Dynamically Adapting Clients to Web Services Changing

Mehdi Ben Hmida, Céline Boutrous Saab, Serge Haddad, Valérie Monfort and Ricardo Tomaz Ferraz

Abstract. Web Services are 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 these works in order to automatically create extended BPEL processes and generate clients which dynamically adapt themselves to the service changing.

Keywords. 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 are published, located, and invoked across the Web" [1]. They are based on a set of XML [2] standards to make them 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 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 has 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 we 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.

In our previous works, we addressed service adaptability and client interaction issues. We proposed an Aspect Oriented Programming (AOP) [8] approach which aims to change elementary WSs at runtime [10, 9]. We also proposed a Process Algebra (PA) approach which solves the interaction problem between BPEL processes and their clients. In this paper, we extend these works in order to reach the objectives previously discussed.

This paper is organized as follows: section 2 presents the Aspect Oriented Programming (AOP) paradigm. Section 3 briefly presents our previous AOP approach for elementary WSs, then shows its extension to support BPEL processes. We also present the architecture of our *extended BPEL generator* tool which integrates these concepts. Section 4 presents the process algebra formalism which supports change-prone BPEL processes. This formalism leads us to generate clients that adapt themselves to the service changes. Section 5 discusses related works. We conclude and present future works in section 6.

2. Aspect Oriented Programming (AOP)

Many researches [12, 13, 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 conditional joinpoints.

3. *Advices*: They are codes that are 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.

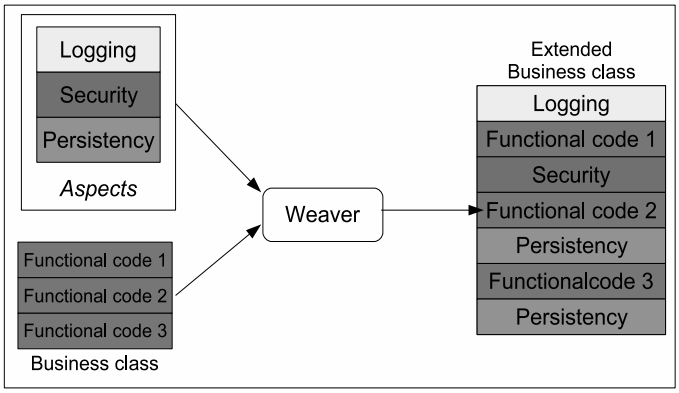

FIGURE 1. The weaving process

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.

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).

3. Adapting BPEL processes

In our previous approach, we developed an AOP-based tool named *Aspect Service Weaver* (*ASW*) [10, 9]. The ASW intercepts the SOAP messages between a client and an elementary WS , then verifies during the interaction if 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 [24]. 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 BPEL processes in order to automatically generate extended BPEL processes without touching the base implementation. The new document is deployed on a standard BPEL engine. It 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 poincuts are specified on the BPEL document by using XPath.
3. The advices services are BPEL processes implementing the new behaviour.

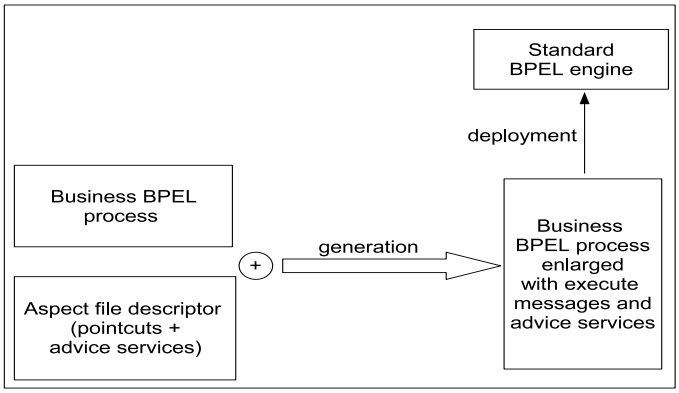

FIGURE 2. The extended executable BPEL process.

We also add to the generated process, a replying activity before each inserted advice service (figure 2). This activity sends to the client a message called *execute*. This message advertise the client about the execution of a new behaviour. It encapsulates two kinds of information: the identifier of the advice service and its corresponding interaction protocol. This message is necessary since the new behaviour can require new information exchange involving messages not expected by the client and leading to execution failures. At the client implementation, the developer has to handle this type of message: it has to extract the interaction protocol of the *advice service* and integrate it in its behaviour. This part is detailed in the next section.

3.1. Extended BPEL generator

These previous concepts are concretisized through the architecture of our tool named *extended BPEL generator*. The tool contains the following components (figure 3):

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.

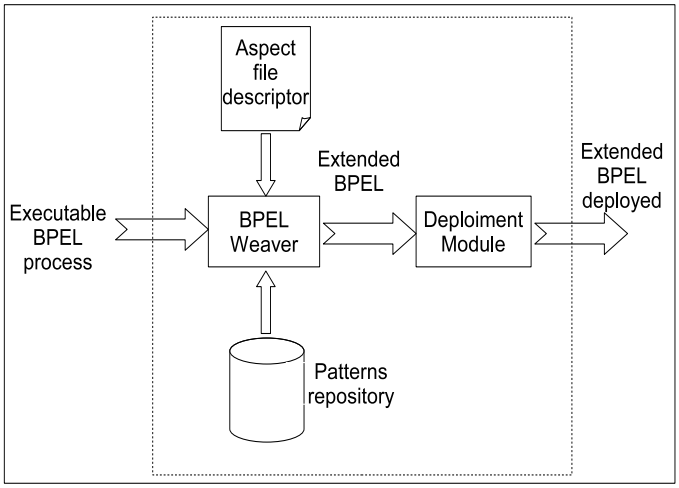

FIGURE 3. The extended BPEL generator.

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 4 shows the transformations made on the base process $sequence(receive(ResReq), switch(reply(ResResp), reply(error))$ 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 4.d), the specified $joinpoint$ is replaced by the advice service and the *execute* 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.

3.2. 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).

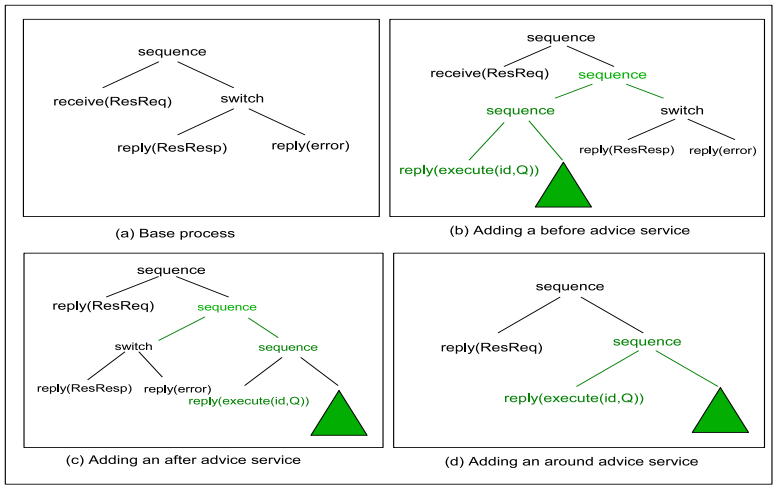

FIGURE 4. Syntactic transformations on the base executable BPEL process.

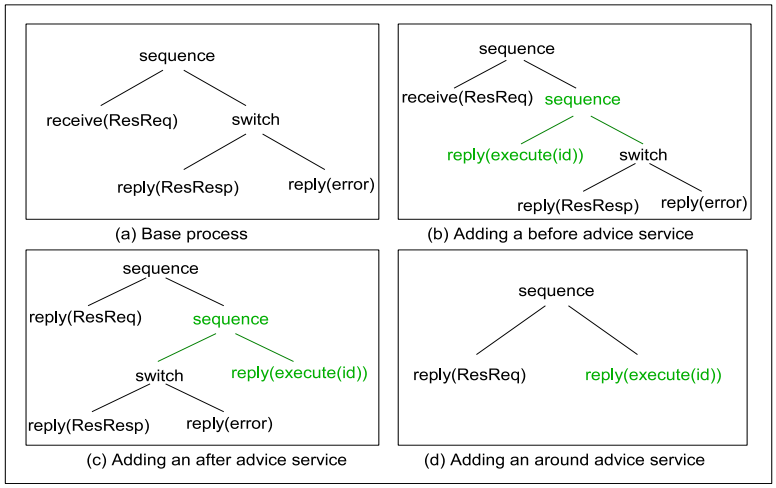

FIGURE 5. transformations on the syntactic BPEL process tree.

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 5). 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.

4. Generating dynamic clients

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].

However, time is explicitly present in some of the BPEL constructors and thus the standard process algebra semantics are inappropriate for the description of such process. Thus, we defined a new process algebra semantics that associates a timed automaton (TA) [19] with an abstract process [11]. 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 designing an algorithm that generates a client automaton which is in an interaction relation with the WS.

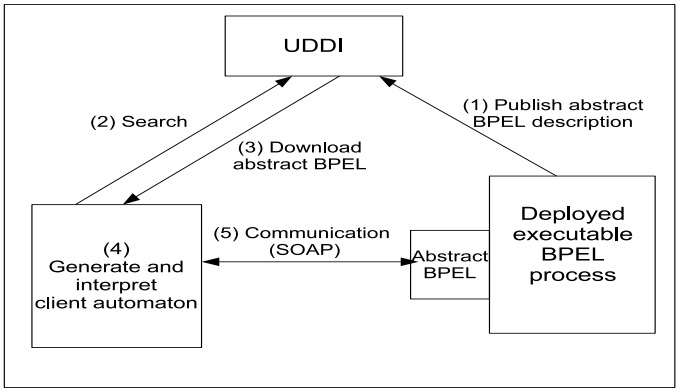

FIGURE 6. Generic client interpreter.

The client automaton is interpreted by our generic client interpreter (figure 6). Our client downloads the abstract BPEL process from an UDDI registry and generates its corresponding TA. Then, based on the TA of the service and the interaction relation, it generates the client TA if the service is not ambiguous. Finally, it executes the client TA and displays graphical interfaces allowing to the human user to enter the messages parameters.

4.1. 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 TA.

2. It simultaneously executes the client TAs of the main process and its *advices clients* TA.
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 the sending and receiving processes in order to handle the $execute(id)$ messages.

4.2. Formalisation steps

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) (not observable from the client side), raise of exceptions ($e \in E$), expiration of timeout (to) and the termination of the process ($\surd$). We distinguish three kinds of actions: the immediate actions corresponding to a logical transition ($\tau, e, \surd$), the asynchronous actions where an unknown amount of time elapses before the occurrence of actions ($?m, !m$) and the synchronous actions (to) which occur after a fixed delay.

Now, we present some operational rules and precisely the new rules for the sending and receiving processes. To see all rules and in particular the handling of clocks in TA, the reader is invited to refer to [11].

For example, the *empty* process which represents the process that does nothing can only terminate by executing the $\surd$ action (0 is the *null* process).

$$empty \xrightarrow{\surd} 0 \tag{4.1}$$

For the sending and receiving processes, we define the following rules.

$$\forall m \neq execute$$
$$*o[m] \xrightarrow{*m} empty \; avec * \in \{?, !\} \tag{4.2}$$

$$!o[m] \xrightarrow{!execute(id)} WaitAdvice(id) \tag{4.3}$$

$$WaitAdvice(id) \xrightarrow{id.\surd} empty \tag{4.4}$$

Rule 4.2 states that 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. In the case of sending an *execute* message, the automaton evolves to an intermediary state named $WaitAdvice(id)$ (rule 4.3). $WaitAdvice(id)$ 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 4.4).

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

termination action and becomes P'. If P terminates and Q can execute an action a and becomes Q', the process $P;Q$ executes the action a then becomes the process Q'.

$$\forall a \neq \surd \frac{P \xrightarrow{a} P'}{P;Q \xrightarrow{a} P';Q} \tag{4.5}$$

$$\frac{P \xrightarrow{\surd} \ and \ Q \xrightarrow{a} Q'}{P;Q \xrightarrow{a} Q'} \tag{4.6}$$

Finally, the $switch\{P_i\}_{i\in I}$ process evaluates an internal condition represented by τ then becomes the process P_i.

$$\forall\ i\ \in\ I,\ switch\{P_i\}_{i\in I} \xrightarrow{\tau} P_i \tag{4.7}$$

4.3. Execution Scenario

Considering the abstract BPEL process defined in section 2. 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.

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

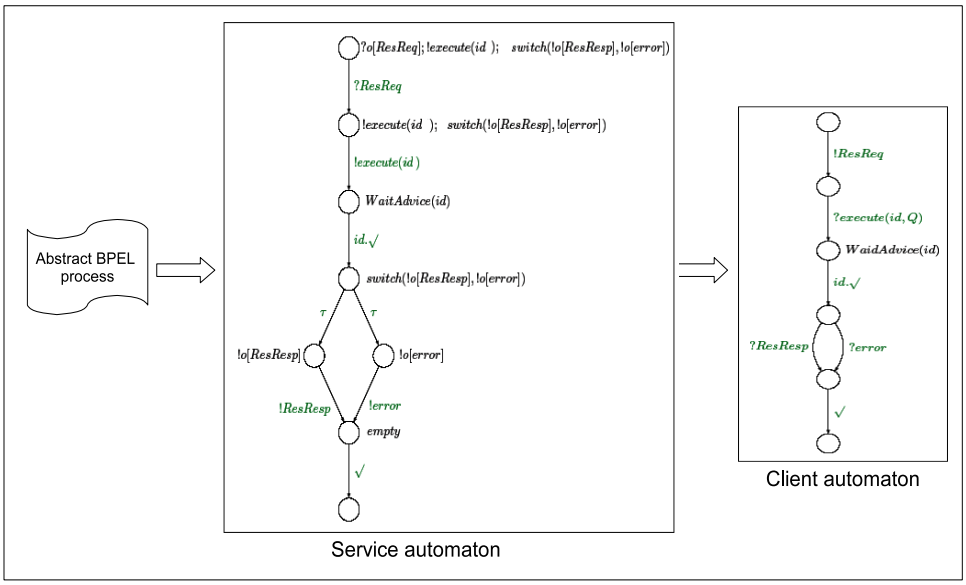

FIGURE 7. Adaptable service and client automata

At the execution, our dynamic client interpreter downloads the extended abstract BPEL specification. Then, it generates the corresponding service TA based

on the operational rules previously defined. Then, based on the service TA and the interaction relation, our client generates the client TA and begins its interpretation. Figure 4.3 shows the generation process.

When our client receives an $execute(id)$ 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 $!o[authDataRequest]\ ;\ ?o[authDataResp]\ ;\ P1$. 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. Our 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, our client makes synchronisation with the main client automaton. it deletes the *advice client*, performs the $id.\surd$ action and continues the execution of the main client automaton (figure 8.(right)).

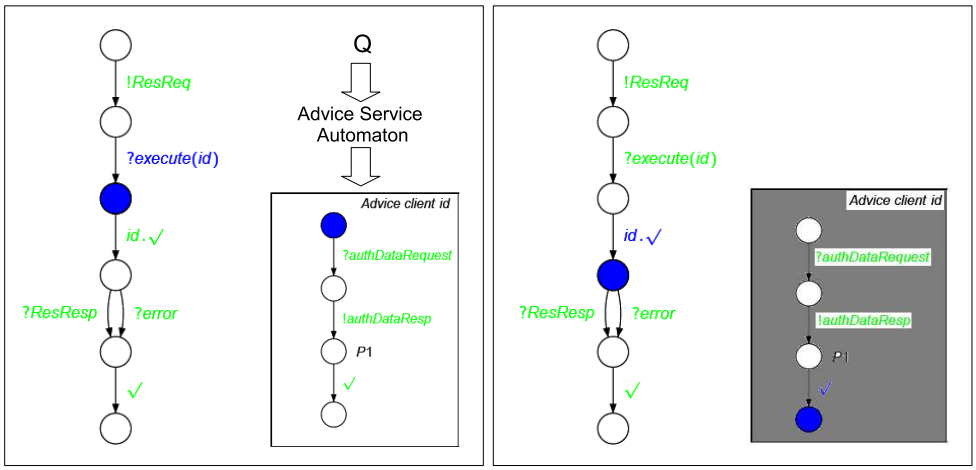

FIGURE 8. Reception of an $execute(id, Q)$ message (left) and the terminaison of an advice service (right)

5. Related works

In [12] and [13], 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) [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. 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.

Some proposals have emerged recently to abstractly describe WSs, most of them are grounded on transition system models (Labelled Transition Systems, Petri nets, etc.) [26, 27, 28]. These works propose to formally specify composite WSs and handle the verification and the automatic composition issues. But, neither of these works propose to formalize the dynamics of SOA architectures and to handle runtime interaction changes.

6. Conclusion

In this paper, we proposed a solution based on AOP and PA to handle dynamic changes in the WS context. 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 extend the work to take into account the client execution context. We also want to formally handle the aspects interactions issue (aspects applied at the same joinpoint). Finally, we plane to improve the current ASW prototype 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). Available at: http://www.w3.org/XML/.

[3] Web Services Architecture, W3C Working Draft 14 November 2002. Available at: http://www.w3.org/TR/ws-arch/

[4] Andrews, T. et al., Business process execution language for web services (2003). Available at http://www-128.ibm.com/developerworks/library/specification/ws-bpel/

[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). Available at: http://www.w3.org/TR/xpath

[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).

Acknowledgment

Many thanks to our TEX-pert for developing this class file.

Mehdi Ben Hmida, Céline Boutrous Saab, Serge Haddad, Valérie Monfort
LAMSADE-CNRS, Université Paris-Dauphine,
Place du Maréchal de Lattre Tassigny,
Paris Cedex 16, France
e-mail: {mehdi.benhmida, haddad, celine.boutrous-saab}@lamsade.dauphine.fr

Ricardo Tomaz Ferraz
CRI, Université Paris 1 Sorbonne,
90 rue de Tolbiac,
75013 Paris, France
e-mail: {valerie.monfort, ricardo.ferraz-tomaz}@univ-paris1.fr