Dynamically Adapting Clients to Web Services Changing

Mehdi Ben Hmida[1], Céline Boutrous-Saab[1], Serge Haddad[1], Valérie Monfort[1,2], and Ricardo Tomaz Ferraz[2]

[1] LAMSADE, Université Paris 9 Dauphine,
Place du Maréchal de Lattre Tassigny
Paris Cedex 16, France
[2] CRI, Université Paris 1 Sorbonne,
90, rue de Tolbiac
75013 Paris, France

Abstract. Web Service is the fitted technical solution which provides the required loose coupling to achieve Service Oriented Architecture (SOA). However, there is still much to be done in order to increase flexibility and adaptability to SOA-based applications. In previous researches, we proposed approaches based on Aspect Oriented Programming (AOP) and Process Algebra (PA) to address flexibility and client generation issues in the Web Service context. In this paper, we extend our previous formalism defined for abstract BPEL processes, with three AOP constructs. The new formalism allows to specify dynamic change-prone BPEL processes. We also define the extended interaction relation which characterizes the concept of correct interaction between the adaptable BPEL process and its client. Then, we propose an algorithm to generate a client which dynamically adapt itself to the service changing.

Key words: Service Oriented Architecture (SOA), Web Services (WS), BPEL4WS, Aspect Oriented Programming (AOP), Process Algebra (PA).

1 Introduction

Web Services (WS) are "self contained, self-describing modular applications that can be published, located, and invoked across the Web" [1]. They are based on a set of XML [2] standards to make it more portable than previous middleware technologies [3]. WSs need to be composed to fulfill business requirements. The *Business Process Execution Language for Web Services* (*BPEL4WS* or *BPEL*) has been proposed for this purpose and becomes a standard [4]. BPEL supports two different types of business processes:

1. *Executable processes* specify the exact details of business processes and are executed by a BPEL engine.
2. *Abstract business processes* specify the public message exchange between the client and the service (the interaction protocol).

Web Service technology is asked to handle the same features as middlewares such as DCOM [5], J2EE [6] or CORBA [7] already handle. The features, such as security, reliability, or transactional mechanisms, can be considered as non-functional aspects. Obviously, these aspects are crucial for business purposes and one cannot build any genuine IS without consideration for them.

However, managing these aspects is likely to involve a great loss in interoperability and flexibility. This effect has already been experienced with the above middleware technologies. Mostly, middleware delegates these tasks to the underlying platform, hiding these advanced mechanisms from the developer, and then establishing a solid bond between the application and the platform.

Moreover, WS providers are faced to some important difficulties to change their services behaviours because WSs are shared by many clients and a minor change leads to client execution problems.

Regarding the above limitations, we identified two requirements that Web Service technology has to handle:

1. Service providers need to dynamically change the behaviour of an already existing Web Service to adapt it to the new applications requirements.
2. The Web Service client needs to dynamically adapt itself to the service changing to avoid execution failures.

Previously, we proposed two approaches to address flexibility and client generation. The first approach is based on AOP [8] and aims to change the WS behaviour without touching its implementation [9, 10]. The second is based on Process Algebra (PA) and defines a formalism for the abstract BPEL language in order to automatically generate a client which correctly interacts with the service [11].

In this paper, we extend our BPEL specification formalism (defined in the second approach) to generate a client which is able to dynamically adapt itself to a service changing. More precisely, we add three AOP constructs and their operational semantics. These constructs describe change-prone BPEL processes.

This paper is organized as follows: section 2 presents our previous works. Section 3 defines the extended BPEL formalism. Section 4 applies our approach to a concrete case study taken from industry. Section 5 draws limitations and some possible enhancements for the approach. Section 6 draws comparisons with other works. We conclude and enumerate future works in section 7.

2 Our previous approaches

2.1 Aspect Oriented Programming (AOP)

Many researches [12–14] consider Aspect Oriented Programming AOP as an answer to improve WS flexibility. AOP is a paradigm that enables the modularization of crosscutting concerns into single units called aspects, which are modular units of crosscutting implementation. AOP concepts were formulated by Chris Maeda and Gregor Kiczales.[8]

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 such as AspectJ [15], JBoss AOP [16], AspectWerkz [17], Spring AOP [18], etc. are implemented over a set of definitions:

1. Joinpoints: They denote the locations in the program that are affected by a particular crosscutting concern.
2. Pointcuts: They specify a collection of joinpoints.
3. Advices: They are codes that run upon meeting several conditions. Advices can be executed before, after or around a joinpoint.

To better clarify, consider the classical example to implement a logging functionality. Logging code 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 Object-Oriented Programming, every time we need to introduce the logging functionality in an application, the programmer must add the logging code into the appropriate objects. 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 1 shows the weaving process.

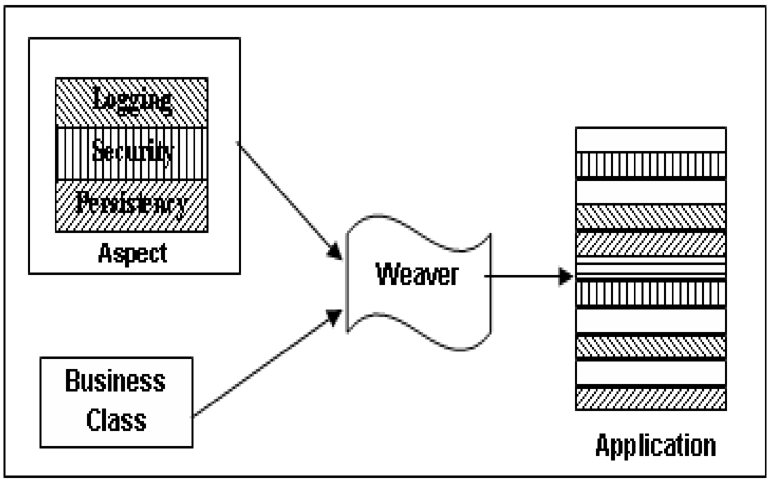


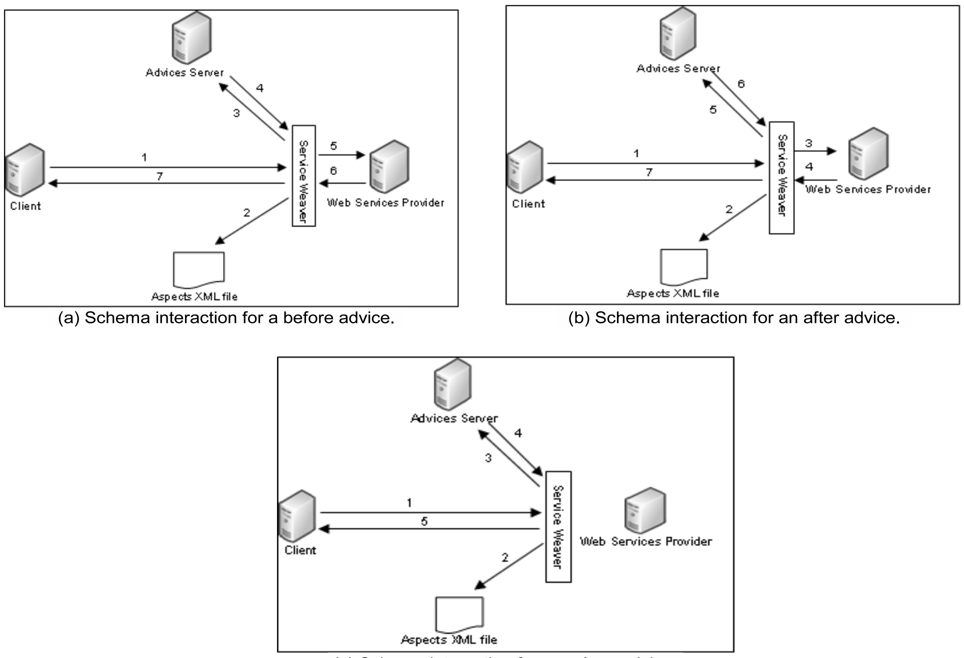
Fig. 1. The weaving process

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.

2.2 Aspect Service Weaver (ASW)

In our first approach, we developed an AOP-based tool named *Aspect Service Weaver* (*ASW*) [9, 10]. The ASW intercepts the SOAP messages between a client and a WS and redirects these messages for other WSs (*advice services*) that implement the new behaviours. We use the AOP weaving time to intercept and redirect the messages (before or after an original WS invocation). The advice services are registered in a file called "*aspect services file descriptor*". The pointcut language is based on XPath [24]. XPath queries are applied on *BPEL* to select the set of methods on which aspect services are inserted.

Based on a well defined set of rules the engine generates complex interactions among original Web Services methods and advice services.



(a) Schema interaction for a before advice.

(b) Schema interaction for an after advice.

(c) Schema interaction for a replace advice.

Fig. 2. Interactions schemas

For example, whenever a new behaviour is inserted before a service method invocation (figure 2.a), the engine redirects the request to the proper advice service (3) before the request reaches the original method. Then, it captures the http/SOAP answer (4) and sends it to the original method (5). The original method performs its task and sends back the answer to the engine (6). Finally, the engine dispatches the final answer to the client caller (7). The interactions performed whenever an "after" or a "replace" advice is detected are described in the figure 2.b and figure 2.c, respectively.

But, what happens if the advice service requires new interactions with the client? These interactions are not expected and leads to client execution failures.

We solve this problem by extending the abstract BPEL formalism defined in our second approach [11]. In that work, we proposed a dense time operational semantics for abstract BPEL constructs based on Timed Automata (TA)[19]. The theorical developments follow these steps: associating operational rules with each abstract BPEL construct, defining an interaction relation which formalizes the concept of a correct interaction between two communicating systems (the client and the WS), and the design of an algorithm that generates a client automaton which is in an interaction relation with the WS. The client automaton is interpreted by the client program.

3 Adaptable BPEL Processes Formalisation

BPEL provides a set of operators describing in a modular way the observable behaviour of an abstract process. As shown in [20], this kind of process description is close to the process algebra paradigm illustrated for instance by CCS [21], CSP [22] and ACP [23]. 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. Thus our semantics associates a timed automaton [19] (TA) with an abstract process.

Let us briefly present TA and our formal semantic for abstract BPEL processes.

3.1 formal semantics for abstract BPEL processes

TA is a tuple $T = (L, C, B, A, E, S0, F)$, where L is the set of locations (or control states), C is the set of clocks, B is the set of boolean expressions built other atomic boolean propositions, 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. $S0$ 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 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 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 $(S0, 0)$, 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.

1. 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.
2. A transition $(s, v) \xrightarrow{d(t)} (s, v+t)$ corresponds to some time passing in location s and may occur iff $(\forall t' < t\ I(s)(v + t') = tt$.

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

In order to formalize BPEL as dense timed process algebra, we have to define the actions (alphabet) of the process algebra. The possible actions are message receiving (?m) and sending (!m), internal actions (τ), raise of exceptions ($e \in E$), expiration of timeout (t0) and the termination of the process ($\surd$). We distinguish three kind 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 actions ($?m, !m$) and the synchronous actions (t0) which occur after a fixed delay.

Now, we present some operational rules useful to better understand the rest of the paper. To see all rules, the reader is invited to read [11].

For example, the process ?o[m] (resp. !o[m]) which corresponds to the reception of a message of type m (resp. sending of message of type m) executes the action ?m (resp. the action !m) which corresponds to the message reception action (resp. the message sending action) and becomes the empty process (the process which does nothing).

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

The sequential process P;Q (P and Q are BPEL processes) which corresponds to the execution of the process P followed by the execution of the process Q, becomes the process P';Q if the process P executes an action a different from termination action and becomes P'. If P terminates, the process P;Q becomes the the process Q.

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

$$\frac{P \xrightarrow{\surd}}{P; Q \xrightarrow{\surd} Q}$$

Let us now present the new operators added to the formalism in order to formally handle adaptable BPEL process.

3.2 Extended Formalism for Adaptable BPEL processes

We extend the previous defined formalism with three AOP constructs. The new constructs enable to specify processes which behaviours are prone to change. So, given an abstract BPEL process P defining the original behaviour, we have:

1. $Before(P)$ specifies that P may have a new behaviour executed before it.
2. $After(P)$ specifies that P may have a new behaviour executed after it.
3. $Around(P)$ specifies that may have a new behaviour that replaces P.

We insert two parameterized actions in the alphabets of the TA. These actions are labelled : $!esc[Q]$ and $?esc[Q]$ (where Q is an abstract BPEL process). $!esc[Q]$ specifies that the ASW sends a message, containing the new behaviour Q to the client, by performing the action $!esc$. $?esc[Q]$ specifies that the client is waiting to receive a message, containing the new behaviour Q.

Now, we present the operators semantic rules. We stress that our operators do not take into account the time because we consider that the introduction of the new behaviour is an immediate action.

Process $Before(P)$ sends, by performing the action $!esc$, a message containing the new behaviour Q and becomes the sequential process $Q; P$. If no behaviour is inserted, Q is the *empty* process (the process which does nothing).

$$Before(P) \xrightarrow{!esc(Q)} Q; P$$

Process $After(P)$ becomes process $After(P')$ if P performs an action a, different from the termination action $\surd$, and becomes P'. If P terminates, process $After(P)$ sends, by performing action $!esc$, a message containing the new behaviour Q and then becomes Q. Q is *empty* process if there is not new behaviour.

$$\forall a \neq \surd \frac{P \xrightarrow{\text{a}} P'}{After(P) \xrightarrow{\text{a}} After(P')}$$

$$\frac{P \xrightarrow{\surd}}{After(P) \xrightarrow{!esc(Q)} Q}$$

Finally, process $Around(P)$ sends, by performing action $!esc$, a message containing the new behaviour Q and then becomes Q. If no behaviour is inserted, $Around(P)$ performs action $!esc(empty)$ and becomes P.

$$\forall Q \neq empty \quad Around(P) \xrightarrow{!esc(Q)} Q$$

$$Around(P) \xrightarrow{!esc(empty)} P$$

3.3 The 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. In addition to these messages, the client also expect the *esc* operation message sent by the ASW and containing the automaton of the new behaviour.

Classical comparison relations between timed systems are language equality and bisimulation. But, equality relation does not take into account intermediate actions and the bisimulation relation does not distinguish different kinds of actions. We have then defined a relation analogous to the bisimulation relation which take into account the kind of actions. The interaction relation definition is made of three steps:

First, since the relation abstracts the internal actions, we define derivations abstracting invisible actions. If action a is not time passing ($d(t)$), $s \rightarrow a \rightarrow s'$ means that process can evolve from s to s' by first executing (possibly several) invisible τ actions, then a and again τ actions. For time passing actions d(t), $s \rightarrow d(t) \rightarrow s'$ means that the process can evolve from s to s' with internal actions interleaved with time passing, and with a total time passing of t.

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

Finally, TA Q and R interacts if for each state of a TA P able to do action a, its equivalent state of the interaction TA Q is able to do $c(a)$.

3.4 Algorithms for the adaptable BPEL process and client automata generation

The ASW takes the abstract BPEL process and the "Aspect Services file descriptor" as input and generates the automaton corresponding to the original BPEL process and its new behaviours. The automaton generation follows these steps:

1. ASW generates the automaton of the original BPEL process [11].
2. ASW looks in the "aspect services file descriptor" for joinpoint matchings with the original BPEL process.
3. If there is a joinpoint matching, the ASW replaces in the automaton, the identified joinpoint by the adequate AOP operator.

Regarding client generation, the algorithm is based on a kind of automaton determinization of the service. The algorithms follow these rules:

1. A node of the deterministic client automaton is a set of nodes of the service automaton. This set must not include AOP operators nodes (before, after or around).
2. Each AOP node in the service automaton corresponds to a special node in the client automaton. The client stops its execution at this special node, in order to replace it with the automaton of the new behaviour.
3. Similarly to the approaches which determinize subclasses of timed automata, 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.

Let us now explain the approach principles through a more realistic scenario taken from one of our industrial projects.

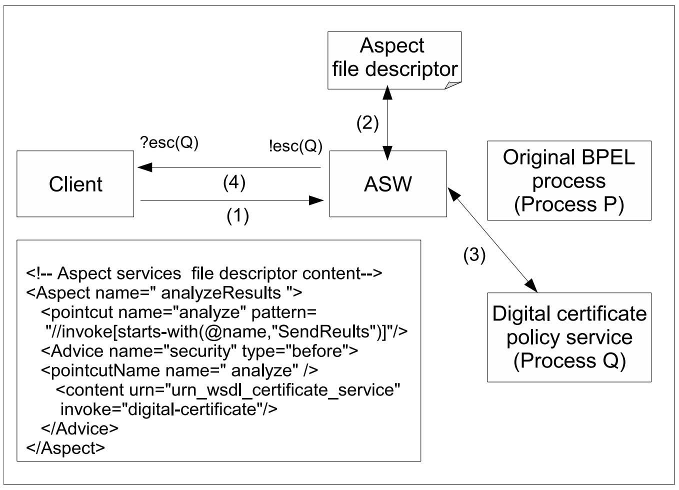
4 Concrete scenario

A company aims to develop automatons to analyse blood plasma, which means patient data information has to be highly reliable and correct. In order to support consequent evolution and successive reuse of the machines, the company decided to define and to promote a flexible and adaptable architecture according to the new emerging requirements. We decided to use Web Services technology to implement this architecture.

The Application must display specific Human Machine Interface (HMI) according to profile and maturity level of the user. Access is allowed or denied according to user profile and protected from unauthenticated usage.

Consider the hypothetical scenario where an original Web Service has an authentication policy like Kerberos token and, that after a while, the company resolves to replace this authentication policy for a digital certificate security policy. Using a classical approach, considering that a change is required in the authentication method signature, clients have to rewrite manually the invocation of methods authenticated with the new security policy.

Using the ASW engine, the programmer can, for instance, develop an aspect Service called ”analyseResults” and specify, in the “aspect services file descriptor”, that before the invocation of the authenticated methods in the Original Web Service the engine must invoke the advice service “digital-certificate”. This way, on the next Original Web Service invocation, the ASW will be aware of the rules specified in the “aspect services file descriptor” and will generate the automaton corresponding to the original BPEL process and its new behaviours.


Aspect file descriptor
(2)
?esc(Q)
!esc(Q)
Client
(4)
(1)
ASW
Original BPEL process (Process P)
(3)
Digital certificate policy service (Process Q)

```
<!-- Aspect services  file descriptor content-->
<Aspect name=" analyzeResults ">
  <pointcut name="analyze" pattern=
   "//invoke[starts-with(@name,"SendReults")]"/>
  <Advice name="security" type="before">
  <pointcutName name=" analyze" />
    <content urn="urn_wsdl_certificate_service"
     invoke="digital-certificate"/>
  </Advice>
</Aspect>
```

Fig. 3. An interaction scenario for the insertion of new security policy

The “aspect services file descriptor” in fig. 3 indicates to the engine that always the methods that match with the XPath expression “//invoke[starts-with(@name,“SendResult”)]” (equivalent to the invocation of methods whose names match the regular expression “SendResult*”) were invoked the engine must invoke before the advice service digital-certificate.

At runtime, the ASW will inform the clients about the new security policy and then send it the corresponding automaton. So, when the ASW receives a SOAP message from the client (step 1 in fig. 3) to invoke an authenticated method, then it looks in the “aspect services file descriptor” and finds a digital-certificate security policy inserted before the method (step 2 in fig. 3), then it generates the automaton of the new security policy (step 3 fig. 3) and sends it to the client by invoking the operation !*esc* (step 4 in fig. 3).

The client handles the operation ?*esc*. It extracts the automaton of the digital-certificate service from the received message and suspends its execution to integrate the new behaviour.

5 Discussion

Our approach let us adapt all Web service clients to the service changing without stopping them and rewriting clients programs code. However, the approach does not allow to adapt the client behaviour giving its context, nor to adapt the service execution giving a client. Consider, from one hand, that depending on the client location, the company wants to offer different levels of authentication. For instance, local clients can still use the digital authentication and have access to the entire service, whereas remote clients can only have access to some restricted services (for example viewing some information) by using the old authentication (Kerberos). From the other hand, the service offers also the possibility to receive normal or compressed XML. A client can, for example, specify to receive compressed XML when he uses a remote connection and normal XML otherwise. In this case, we have to take into account each client’s context and to adapt the offered service to each client.

The ASW does not support such possibilities since there is no way to get the information concerning a client and there is no possibility to allow a specific treatment per client. The behaviour of the service can only be modified globally and not depending on the client execution context.

Thus, the ASW may have in addition to the Aspect Services File descriptor”, a behaviour file per client. The ASW receives from the client in addition to the invocation message itself, the client context in the SOAP header. Giving the information it contains, the ASW executes the service and returns the result, or detects that the client context has changed and requires to adapt the service and the client behaviours consequently. It generates then the corresponding automaton and sends it to the client by invoking the operation !*esc*. The client before resuming its execution handles the execution of the ?*esc*. It extracts the automaton and integrates the new behaviour. Thus the client must be able to receive the operation !*esc* at any moment.

6 Related works

In [12] and [13], the authors propose AOP-based approaches that improve the WS flexibility. These approaches define specific AOP languages to add dynamically new behaviours to BPEL processes. But, neither of them address the client execution errors thrown by the inserted behaviour.

Web Service Management Layer (WSML) [14] 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. Conversely, our approach propose to adapt a client to a modified WS and not to replace it with another one.

Some proposals have emerged recently to abstractly describe WSs, most of which are grounded on transition system models (Labelled Transition Systems, Petri nets, etc.) [26–28]. But, none of them address the client generation issue.

7 Conclusion

In this paper, we have proposed a solution based on AOP and PA to adapt a client to a modified BPEL process. We have extended our abstract BPEL formalism with three AOP constructs specifying change-prone processes. We have also described two algorithms to generate the service and the client automata.

As future works, we propose To extend the ASW to take into account the client execution context. We also propose to formally handle the aspects interaction issue (Aspects applied at the same joinpoint). Finally, we propose to add the new formalism to the current ASW implementation as proof-of-concepts.

References

1. Tidwell, D., Web services - the web's next revolution. IBM developerWorks (2000).
2. Extensible Markup Language(XML) 1.0, W3C Recommendation, February (2004).
3. Web Services Architecture, W3C Working Draft 14 November 2002.
4. Andrews, T. et al., Business process execution language for web services (2003).
5. DCOM Architecture, Microsoft Corporation, technical report, 1998.
6. Java Platform Enterprise Edition(J2EE), web site available at http://java.sun.com /javaee/index.jsp.
7. Object Management Group (OMG), Common Object Request Broker Architecture (CORBA/IIOP), revision 3.0.3, 2004.
8. G. Kiczales et al. , Aspect-Oriented Programming, in proc. of ECOOP'97. LNCS 1241, Spinger-Verlag, (1997).
9. 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).
10. 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).
11. 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).

12. Anis Charfi and Mira Mezini. Aspect-oriented web service composition with ao4bpel. In ECOWS, volume 3250 of LNCS, pages 168-182, Springer, (2004).
13. Carine Courbis and Anthony Finkelstein. Weaving aspects into web service orchestrations. In ICWS, pages 219-226, (2005).
14. 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).
15. R. Laddad, ASPECTJ in Action: Practical Aspect-Oriented Programming, Portland : Book News, Inc, 2004.
16. JBoss AOP, Web site availble at http://www.jboss.org.
17. AspectWerkz, Web site available at http://Aspectwerkz.codehaus.org.
18. Spring AOP platform, Web site available at http://www.springframework.org/docs/ reference/aop.html.
19. R. ALur and D.L. Dill, "A theory of Timed Automata", Theorotical Computer Science, 126, pp. 193-235, 1994.
20. Staab, S., van der Aalst, W., Benjamins, V., Sheth, A., Miller, J., Bussler, C., Maedche, A., Fensel, D., and Gannon, D. (2003). Web services: Been there, done that? IEEE Intelligent Systems, 18:72-85.
21. Milner, R. (1989). Communication and Concurrency. Prentice-Hall, Englewood Cliffs, NJ, USA.
22. Hoare, C. (1985). Communicating Sequential Processes. Prentice Hall, Englewood Cliffs, NJ, USA.
23. Bergstra, J. and Klop, J. (1984). Process algebra for synchronous communication. Information and Control, 60(1- 3):109-137.
24. XML Path Language (XPath) Ver. 1.0, W3C Recommendation 16 November (1999).
25. X. Nicollin and J. Sifakis. The algebra of timed process, atp: Theory and application. Technical report, Information and Computation (1994).
26. R. Hamadi and B. Benatallah, A Petri Net-based Model for Web Service Composition, Proceedings of Australasian Database Conference, Australia (2003).
27. X. Fu, T. Bultan, and J. Su., Analysis of Interacting BPEL Web Services, In Proc. of WWW'04, ACM Press, USA (2004).
28. A. Ferrara, Web Services: A Process Algebra Approach, Proceedings of the 2nd International Conference on Service Oriented Computing, ACM Press, USA (2004).