A Case Study in Model-based Adaptation of Web Services

Javier Cámara[1], José Antonio Martín[2], Gwen Salaün[3], Carlos Canal[2], and Ernesto Pimentel[2]

[1] INRIA Rhône-Alpes, France
Javier.Camara-Moreno@inria.fr
[2] Department of Computer Science, University of Málaga, Spain
{jamartin,canal,ernesto}@lcc.uma.es
[3] Grenoble INP-INRIA-LIG, France
Gwen.Salaun@inria.fr

Abstract. Developing systems through the composition of reusable software services is not straightforward in most situations since different kinds of mismatch may occur among their public interfaces. Service adaptation plays a key role in the development of such systems by solving, as automatically as possible, mismatch cases at the different interoperability levels among interfaces by synthesizing a mediating adaptor between services. In this paper, we show the application of model-based adaptation techniques for the construction of service-based systems on a case study. We describe each step of the adaptation process, starting with the automatic extraction of behavioural models from existing interface descriptions, until the final adaptor implementation is generated for the target platform.

1 Introduction

The widespread adoption of Service-Oriented Architectures in the last few years has fostered the need to develop complex systems in a timely and cost-effective manner by assembling reusable software services. These can be considered as blocks of functionality which are often developed using different technologies and platforms. On the one hand, SOA enables developers to build applications almost entirely from existing services which have already been tested, resulting in a speed-up of the development process without compromising quality. On the other hand, the potential heterogeneity of the different services often results in interoperability issues at different levels which have to be solved by system architects on an ad-hoc basis.

In order to ensure interoperability, service interfaces must provide a comprehensive description of the way in which they have to be accessed by service consumers. Composition of services is seldom achieved seamlessly because mismatch may occur at the different interoperability levels (*i.e.,* signature, interaction protocol/behaviour, quality of service, and functional). *Software adaptation* [6,17] is a recent discipline which aims at generating, as automatically as possible, mediating services called *adaptors*, used to solve mismatches among services in a non-intrusive way.

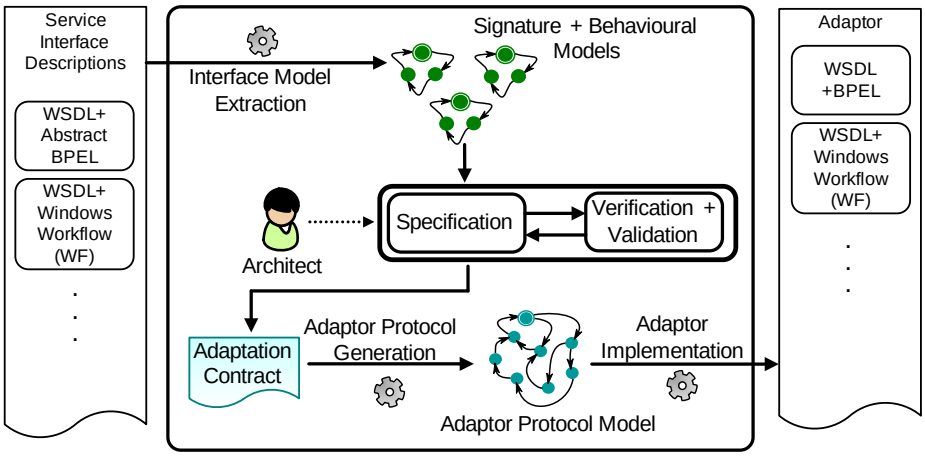

Fig. 1. Generative adaptation process

So far, most adaptation approaches have assumed interface descriptions that include signatures (operation names and types) and behaviours (interaction protocols). Describing protocols in service interfaces is essential because erroneous executions or deadlock situations may occur if the designer does not consider them while composing services.

In this paper, we show the application of model-based adaptation techniques [5,12] (see Figure 1) for Web services, focusing on a case study that we use to illustrate the different steps in the approach. The process starts with the automatic extraction of behavioural service models from existing interface descriptions. These descriptions include a WSDL specification of the different operations made available at the service interface, as well as a specification of the behaviour of the service which can be given in languages such as Abstract BPEL, or Windows Workflows. Next, the designer can build an adaptation contract, which is an abstract specification of how mismatch cases among the different service interfaces can be solved. This is not a trivial task, therefore we propose a graphical representation and an interactive environment to guide the designer through the process. Once the adaptation contract is built, its design can be validated and verified using techniques which enable the visual simulation of the execution of the system step-by-step, finding out as well which parts of the system lead to erroneous behaviour (deadlocks, infinite loops, violation of safety properties, etc.). In such a way, the designer can check if the behaviour of the system complies with his/her intentions. Once the designer is satisfied with the design, an adaptor protocol model can be generated and implemented into an actual adaptor which can be deployed in the target platform.

The rest of this paper is structured as follows: first, we present in Section 2 a description of the case study that we use to illustrate the different steps of the development process in the remaining sections. Section 3 presents an overview of the adaptation process for our case study, starting with the extraction of behavioural models from existing

interface descriptions of the services that we intend to reuse in the system given in WSDL and Abstract BPEL (Section 3.1). Section 3.2 illustrates the contract specification process for our case study. Sections 3.3 and 3.4 describe the adaptor protocol generation and its implementation into an actual adaptor using BPEL as target language, respectively. Section 4 concludes the paper.

2 Case Study: Online Medical Management System

We present a case study in the context of a health care organization, which describes the development of a management system which is required to handle online patient appointments with general practitioners, as well as with specialist doctors in the organization. In particular, the system must be able to create appointments for valid system users who are provided with a username and an access password. After logging in to the system, the user must be able to request an appointment for a given date either with a general practitioner, or with a specialist doctor. After checking doctor availability, the system will return a ticket identifier to the user that corresponds to the provided appointment. If the system does not find a time slot for the user request, the user should be allowed to perform additional requests for further appointments.

In order to build this new system, we aim at reusing two existing sub-systems whose functionality is exposed through different services:

- Service ServerDoc handles appointments with general practitioners.
- Service ServerEsp handles appointments with specialist doctors.

Furthermore, we also reuse a client that implements an example of user requirements. It is worth observing that this client enables the user to perform requests both to general practitioners and specialist doctors in any arbitrary order, whereas a new policy within the organization establishes that users should not be allowed to schedule appointments with specialist doctors without a prior appointment with a general practitioner. Hence, this is an important requirement that the system resulting from our service composition must meet.

3 Overview of the Adaptation Process

3.1 Interface Model Extraction

We assume that service interfaces are specified using both a signature and a protocol. Signatures correspond to operation names associated with arguments and return types relative to the messages and data being exchanged when the operation is called. Protocols are represented by means of Symbolic Transition Systems (STSs), which are Labelled Transition Systems (LTSs) extended with value passing [15]. This formal model has been chosen because it is simple, graphical, and provides a good level of abstraction to tackle verification, composition, or adaptation issues [9,10,16].

At the user level, developers can specify service interfaces (signatures and protocols) using respectively WSDL, and Abstract BPEL (ABPEL) or WF workflows (AWF) [7].

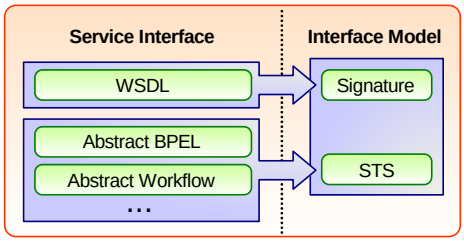

Fig. 2. Interface model extraction.

In order to build the interface models of the services and the client to be reused in the system, we parse their WSDL descriptions and generate the corresponding signatures. Moreover, we can generate the behavioural part of the model (STSs) from service interfaces specified using ABPEL or AWF. To ease the addition of other possible notations to describe service interfaces, we use as an intermediate step in this parsing process an abstract Web services class (AWS). Thus, one can add as a front-end another description language with its parser to AWS, and take advantage of the existing parser from AWS to our model (see Figure 2).

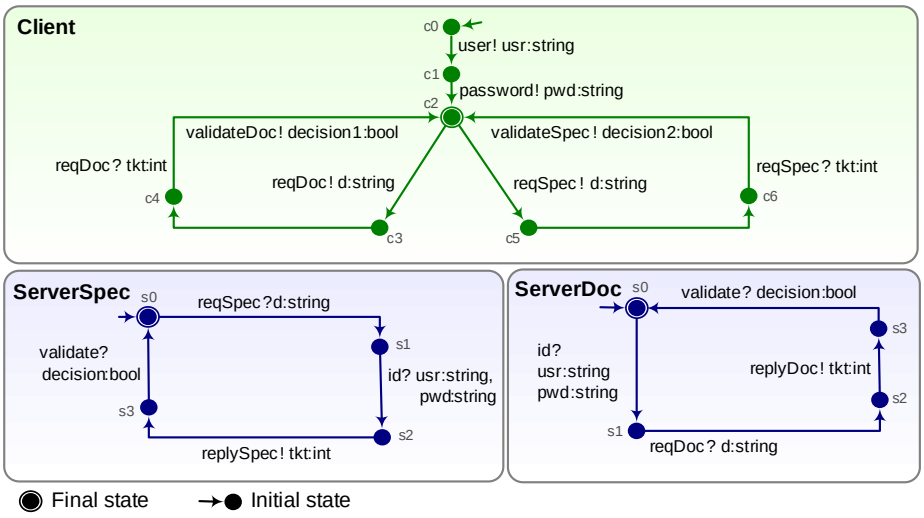

Fig. 3. Behavioural models for the different services and the client.

Example. The STSs depicted in Figure 3 are obtained after the application of the parsing process to each of the elements of the (running) example.

- The Client can first log on to a server by sending respectively his/her user name (user!) and password (password!). Then, depending on his/her preferences, the client can stop at this point, or ask for an appointment either with a general practitioner (reqDoc!) or a specialist doctor (reqSpec!), and then receive an appointment identifier. Finally, the client can accept or reject the appointment obtained if it is not convenient for him/her (validateDoc/validateSpec!).
- Service ServerDoc first receives the client user name and password (id?). Next, this service receives a request for an appointment with a general practitioner (reqDoc?)

and replies (replyDoc!). Finally, the service waits for an acknowledgement from the user either accepting or rejecting the provided appointment (validate?).
– Service ServerSpec first receives a request for an appointment with a specialist doctor (reqSpec?), followed by the client user name and password (id?). After checking doctor availability for the given date, an appointment identifier is returned (replySpec!) to the client. As it happened in the case of the ServerDoc service, ServerSpec finishes its interaction waiting for an acknowledgement from the user either accepting or rejecting the provided appointment (validate?).

The composition of the different services in our example is subject to different mismatch situations among their interfaces:

– **Name mismatch** occurs if a service expects a particular message, while its counterpart sends one with a different name (*e.g.*, service ServerDoc sends replyDoc!, whereas the client is expecting reqDoc?).
– **N to M correspondence** appears if a message on a particular interface corresponds to several ones in its counterpart's interface (or similarly, a message has no correspondence at all). In Figure 3 it can be observed that while the client intends to log in to a service sending user! and password! subsequently, service ServerDoc expects only message id? for authentication.
– **Incompatible order of messages.** The relative order of operation invocations among the different protocols involved is not compatible. We may observe this in our example when the client first sends its authentication information and then requests an appointment with a specialist doctor, whereas the ServerSpec service expects these messages in the inverse order.
– **Argument mismatch** may occur when the number and/or type of arguments either being sent or received do not match between the operations on the different interfaces. This can be observed in ServerDoc, when id? expects both a username (usr) and a password (pwd). The first data term corresponds to user! on the client interface, whereas the second belongs to password!.

3.2 Adaptation Contract Specification

Once the interface models have been extracted from the WSDL and ABPEL descriptions, we can use them to produce the adaptation contract for our system. In particular, we use *vectors* and a *vector LTS* (VLTS) as adaptation contract specification language [14,6,12]. A vector contains a set of events (message, direction, list of parameters). Each event is executed by one service, and the overall result corresponds to one or several interactions between the involved services and the adaptor. Vectors express correspondences between messages, like bindings between ports, or connectors in architectural descriptions. In particular, we consider a binary communication model, therefore our vectors are always reduced to one event (when a service evolves independently) or two (when services communicate indirectly through the adaptor). Furthermore, variables are used as placeholders in message parameters. The same variable names appearing in different labels (possibly in different vectors) relate sent and received arguments in the messages.

In addition to vectors, the contract notation includes a Labelled Transition System (LTS) with vectors on transitions (that we call vector LTS or VLTS). This element is used as a guide in the application order of interactions specified by vectors. VLTSs go beyond port and parameter bindings, and express more advanced adaptation properties (such as imposing a sequence of vectors or a choice between some of them). If the application order of vectors does not matter, the VLTS contains a single state and all transitions looping on it.

$$
\begin{array}{lcl}
\{v_{user} & = & \langle c : \mathsf{user!U} \rangle, \\
v_{pwd} & = & \langle c : \mathsf{password!P} \rangle, \\
v_{vidsp} & = & \langle ss : \mathsf{id?U,P} \rangle, \\
v_{viddoc} & = & \langle sd : \mathsf{id?U,P} \rangle, \\
v_{reqsp1} & = & \langle c : \mathsf{reqSpec!DATE}; ss : \mathsf{reqSpec?DATE} \rangle, \\
v_{reqsp2} & = & \langle c : \mathsf{reqSpec?RES1}; ss : \mathsf{replySpec!RES1} \rangle, \\
v_{reqdoc1} & = & \langle c : \mathsf{reqDoc!DATE}; sd : \mathsf{reqDoc?DATE} \rangle, \\
v_{reqdoc2} & = & \langle c : \mathsf{reqDoc?RES2}; sd : \mathsf{replyDoc!RES2} \rangle, \\
v_{valid1} & = & \langle sd : \mathsf{validate?B1}; c : \mathsf{validateDoc!B1} \rangle, \\
v_{valid2} & = & \langle ss : \mathsf{validate?B2}; c : \mathsf{validateSpec!B2} \rangle \ \}
\end{array}
$$

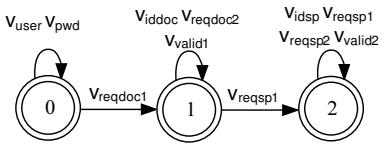

Fig. 4. Adaptation contract for our example: vectors (left) and VLTS (right).

Example. Going back to our on-line medical management system described in Section 2, let us recall that we intend to compose our services into a working system where the client can request an appointment with a general practitioner, or also request an appointment with a specialist doctor, provided that there is a previous appointment with a general practitioner (*i.e.*, the client cannot directly schedule an appointment with a specialist).

Figure 4 displays the set of vectors used to solve mismatch among our interfaces. In order to understand how vectors relate messages on the different interfaces, let us focus on v_{reqsp2}, for instance. Here we may observe that the message reqSpec? on the client interface is related with the replySpec! message on the ServerSpec interface the appointment ticket identifier argument on both messages is related by placeholder RES1 (please refer to Figure 7 for more details about how placeholders relate message arguments). However, expressing correspondences between messages is not always as straightforward as in the previous example. We may now focus on the initial part of the composition, where we want to connect the general practitioner server (ServerDoc) with the client, and make authentication work correctly. For this, we need three vectors, respectively v_{user}, v_{pwd} and v_{viddoc}, in which we solve existing mismatches by relating different message names (id, user and password). Here, we first specify the independent evolution of the client through user! and password! (this is expressed by vectors v_{user} and v_{pwd}, which only contain one message). Next, we also specify the independent evolution of ServerDoc through v_{viddoc}. Exchanged data parameters among the three involved messages in the vectors are connected using placeholders U and P.

Regarding the specification of additional constraints on the composition, we can observe on the right-hand side of Figure 4 that the VLTS for the contract constrains the

interaction of the Client, ServerDoc, and ServerSpec interfaces by imposing the request for an appointment with a general practitioner ($v_{reqdoc1}$) always before the request of an appointment with a specialist doctor (v_{reqsp1}). This is achieved by excluding v_{reqsp1} from the possible transitions in state 0, and including the transition $(0, v_{reqdoc1}, 1)$. □

In order to make the specification as simple and user-friendly as possible, we employ interactive specification techniques to support the architect through this process. To this purpose, we use a notation to graphically make explicit bindings between ports using an interactive environment that enables graphical contract construction and verification called ACIDE. The graphical notation for a service interface includes a representation of its behavioural model (STS) and a collection of ports. Each label on the STS corresponds to a *port* in the graphical description of the interface. Ports include a *data port* for each parameter contained in the parameter list of the label. Figure 5 summarizes ports and bindings used in our notation.

Correspondences between the different service interfaces are represented as *port bindings* and *data port bindings* (solid and dashed connector lines, respectively). Starting from the graphical representation of the interfaces, the architect can build a contract between them by successively connecting ports and data ports. This results in the creation of bindings which specify how the interactions should be carried out between the services. It is also possible to add a T-shaped *port cap* on a port in order to indicate that it does not have to be connected anywhere.

Fig. 5. Graphical notation: ports and bindings

The VLTS imposing an order on the application of the bindings is built implicitly in ACIDE as new bindings are created by the designer. Initially, the VLTS has a single state and no transitions. Each time a new connection is made, the VLTS is extended in a different way, depending on the current VLTS extension mode selected by the user:

- **Abstract mode.** No order on the application of the bindings is imposed. Creating a binding in this mode results in the creation of a transition looping on the current state in the VLTS.
- **Sequential mode.** Bindings created in this mode must be executed one after the other. This results in the extension of the VLTS with a fresh state and a transition from the current state to the new one. Once this transition is added, the newly created state becomes the current VLTS state.
- **Branching mode.** Bindings created in this mode are mutually exclusive. The VLTS is extended in this case with a fresh state and a transition to it from the current state. Unlike in sequential mode, the current state is not updated.

Using this implicit method it is possible to build a VLTS for most contracts. However, the user is also able to directly manipulate the VLTS from within the graphical environment in order to adjust it to particular situations such as a binding representing an interaction which has to be carried out more than once in different parts of the specification.

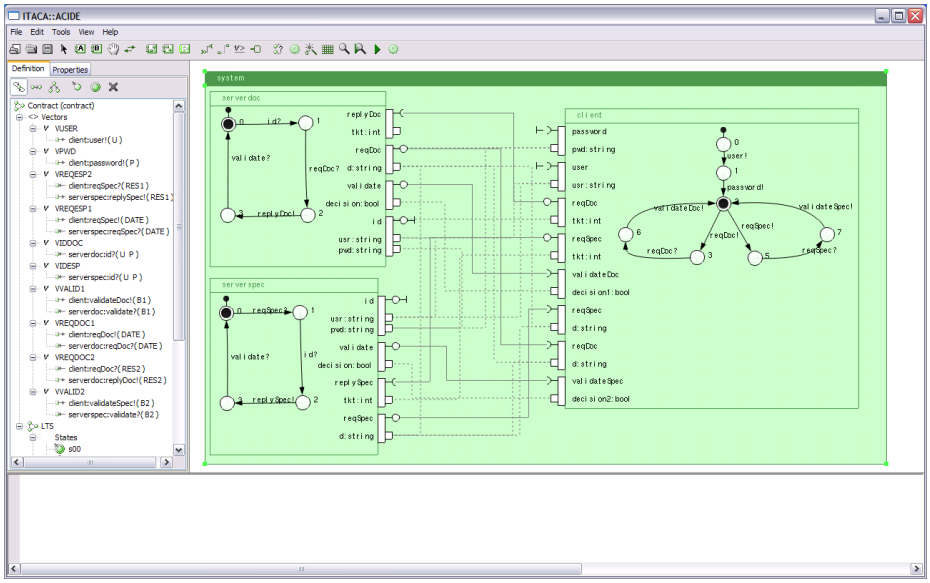

Fig. 6. Contract specification for the Online Medical System in ACIDE

During the specification of the contract, we can also make use of a set of validation and verification techniques to check that an adaptation contract makes the involved services work correctly. These techniques are intended to help the designer in understanding potential problematic behaviours of the system which are not obvious (even to the trained eye) just by observing service interaction protocols and adaptation contracts. These problems may include potential deadlocks, as well as unintended interactions that are not explicitly addressed at the contract level, which is only an abstract specification of the adaptation and does not take into account every interaction scenario among services. These techniques are completely automated, and include four kinds of checks: (i) static checks on the contract *wrt.* STS service interfaces involved, (ii) simulation of the system execution, (iii) trace-checking to find potential deadlocking executions and infinite loops, and (iv) verification of temporal logic formulas.

Example. Figure 6 shows a screenshot of the graphical representation of our final adaptation contract specification for the medical management system in ACIDE. If we focus on the graphical representation of the ServerDoc, it can be observed that it contains a port for the reception of the reqDoc? request with a data port attached representing the date, and another port for the emission of the replyDoc response with a data port attached representing the ticket identifier issued for the given date. Moreover, it can be observed that the interactions expressed by vectors in our contract are represented by port bindings in the graphical environment.

3.3 Generation of the Adaptor Protocol

Being given a set of service interfaces and an adaptation contract, an adaptor protocol can be generated using automatic techniques as those presented in [12]. An adaptor is a third-party component that is in charge of coordinating the services involved in the system with respect to the set of constraints defined in the contract. Consequently, all the services communicate through the adaptor, which is able to compensate mismatches by making required connections as specified in the contract.

From adaptor protocols, either a central adaptor can be implemented, or several service wrappers can be generated to distribute the adaptation and preserve parallelism in the system's execution. In the former case, the implementation of executable adaptors from adaptor protocols can be achieved for instance using Pi4SOA technologies [1], or techniques presented in [12] and [7] for BPEL and Windows Workflow Foundation, respectively. In the latter case, each wrapper constrains the functionality of its service to make it respect the adaptation contract specification [15].

Adaptor and wrapper protocols are automatically generated in two steps: (i) system's constraints are encoded into the LOTOS [11] process algebra, and (ii) adaptor and wrapper protocols are computed from this encoding using on-the-fly exploration and reduction techniques. These techniques are platform-independent, therefore while exploring the state space, all the behaviours (interleaving) respecting the adaptation constraints are kept in the adaptor model. The reader interested in more details may refer to [12,15].

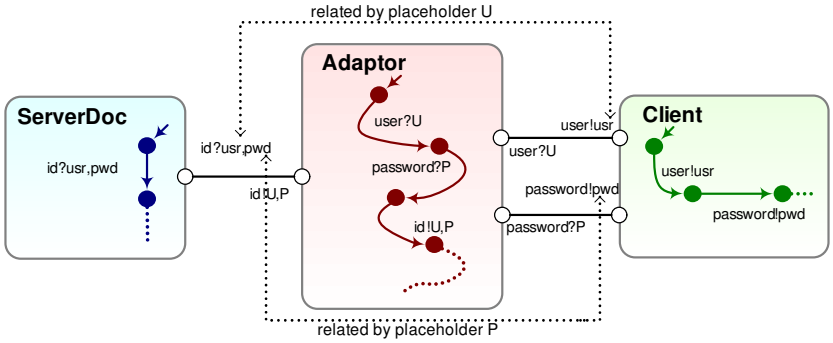

Fig. 7. Example of adaptation for authentication mismatches

Example. Figure 7 shows a small portion of the adaptor protocol generated from the three vectors $v_{user} = \langle c : \mathsf{user!U} \rangle$, $v_{pwd} = \langle c : \mathsf{password!P} \rangle$ and $v_{viddoc} = \langle sd : \mathsf{id?U,P} \rangle$ given in Figure 4. This makes service ServerDoc and the Client interact correctly. We emphasize that the adaptor synchronizes with the services using the same name of messages but the reversed directions, *e.g.*, communication between id? in ServerDoc and id! in the adaptor. Furthermore, when a vector includes more than one communication action, the adaptor always starts the set of interactions formalized in the vector with the

receptions (which correspond to emissions on service interfaces), and next handles the emissions.

Figure 8 displays the adaptor protocol generated using the adaptation contract shown in Figure 4. This adaptor contains 51 states and 73 transitions, and therefore has reasonable size and complexity. Interaction starts by receiving the user?, password? and reqDoc? messages sent by the Client. Next, the adaptor starts interacting with ServerDoc by first sending authentication information (id!) and then the request posted previously by the client (reqDoc!). Once the client and the doctor have ended their interaction, the adaptor can send a request to interact with the specialist (reqSpec?), this is the case in state 25 for example. Note that in the bottom part of the adaptor protocol, corresponding to the interaction with the specialist, the adaptor can treat several specialist requests (*e.g.*, reqSpec? in state 31). Notice also that the adaptor can terminate at different points of its execution (transitions labelled with FINAL). The adaptor protocol corrects all the mismatch cases presented in Section 3.1, for instance we can see in state 33 that the adaptor can submit first the request to the specialist (reqSpec!) and then the authentication information in state 38 (id!) solving the reordering problem existing between the client and the specialist service.

If we consider the adaptation contract with vectors only (the corresponding VLTS consists of a single state with all vector transitions looping on it), the adaptor protocol consists of 243 states and 438 transitions. This quite high number of states and transitions is due to the release of constraints specified in the original VLTS which imposes sequentiality on the system (interactions first with the doctor and in a second step with the specialist), thus reducing interleaving.

3.4 Implementation

Our internal model (STS) can express some additional behaviours (interleaving) that cannot be implemented into executable languages (*e.g.,* BPEL). To make platform-independent adaptor protocols implementable *wrt.* a specific platform we proceed in two steps: (i) filtering the interleaving cases that cannot be implemented, and (ii) encoding the filtered model into the corresponding implementation language.

Filtering. Techniques presented in this paper to generate adaptor protocols are platform-independent, therefore the adaptor model may contain parts which cannot be implemented in a given language (*e.g.*, BPEL). This filtering step aims at removing in the adaptor protocol these non-implementable parts. These parts represent the interleaving of parallel operations and they are not necessary for the functionality of the system. As an example, if there are several emission transitions going out from the same state, this leads to non-determinism which cannot be implemented in BPEL. One of the filtering rules states that in such a case, only one emission is kept. In this paper, we reuse filtering rules presented in [12]. We show in Figure 9 the filtered adaptor protocol. One can see that the protocol contains first a sequence corresponding to the interaction with ServerDoc, and next a loop corresponding to the interaction with ServerSpec. The client must start interacting with ServerDoc, and in state 11, (s)he can choose either to interact again with ServerSpec, or to stop. This protocol can be implemented in BPEL as we will see in the remainder of this section.

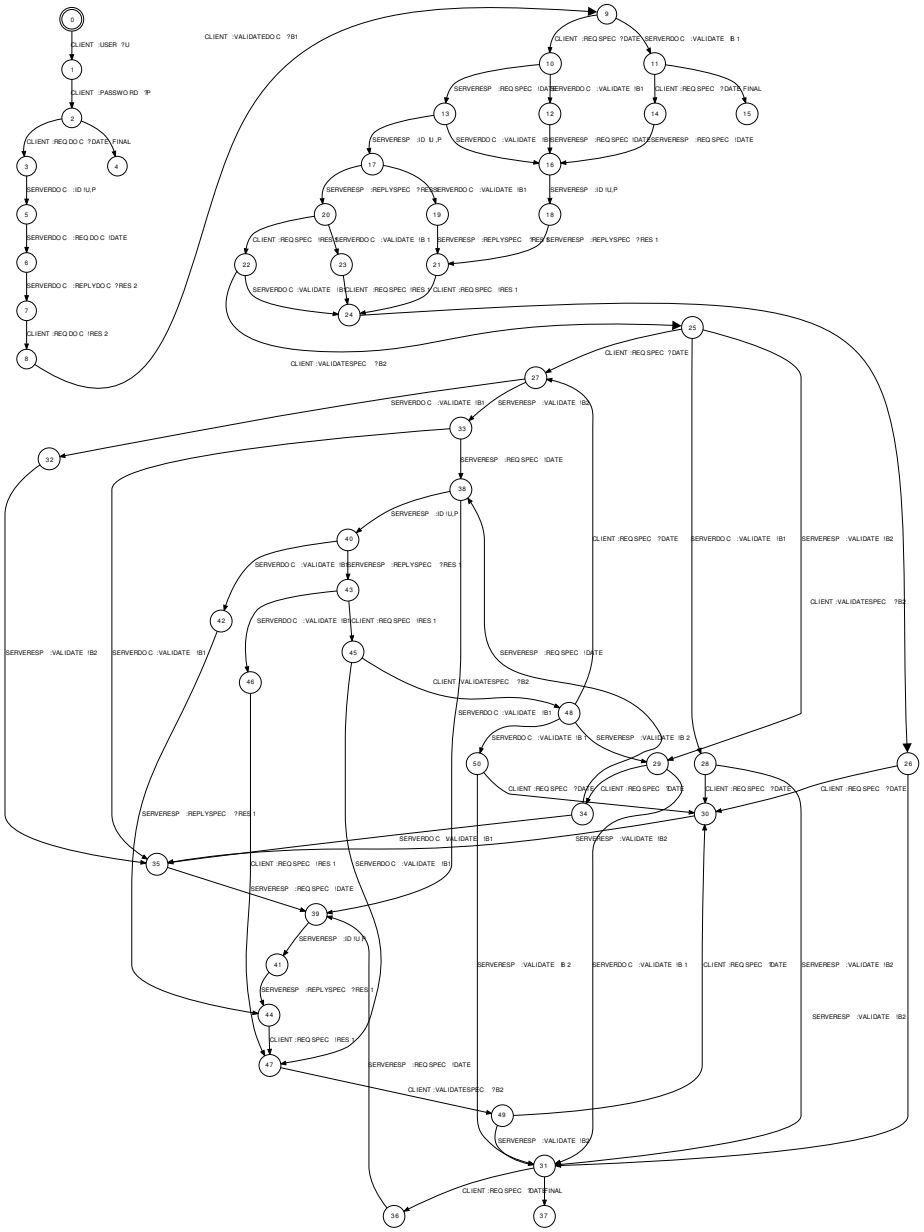

Fig. 8. Adaptor protocol generated for the Online Medical System example

BPEL implementation. The adaptor protocol is implemented using a state machine pattern. The main body of the BPEL process corresponds to a global *while* activity with

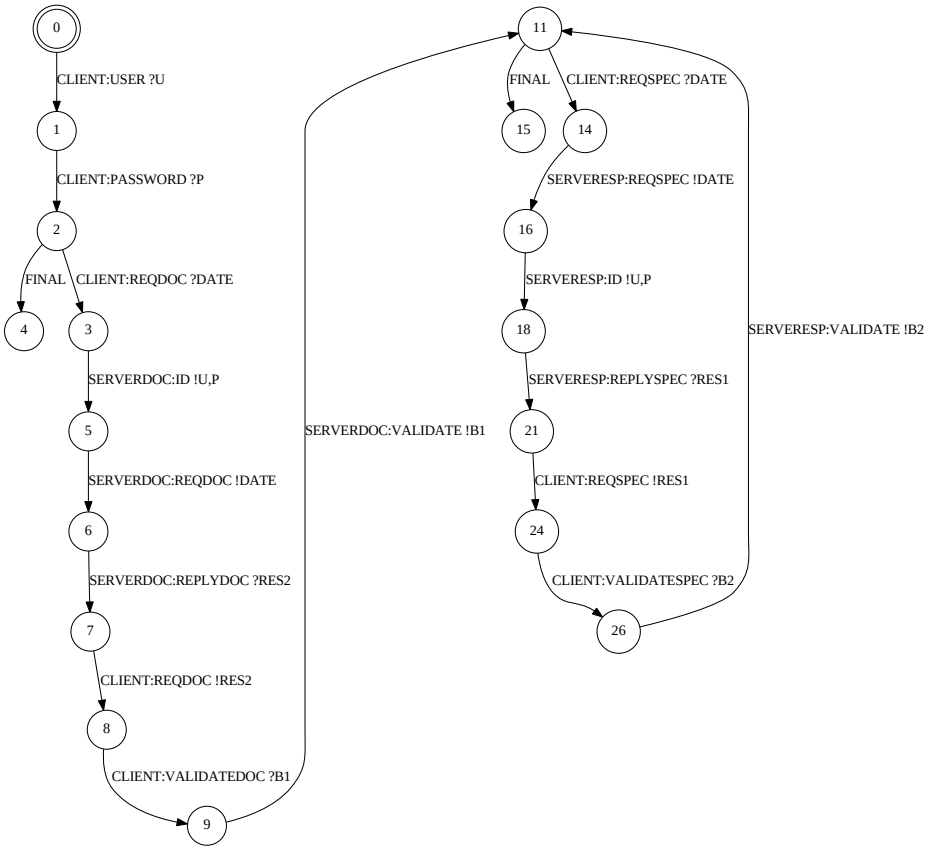

Fig. 9. Filtered adaptor protocol obtained for the Online Medical System example

if statements used inside it to encode adaptor states and *pick* statements to choose which branch to follow depending on the received messages. Variables are used to store data passing through the adaptor and the current state of the protocol. Timeouts (*onAlarm* activities) are used in these *pick* activities to model FINAL transitions. Every state in the adaptor behaviour with several incoming or outgoing transitions is encoded as a new branch of the *if* activity in the implementation. That branch might contain an internal *pick* activity with sequences of communication activities alternated with assignments to update the variables and change the current state of the execution. Adaptors whose protocol is a global loop beginning with a final state (such as services ServerDoc and ServerSpec) are modelled as a sequence, therefore every iteration of the global loop will be a new instantiation of the adaptor. Let us note that the implementation of the adaptor is constrained by the specifics of the implementation of the services and the actual BPEL engine. These might avoid the proper implementation and execution of the adaptor. For instance, BPEL allows bi-directional and blocking *invoke* activities and

their corresponding *receive* activities, in some scenarios, these could block the adaptor and avoid its proper execution. In addition, current BPEL engines do not fully support BPEL 2.0, for instance, Glassfish 2.1.x does not execute properly BPEL processes with two different *receive* activities of the same operation and partner link. This restricts even further the implementation of the adaptor.

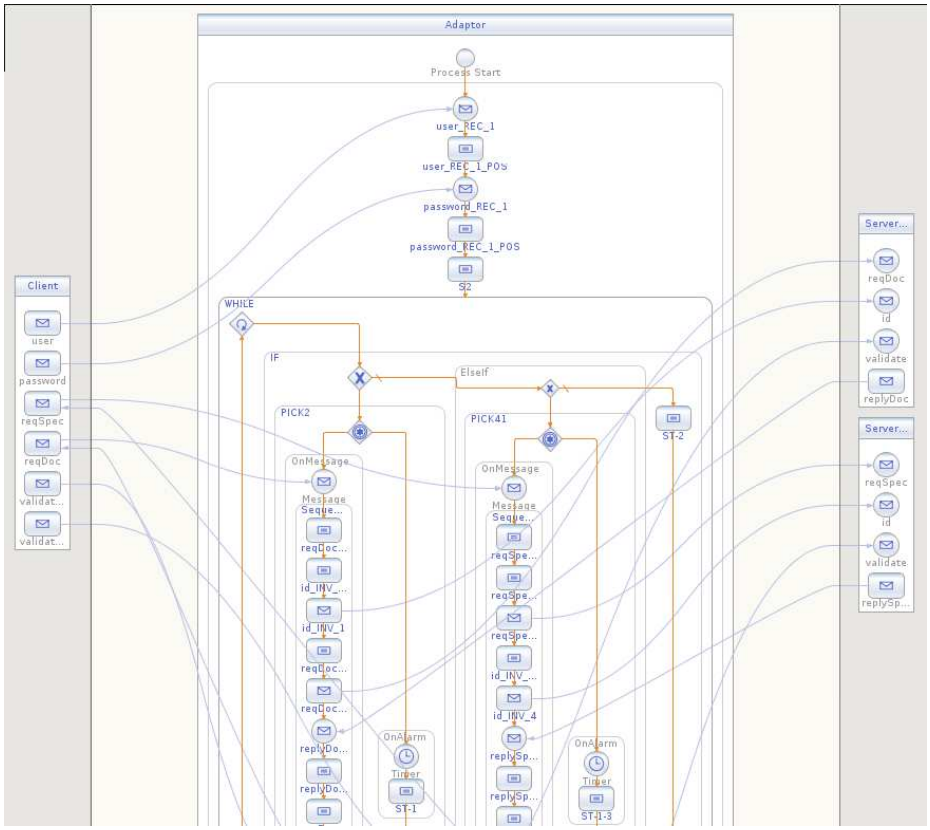

Fig. 10. Implementation of the adaptor in BPEL

Example. Figure 9 shows the behaviour of the filtered adaptor corresponding to the running example, whereas Figure 10 displays a graphical model of part of its implementation in BPEL. The client interface and the intarfaces of the two servers are located on the left-hand side and right-hand side of the BPEL implementation, respectively. This adaptor presents an initial log-in sequence followed by the *while* activity with a nested *if* activity, as previously mentioned. In the first iteration of the loop, the current state of the adaptor is 2 (see Figure 9), therefore the execution continues through the *if* and *pick* activities on the left-hand side of the loop. We have an *onAlarm* which finishes

the session because the client might not perform any request at all. Otherwise, once a request for a general practitioner is processed, the current state of the adaptor is set to be 11 and we iterate once more. In the second iteration, we have an analogous structure for specialists requests in the right-hand side of the loop. In this case, however, the adaptor can process several of such requests (or none) and, when there are not anymore requests, the *onAlarm* activity triggers the end of the session.

4 Concluding Remarks

Building systems by adapting a set of reusable software services whose functionality is accessed through their behavioural interfaces is an error-prone task which can be supported by assisting developers with the automatic procedures and tools supplied by model-based software adaptation. In particular, existing works dedicated to model-based behavioural adaptation are often classified in two families. A first class of existing works can be referred to as *restrictive approaches* [3,4,13], and favour the full automation of the process, trying to solve interoperability issues by pruning the behaviours that may lead to mismatch, thus restricting the functionality of the services involved. These techniques are limited since they are not able to fix subtle incompatibilities between service protocols by remembering and reordering messages and their parameters when necessary. A second class of solution which can be referred to as *generative approaches* [2,6,8] avoid the arbitrary restriction of service behaviour, and supports the specification of advanced adaptation scenarios. Generative approaches build adaptors automatically from an abstract specification of how the different mismatch situations can be solved, often referred to as *adaptation contract*.

In this paper, we have shown the application of model-based adaptation techniques for Web services, using a case study on the development of an online medical management system to illustrate all the steps of the process. The approach used gathers desirable features from existing model-based behavioural adaptation approaches in a single process. All the steps of the process presented in this paper are fully tool-supported by a toolbox called ITACA [5], which enables the specification and verification of adaptation contracts, automates the generation of adaptor protocols, and relates our abstract models with implementation languages.

With respect to the results that we obtained from the application of the approach to this and other case studies, we were able to assess that there is a remarkable reduction both in the amount of effort that the developer has to put into building the adaptor, as well as in the number of errors present in the final result. Since the test cases used so far for our experiments were of a small-medium size and complexity, we think that the difficulty of specifying contracts for bigger systems involving dozens of services would be not manageable by the developers without tool-supported, model-based adaptation techniques. This puts forward the importance of providing such support for the development of service-based systems.

Acknowledgements. This work has been partially supported by the project TIN2008-05932 funded by the Spanish Ministry of Innovation and Science (MICINN), and project P06-TIC-02250 funded by the *Junta de Andalucía*.

References

1. Pi4SOA Project. `www.pi4soa.org`.
2. A. Brogi A. Bracciali and C. Canal. A formal approach to component adaptation. *The Journal of Systems and Software*, 74:45–54, 2005.
3. M. Autili, P. Inverardi, A. Navarra, and M. Tivoli. SYNTHESIS: A Tool for Automatically Assembling Correct and Distributed Component-based Systems. In *Proc. of ICSE'07*, pages 784–787. IEEE Computer Society, 2007.
4. A. Brogi and R. Popescu. Automated generation of BPEL adapters. In *Proc. of ICSOC '06*, volume 4294 of *LNCS*, pages 27–39. Springer, 2006.
5. J. Cámara, J.A. Martín, G. Salaün, J. Cubo, M. Ouederni, C. Canal, and E. Pimentel. ITACA: An Integrated Toolbox for the Automatic Composition and Adaptation of Web Services. In *Proc. of ICSE '09*, pages 627–630. IEEE Computer Society, 2009.
6. C. Canal, P. Poizat, and G. Salaün. Model-based adaptation of behavioural mismatching components. *IEEE Transactions on Software Engineering*, 4(34):546–563, 2008.
7. J. Cubo, G. Salaün, C. Canal, E. Pimentel, and P. Poizat. A Model-Based Approach to the Verification and Adaptation of WF/.NET Components. In *Proc. of FACS'07*, volume 215 of *ENTCS*, pages 39–55. Elsevier, 2007.
8. M. Dumas, M. Spork, and K. Wang. Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation. In *In Proc. of BPM'06*, volume 4102 of *LNCS*, pages 65–80. Springer, 2006.
9. H. Foster, S. Uchitel, and J. Kramer. LTSA-WS: A Tool for Model-based Verification of Web Service Compositions and Choreography. In *Proc. of ICSE'06*, pages 771–774. ACM Press, 2006.
10. X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *Proc. of WWW'04*, pages 621–630. ACM Press, 2004.
11. ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, ISO, 1989.
12. R. Mateescu, P. Poizat, and G. Salaün. Adaptation of Service Protocols using Process Algebra and On-the-Fly Reduction Techniques. In *Proc. of ICSOC'08*, LNCS, pages 84–99. Springer, 2008.
13. H.R.M. Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-automated adaptation of service interactions. In *Proc. of WWW '07*, pages 993–1002. ACM, 2007.
14. P. Poizat and G. Salaün. Adaptation of Open Component-based Systems. In *Proc. of FMOODS'07*, volume 4468 of *LNCS*, pages 141–156. Springer, 2007.
15. G. Salaün. Generation of Service Wrapper Protocols from Choreography Specifications. In *Proc. of SEFM'08*, pages 313–322. IEEE Computer Society, 2008.
16. G. Salaün, L. Bordeaux, and M. Schaerf. Describing and Reasoning on Web Services using Process Algebra. *International Journal of Business Process Integration and Management*, 1(2):116–128, 2006.
17. D.M. Yellin and R.E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 2(19):292–333, 1997.