. It includes all information that an application needs to invoke such as the message structure, the response structure and some binding information like the transport protocol, the port address, etc.

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

In order to deal with the behavioural aspects of complex services, some industrial and academic specifications languages have been introduced. Among them, "Business Process Execution Language for Web Services" (BPEL4WS or more succinctly BPEL) has been proposed by leading actors of industry (BEA, IBM, and Microsoft) and has quickly become a standard [1].

The two facets of complex Web services. BPEL supports two different types of business processes (see for instance [16], [17]):

– Executable processes specify the exact details of business processes. They can be executed by an orchestration engine.
– Abstract business protocols specify the public message exchange between the client and the service. They do not include the internal details of process flows but are required in order for the client to correctly interact with the service.

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

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

Adaptation and Web services. Aspect oriented programming (AOP) helps the programmer to isolate non functional software (like authentication and logging) from business software. Using AOP eases the modification of implemented policies as it does not impact the functional part. However in the context of JAVA, it requires either to change the compiler, the loader or the virtual machine. AOP is also desirable for Web services since they require a lot of non functional codes but the integration of AOP in a Web service framework raises significant difficulties.

Previous contributions. In our previous works, we have addressed both service adaptability and client interaction issues but separately.

First, we have specified what is an external behaviour, i.e. we have given an operational semantics to an abstract BPEL specification in terms of a time transition system [11, 12]. The semantics is obtained by a set of rules in a modular way. Given a constructor of the language and the behaviour of some components, a rule specifies a possible transition of a service built via this constructor applied on these components. As previously discussed, the transition system is generally non deterministic. Then we have defined a relation between two communicating systems which formalizes the concept of a correct interaction. There are standard relations between dynamic systems like the language equivalence and the bisimulation equivalence but none of them matches our needs. Thus we have introduced the interaction relation which can be viewed as a bisimulation relation modified in order to capture the nature of the events (i.e. the sending of a message is an action whereas the reception is a reaction). Afterwards we have focused on the synthesis of a client which is in an interaction relation with the transition system corresponding to the system. The client we look for must be implementable, in

other words it should be a deterministic automaton. It has appeared that some BPEL specifications do not admit such a client i.e. they are inherently ambiguous. Thus the algorithm we have developed either detects the ambiguity of the Web service or generates a deterministic automaton satisfying the interaction relation. The core of this algorithm is a kind of determinisation of the transition system of the service.

Independently we have proposed an Aspect Oriented Programming (AOP) [18] approach which aims to change elementary Web services at runtime [3, 24].

Our contributions. Here, we extend these works by providing:

- A method to design, deploy and publish aspect-oriented Web services;
- A formal semantics for aspect-oriented Web services;
- An algorithm that generates a client (i.e. an automata) based on this semantics or detect that the service is ambiguous;
- A client interpreter able to handle interactions not fully specified in the published Web service description.

This paper is organized as follows. Section 2 details the approach for synthesis of client for service without adaptation. Section 3 presents the generalisation to aspect oriented web services. Section 4 discusses related work. Finally in section 5 we conclude and give some perspectives to this work.

2 Client synthesis for Web services

In this section, we develop the principle of client synthesis for services *without* adaptation.

2.1 A formal semantics for BPEL abstract processes

BPEL provides a set of operators describing in a modular way the observable behaviour of an abstract process. As shown in [22], this kind of process description is close to the process algebra paradigm illustrated for instance by CCS [20], CSP [14] and ACP [4]. However, time is explicitly present in some of the BPEL constructors and thus the standard process algebra semantics are inappropriate for the semantics of such a process. In order to model time, we have chosen a discrete time semantics since on the one hand the theory of dense time is more involved and on the other hand timing requirements in Web services are simpler than in real-time systems. Our semantics associates a finite automaton with an abstract process.

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

- A time unit elapsing is denoted by χ.
- Silent actions, denoted by τ cannot be observed by the client. They correspond to decisions taken by the server (evaluation of a condition for switch, while, etc.).
- Exceptions; the set of exception events is denoted by Ex.

– In order to control that the client correctly detects the end of the service, we introduce $\surd$, the termination event. This action will also simplify the definition of the operational semantics.
– Sending and receiving messages: the set of types of messages will be denoted by M. The emission is denoted by $!m$ and the reception is denoted by $?m$. We also set $!M = \{!m \,|\, m \in M\}$ and $?M = \{?m \,|\, m \in M\}$ and the wildcard $*$ may be substituted for ! or ?.

Actions different from time elapsing can be classified as immediate (τ, $\surd$ and exceptions) or delayed (emissions and receptions). The first kind of actions are performed in null time (w.r.t. the time scale) and thus in our semantics have priority over the other actions including time elapsing.

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

The transitions of the automaton The transitions starting from a state are obtained by a top down analysis of the process expression labelling this state. This analysis is usually defined with the help of operational semantic rules. The definition of a semantic rule $[op_x]$ for a generic process $P = op_x(P_1, P_2, \ldots)$ includes the following parts:

– a boolean expression over some potential transitions of selected components of P: $Bexp(\{P_{o(i)} \xrightarrow{\alpha_i} P'_{o(i)}\})$;
– this condition is enforced by a second condition on the occurring labels denoted by $guard(\{\alpha_i\})$.
– If the two conditions are fulfilled then a state transition for P is possible where the label $Lexp(\{\alpha_i\})$ is an expression depending on the labels of subprocesses transition and
– the new process is an expression $Nexp(P, \{P'_{o(i)}\})$ depending on the original process and the new subprocesses.

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

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

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

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

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

The $?o[m]$ *and* $!o[m]$ *processes* The process $?o[m]$ (which corresponds to the input operation of WSDL) consists in receiving a message of type m. The process $!o[m]$ (which corresponds to the notification operation of WSDL) consists in sending a message of type m. We consider only these two types of WSDL operations. The two other types can be built with the sequence constructor (see below). Since these actions are not immediate, time can elapse. This leads to the two rules below.

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

The `throw` *process* The process $\texttt{throw}[e]$ raises an exception e which must be catched in some `scope` process.

$$\texttt{throw}[e] \xrightarrow{e} 0$$

We also introduce an auxiliary process `time` that represents time elapsing (not present in the BPEL definition).

$$\texttt{time} \xrightarrow{\chi} \texttt{time}$$

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

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

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

Remark. The set of rules will imply that if there is an action $a \neq \surd$ such that $P \xrightarrow{a} P'$, then $P \xrightarrow{\surd}$ cannot occur.

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

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

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

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

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

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

• Individual actions:

1.

$$\forall j \in I \ \frac{P_j \xrightarrow{a} P'}{\texttt{flow}[\{P_i\}_{i\in I}] \xrightarrow{a} \texttt{flow}[\{P_i\}_{i\in I\setminus\{j\}} \cup \{P'\}]} \text{ where } a \in Ex \cup \{\tau\}$$

2.

$$\forall\, m \in M\ \forall j \in I \ \frac{P_j \xrightarrow{*m} P' \wedge \forall\, i \in I\ \forall\, a \in Ex \cup \{\tau\},\ \neg P_i \xrightarrow{a}}{\texttt{flow}[\{P_i\}_{i\in I}] \xrightarrow{*m} \texttt{flow}[\{P_i\}_{i\in I\setminus\{j\}} \cup \{P'\}]}$$

• Time elapsing: all processes must either let time elapse or terminate.

$$\forall J \neq \emptyset\ J \subseteq I \frac{\forall i \in J\ P_i \xrightarrow{\chi} P_i' \wedge \forall i \in I \setminus J\ P_i \xrightarrow{\surd}}{\texttt{flow}[\{P_i\}_{i\in I}] \xrightarrow{\chi} \texttt{flow}[\{P_i'\}_{i\in J} \cup \{P_i\}_{i\in I\setminus J}]}$$

• Termination:

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

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

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

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

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

$$\frac{P \xrightarrow{\surd}}{\texttt{scope}(P, E^d) \xrightarrow{\surd} 0} \qquad \frac{P \xrightarrow{a} P'}{\texttt{scope}(P, E^d) \xrightarrow{a} \texttt{scope}(P', E^d)} \text{ where } a \notin Ex \cup M_I \cup \{\surd, \chi\}$$

• Receiving a message m_i:

$$\forall\, i \in I \;\; \frac{\forall\, a \in Ex \cup \{\tau, \surd\},\ \neg P \xrightarrow{a}}{\texttt{scope}(P, E^d) \xrightarrow{?m_i} P_i}$$

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

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

$$\forall\, e \in Ex \setminus E_J \;\; \frac{P \xrightarrow{e}}{\texttt{scope}(P, E^d) \xrightarrow{e} 0}$$

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

• Time elapsing

$$\forall\, d > 0 \;\; \frac{P \xrightarrow{\chi} P'}{\texttt{scope}(P, E^d) \xrightarrow{\chi} \texttt{scope}(P', E^{d-1})}$$

• Time out

$$\frac{P \xrightarrow{\chi}}{\texttt{scope}(P, E^1) \xrightarrow{\chi} Q}$$

The `pick` *process* can be viewed as a particular case of the `scope` process: $\texttt{pick}(E^d) \equiv \texttt{scope}(\texttt{time}, E^d)$.

2.2 Interaction relation

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

Furthermore, the states of a pair should have a coherent view of the next interaction steps to occur. At first, this implies that the relation must take into account the mutually observable steps. Thus we introduce the observable transition relation of an automaton by $s \stackrel{a}{\Rightarrow} s'$ iff $s \stackrel{\tau^* a \tau^*}{\rightarrow} s'$, $s \stackrel{\epsilon}{\Rightarrow} s'$ iff $s \stackrel{\tau^*}{\rightarrow} s'$.

Once it is done, we could require (like for bisimulation) that if a state s of the pair (s, s') may evolve by an observable transition of its automaton to some new state s_1, s'

should have a similar observable transition leading to a state s'_1 which would compose with s_1, a new pair of consistent views.

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

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

These considerations yield the following formal definition.

Definition 1 (Interaction relation).
Let $A_1 = (S, s_{01}, A, \rightarrow_1)$ *and* $A_2 = (S, s_{02}, A, \rightarrow_2)$ *be two automata. Then* A_1 *and* A_2 correctly interact *iff* $\exists \sim \subseteq S_1 \times S_2$ *such that:*

- $s_{01} \sim s_{02}$
- $\forall s_1, s_2$ *such that* $s_1 \sim s_2$
  - *Let* $a \notin \{?m \mid m \in M\}$ *then*
    - *if* $\exists\, s_1 \stackrel{a}{\Longrightarrow}_1 s'_1$, *then* $\exists\, s_2 \stackrel{\overline{a}}{\Longrightarrow}_2 s'_2$ *with* $s'_1 \sim s'_2$
    - *if* $\exists\, s_2 \stackrel{a}{\Longrightarrow}_2 s'_2$ *then* $\exists\, s_1 \stackrel{\overline{a}}{\Longrightarrow}_1 s'_1$ *with* $s'_1 \sim s'_2$
  - *Let* $m \in M$*; if* $s_1 \stackrel{?m}{\Longrightarrow}_1 s'_1$ *then*
    - $\exists\, s_2^- \stackrel{w}{\Longrightarrow}_2 s_2$, $\exists\, s_2^- \stackrel{w}{\Longrightarrow}_2 s_2^+$, $\exists\, s_2^+ \stackrel{!m}{\Longrightarrow}_2 s'_2$ *with* $s_1 \sim s_2^+$ *and* $s'_1 \sim s'_2$ *where* w *is a word*
    - $\exists\, s_2 \stackrel{!m'}{\Longrightarrow}_2 s'_2$
  - *Let* $m \in M$*; if* $s_2 \stackrel{?m}{\Longrightarrow}_2 s'_2$ *then*
    - $\exists\, s_1^- \stackrel{w}{\Longrightarrow}_1 s_1$, $\exists\, s_1^- \stackrel{w}{\Longrightarrow}_1 s_1^+$, $\exists\, s_1^+ \stackrel{!m}{\Longrightarrow}_1 s'_1$ *with* $s_1^+ \sim s_2$ *and* $s'_1 \sim s'_2$ *where* w *is a word*
    - $\exists\, s_1 \stackrel{!m'}{\Longrightarrow}_1 s'_1$

2.3 Client automaton synthesis

We are now in position to present the client synthesis algorithm. Since the client must be implementable, we require it to be *deterministic*. This consideration leads to choose as model for our client a deterministic automaton which is in interaction relation with the automaton of the BPEL process.

Before developing it, we emphasize that there exist BPEL process which do not admit clients. For instance, process `switch`$[?o[m], ?o[m']]$ internally chooses to receive either a message m or m' and thus no deterministic automaton can correctly interact

with it since it would imply that, in its initial state, the client should send either m or m' while the server would wait the other message. Observe the difference with process $\texttt{switch}[!o[m], !o[m']]$ where a client can be easily designed: it just waits for either m or m'. We say that a process is *ambiguous* if it does not admit a deterministic automaton which is in interaction relation with it.

Here we give an abstract view of the algorithm. A detailed description of the algorithm is given in [11]. The general principle of our algorithm is similar to a determinisation procedure: a state of the TA client will correspond to a subset of states of the TA of the service.

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

First, we compute the ϵ-closure by τ-transitions. If this subset (call it S') of service states is already associated with a state s' of the client, then the transition of the client which has generated the subset is redirected to s'. Otherwise, one creates a new client state (say $snew$) and we go on. We check the interaction relation for transitions. If it is not fulfilled then we stop the construction. We give below an algorithmic description of a step of the algorithm.

If S' has already be analysed and paired with s' **Then**
  one redirects the arc entering s toward s' and one deletes s
Else
  For every a s.t. subset S_a of a-successors of S' is non empty **do**
    If $a \notin \{!m\}_{m \in M}$ and $\exists t \in S'\ \neg t \stackrel{a}{\Longrightarrow}$ **Then** ambiguity
    Else If $a \in \{!m\}_{m \in M}$ and $\exists t \in S'\ \forall m'\ \neg t \stackrel{!m'}{\Longrightarrow}$ **Then** ambiguity
    Else create s_a; add $s \stackrel{a}{\longrightarrow} s_a$; stack (s_a, S_a)

2.4 Client interpreter

We have implemented a client interpreter based on the previous theoretical developments. The interpreter downloads the BPEL description of the service. Then it generates the automaton according to the algorithm and it "executes" this automaton (see figure 1). More precisely:

- It maintains the current state of the automaton.
- It opens (or let open) one input window per enabled action $!m$; this means that the user can choose the type of message it wants to send and to enter the corresponding data. It closes the windows that corresponds to messages now disabled.
- It arms a time out of one time unit.
- It changes the current state,
  - either on reception of a message with opening an output window,

- either on validation of an input window with sending of the message,
- or on triggering of time out.

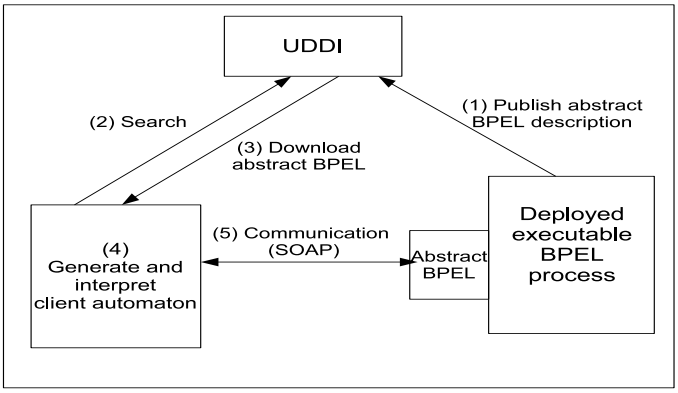

Fig. 1. Generic client interpreter.

3 Aspect Oriented Programming and Web services

3.1 Principles of AOP

AOP is a concept that enables the modularization of crosscutting concerns into single units called aspects, which are modular units of crosscutting implementation [18]. Crosscutting concerns are requirements that cannot be localized to an individual software component and that impact many components. In aspect-speak, these requirements cut across several components. Aspect-oriented languages [19, 15, 2, 21] are based on three paradigms:

1. *Joinpoints*: They denote the locations in the program that are affected by a particular crosscutting concern.
2. *Pointcuts*: They specify a collection of conditional joinpoints.
3. *Advices*: They are codes that are executed *before*, *after* or *around* a joinpoint.

For instance the logging functionality is often scattered horizontally across object hierarchies and has nothing to do with the core functions of the objects it is scattered across. The same is true for other types of code, such as security, exception handling, and transparent persistency. This scattered and unrelated code is known as crosscutting code and is the reason for AOP's existence. Using AOP, we can insert the logging code into the classes that need it with a tool called a *weaver*. This way, objects can focus on their core responsibilities. The figure 2 shows the weaving process. The weaver is in charge for taking the code specified in a traditional (base) programming language, and the additional code specified in an aspect language, and merging the two together. The

weaver has the task to process aspects and component code in order to generate the specified behaviour. The weaver inserts the aspects in the specified joinpoint transversally. The weaving can occur at compile time (modifying the compiler), load time (modifying the class loader) or runtime (modifying the interpreter).

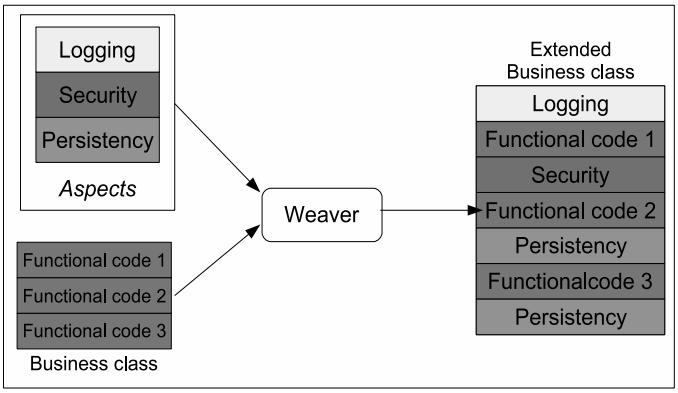

Fig. 2. The weaving process

3.2 Adapting BPEL processes

Several researches [5, 6, 26] consider AOP as an answer to improve WS flexibility. In our previous approach, we developed an AOP-based tool named *Aspect Service Weaver* (*ASW*) [3, 24]. The ASW intercepts the SOAP messages between a client and an elementary WS, then it verifies during the interaction whether there is a new behaviour introduced (*advice service*). We use the AOP weaving time to add the new behaviour (*before*, *around* or *after* an activity execution). The advices services are elementary WSs whose references are registered in a file called "*aspect services file descriptor*". The pointcut language is based on XPath [28]. XPath queries are applied on the service description (WSDL) to select the set of methods on which the advice services are inserted.

We extend this approach to BPEL processes. We apply the AOP concepts to a BPEL processe in order to automatically generate an extended BPEL process without modifying the BPEL engine. This process contains the base BPEL process and the advices services. We apply the AOP concepts on BPEL processes in the following way:

1. A joinpoint is a simple or structured BPEL activity.
2. The pointcuts are specified on the BPEL document by using XPath.
3. The advices services are BPEL processes implementing the additional behaviour.

We also add to the generated process, a replying activity before each inserted advice service (see figure 3). This activity sends to the client a message called $execute$. This

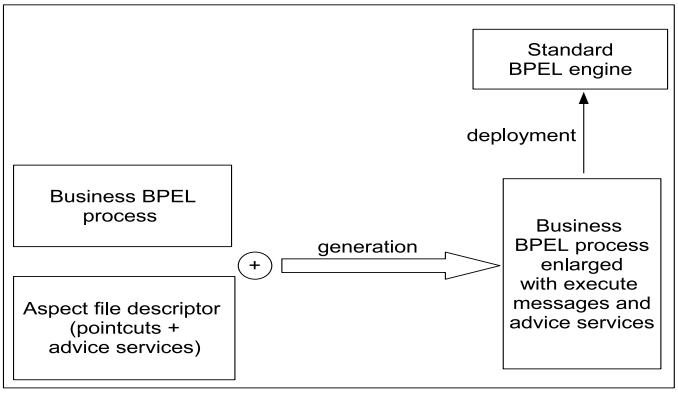

Fig. 3. The extended executable BPEL process.

message informs the client about the execution of an additional behaviour. It encapsulates two kinds of information: the identifier of the advice service and its corresponding interaction protocol. This message is necessary since this additional behaviour can require new information exchange involving messages not expected by the client and leading to execution failures. At the level of client implementation, the developer has to handle this type of message: it must extract the interaction protocol of the *advice service* and integrate it in its behaviour. This part is detailed later.

3.3 Extended BPEL generator

These previous concepts are concretisized through the architecture of our tool named *extended BPEL generator*. The tool contains the following components (see figure4):

1. The *BPEL weaver*
2. The *aspect services file descriptor*
3. The *service advice repository* (or the *pattern repository*) which contains the services advices present in the system.
4. The *deployment module* which deploys the extended BPEL process an a standard BPEL engine.

The BPEL weaver takes as input the base BPEL process and the *aspect services file descriptor*. Then, it performs transformations on the base BPEL process syntactic tree. It inserts the actions of sending $execute$ messages and the advices services at the selected joinpoints depending on the kind of the *advice service*. The figure 5 shows the transformations made on the base process

$$receive(ResReq); switch(\{reply(ResResp) reply(error)\})$$

which receives a $ResReq$ message then replies by a $ResResp$ or $error$ message depending on a condition (the $switch$ process). In the case of an around *service advice* (figure 5.d), the specified $joinpoint$ is replaced by the advice service and the $execute$

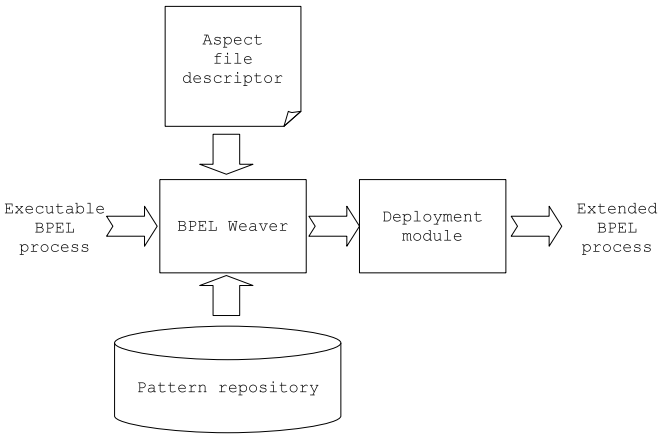

Fig. 4. The extended BPEL generator.

message replying activity, because we consider that the *advice service* can encapsulate the joinpoint. In the figure, a triangle represents an advice service and Q its corresponding interaction protocol.

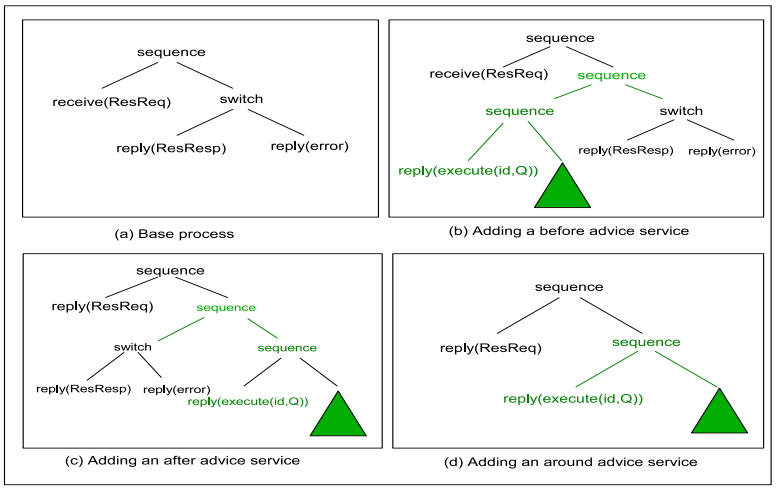

Fig. 5. Syntactic transformations on the base executable BPEL process.

3.4 The extended interaction protocol

The extended executable BPEL process interaction protocol is described by an extended abstract BPEL process which integrates the sending of $execute$ messages. The extended interaction protocol is generated from the base BPEL process and the *aspect service file descriptor* based on the defined pointcuts and the type of advices (before, after or around).

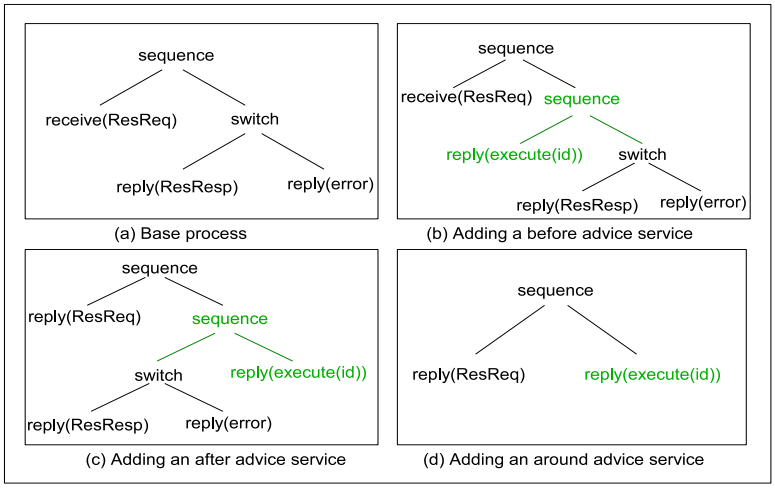

Fig. 6. Transformations on the syntactic BPEL process tree.

The generation process performs transformations on the base abstract BPEL process syntactic tree. It inserts the action of sending $execute$ messages in the selected joinpoints depending on the kind of the *advice service* (figure 6). The *execute* messages contain only the identifier of the *advice service* id. The interaction protocol corresponding to that id is sent to the client at runtime. In this way, the advice service can be changed at any time without requiring a new publication. Since we have chosen to let unchanged the BPEL engine, the weaver acts at deployment time.

3.5 The new operational rules

In order to take into account the special nature of the message $execute$ we modify the operational rules related to messages.

$$\forall m \in M \;\; *o[m] \xrightarrow{\chi} *o[m] \quad \text{with } * \in \{?, !\}$$

$$\forall m \in M \setminus \{execute\} \;\; *o[m] \xrightarrow{*m} \texttt{empty} \quad \text{with } * \in \{?, !\}$$

$$!o[m] \xrightarrow{!execute(id)} WaitAdvice(id) \tag{1}$$

$$WaitAdvice(id) \xrightarrow{id.\surd} empty \quad (2)$$

The two first rules are similar to those presented in subsection 2.1. In the case of sending an $execute$ message, the automaton evolves to an intermediary state named $WaitAdvice(id)$ (rule 1). $WaitAdvice(id)$ waits for the termination of the *advice service* identified by id. When *advice service* id terminates, $WaitAdvice(id)$ state executes $id.\surd$ and becomes $empty$ process (rule 2). In words these two rules mimic the synchronisation corresponding to a procedure call.

3.6 The dynamic client interpreter

In order to communicate with change-prone BPEL processes, we extend the previous client interpreter. The new client has to achieve the following tasks:

1. When the client receives an $execute(id)$ message, it has to extract the *advice service* interaction protocol (identified by id) and generates its corresponding server and client automaton.
2. It simultaneously executes the client automaton of the main process and its *advices clients* automata.
3. It makes synchronisation between the main client TA and the advices clients TA on the termination of services advices execution.

Furthermore, the generation module of the dynamic client interpreter also integrates new operational rules for sending and receiving in order to handle the $execute(id)$ messages.

3.7 Execution Scenario

Let us consider the abstract BPEL process defined before. If we want to add dynamically an authentication process before the $switch$ process, the extended abstract BPEL process have to integrate a sending $execute(id)$ message process before the $switch$ process.

$$?o[ResReq]; !execute(id); switch(!o[ResResp], !o[error])$$

At execution time, the dynamic client interpreter downloads the extended abstract BPEL specification. Then, it generates the corresponding service automaton based on the operational rules previously defined. Then, based on the service automaton and the interaction relation, our client generates the client automaton and begins its interpretation. Figure 3.7 shows the generation process.

When the client receives an $execute(id, Q)$ message, it extracts the abstract BPEL *advice service* process from the message. In our example, the *advice service* is an authentication process which abstract BPEL specification is:

$$!o[authDataRequest]\ ;\ ?o[authDataResp]$$

This process sends an authentication data request to the client asking for authentication data, receives these data then performs some actions to authenticate the user not

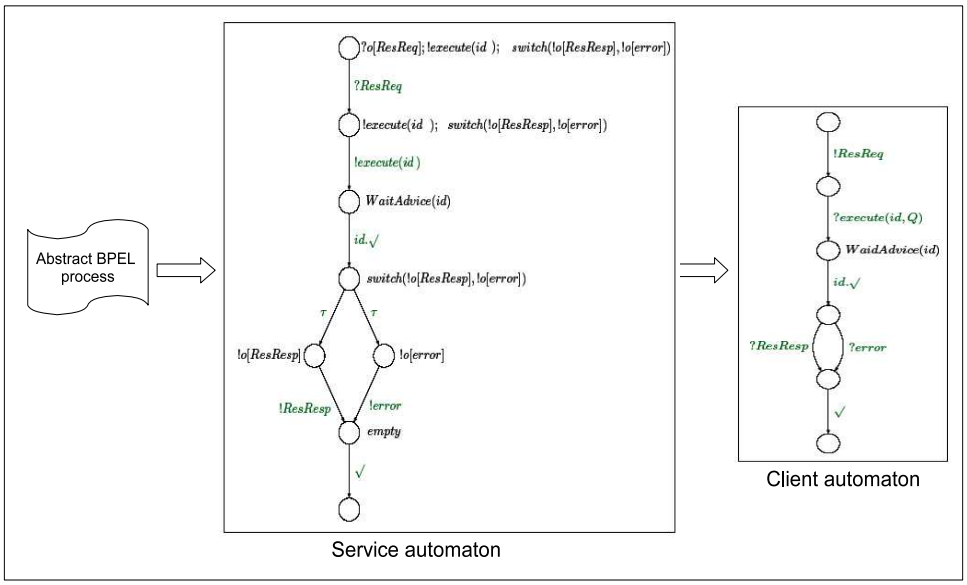

Fig. 7. Adaptable service and client automata

represented here for simplicity. The client generates the corresponding advice client automaton, associates with the received id and begins its execution (Figure 8.(left), states in grey represents the current execution step).

When the *advice client id* terminates, the client synchronises the two automata: it deletes the *advice client*, performs the $id.\surd$ action and continues the execution of the main client automaton (figure 8.(right)).

4 Related work

Formal specification and verification of Web services. Some proposals have emerged recently to abstractly describe WSs, most of them are grounded on transition system models (Labelled Transition Systems, Petri nets, etc.) [13, 7]. The platform WSAT [9, 10] enables designers of a Web service composition to model check properties expressed by LTL formulas with SPIN tool. The formal semantics is obtained by gluing patterns for each BPEL construction. One pattern is connected from its final state to the initial state of next pattern according to the BPEL description with local transitions. This work does not cover the time features and it focuses only on message exchanges: the conversation is obtained by a *virtual watcher* that is supposed to record all messages sequences sent by each peer enrolled in the composition.

Another research of Web services formal semantics is based on a BPEL to Finite State Processes (FSP) translation [8]. This work lies on message sequence charts and the core of the verification mechanism consists to check trace equivalence. Again, the time features of the specification are not taken into account.

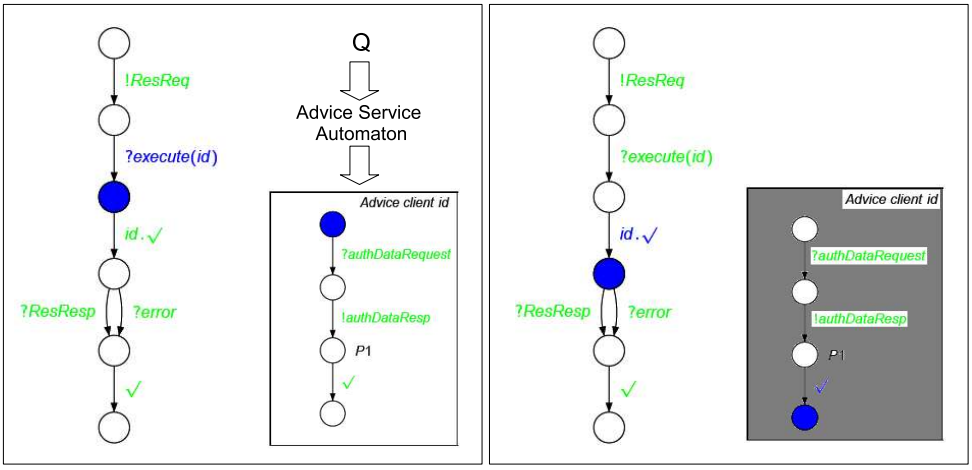

Fig. 8. Reception of an $execute(id, Q)$ message (left) and the termination of an advice service (right)

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

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

AOP and Web services. In [5] and [6], the authors define specific AOP languages to add dynamically new behaviours to BPEL processes. But, neither of these approaches address the client interaction issue. The client has no mean to handle the interactions that can be added or modified during the process execution.

The Web Service Management Layer (WSML) [26] is an AOP-based platform for WSs that allows a more loosely coupling between the client and the server sides. WSML handles the dynamic integration of new WSs in client applications to solve client execution problems. WSML dynamically discover WSs based on matching criteria such as: method signature, interaction protocol or quality of service (QOS) matching. In a complementary way, our work proposes to adapt a client to a modified WS.

5 Conclusion

In this paper, we proposed a solution based on AOP and process algebra to handle dynamic changes in the context of Web services. We extended our previous AOP approach

to support BPEL processes and to handle interaction issues. We also use process algebra formalism to specify change-prone BPEL processes and generate dynamic clients.

As future works, we want to take into account the client execution context. We also want to formally handle the aspect interactions issue (aspects applied at the same join-point). Finally, we plane to improve the current ASW prototype as proof-of-concepts.

References

1. T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic and S. Weerawarana, Business Process Execution Language for Web Services, ftp://www6.software.ibm.com/software/developer/library/ws-bpel11.pdf (2003)
2. AspectWerkz, Web site available at http://Aspectwerkz.codehaus.org.
3. M. Ben Hmida, R. Tomaz Feraz and V. Monfort, Applying AOP concepts to increase Web Service Flexibility, in JDIM journal, ISSN 0972-7272, Vol.4 Iss.1 (2006).
4. J.A. Bergstra and J.W. Klop, Process algebra for synchronous communication, Information and Control, vol. 60, n° 1-3, 109-137 (1984)
5. A. Charfi and M. Mezini. Aspect-oriented web service composition with ao4bpel. In ECOWS, volume 3250 of LNCS, pages 168-182, Springer, (2004).
6. C. Courbis and A. Finkelstein. Weaving aspects into web service orchestrations. In ICWS, pages 219-226, (2005).
7. A. Ferrara, Web Services: A Process Algebra Approach, Proceedings of the 2nd International Conference on Service Oriented Computing, ACM Press, USA (2004).
8. H. Foster, S. Uchitel, J.Magee and J.Kramer, Model-based verification of web service compositions, Proc. of. the 18th Int. Conf. on Automated Software Eng., (2003)
9. X. Fu, T. Bultan and J. Su, Analysis of Interacting BPEL Web Services, Proc. of the 13th International World Wide Web Conference (WWW'04), USA, ACM Press, (2004)
10. X. Fu, T. Bultan and J. Su, WSAT: A Tool for Formal Analysis of Web Services, Proc. of the 16th International Conference on Computer Aided Verification (CAV'04), (2004)
11. 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 (2004)
12. S. Haddad, P. Moreaux and S. Rampacek, Client synthesis for Web Services by way of a timed semantics, In Proc. of ICEIS'06, Paphos-Cyprus (2006).
13. R. Hamadi and B. Benatallah, A Petri Net-based Model for Web Service Composition, Proceedings of Australasian Database Conference, Australia (2003).
14. C.A.R. Hoare, Communicating Sequential Processes, Prentice-Hall, Englewood Cliffs, NJ, USA (1985)
15. JBoss AOP, Web site availble at http://www.jboss.org.
16. M. Juric, BPEL and Java, http://www.theserverside.com/articles/article.tss?l=BPELJava,Online journal theserverside.com (2005)
17. M. Juric, P. Sarang and B. Mathew, Business Process Execution Language for Web Services, Packt Publishing (2005)
18. G. Kiczales et al. , Aspect-Oriented Programming, in proc. of ECOOP'97. LNCS 1241, Spinger-Verlag, (1997).
19. R. Laddad, ASPECTJ in Action: Practical Aspect-Oriented Programming, Portland : Book News, Inc, 2004.
20. R. Milner, Communication and Concurrency, Prentice-Hall, Englewood Cliffs, NJ, USA (1989)

21. Spring AOP platform, Web site available at http://www.springframework.org/docs/ reference/aop.html.
22. S. Staab, W. van der Aalst, V.R. Benjamins, A. Sheth, J.A. Miller, C. Bussler, A. Maedche, D. Fensel and D. Gannon, Web Services: Been There, Done That?, IEEE Intelligent Systems, vol. 18, 72-85, (2003)
23. D. Tidwell, Web services - the web's next revolution. IBM developerWorks (2000).
24. R. F. Tomaz, M. Ben Hmida and V. Monfort, Concrete Solutions for Web Services Adaptability Using Policies and Aspects , The International Journal of Cooperative Information Systems (IJCIS), to be published, (September 2006).
25. K. J. Turner, Formalising Web Services, Proc. of Formal Techniques for Networked and Distributed Systems (FORTE 2005), LNCS 3731 473-488, Taipei, Taiwan (2005)
26. B. Verheecke, M.A. Cibran and V. Jonckers, AOP for Dynamic Configuration and Management of Web Services, ICWS-Europe, LNCS 2853, pages 137-151, (2003).
27. Web Services Description Language (WSDL) 1.1, World Wide Web Consortium, http://www.w3.org/TR/wsdl (2001)
28. XML Path Language (XPath) Ver. 1.0, W3C Recommendation 16 November (1999). Available at: http://www.w3.org/TR/xpath