HAL
open science

Adaptation of Service Protocols using Process Algebra and On-the-Fly Reduction Techniques

Radu Mateescu, Pascal Poizat, Gwen Salaün

To cite this version:

Radu Mateescu, Pascal Poizat, Gwen Salaün. Adaptation of Service Protocols using Process Algebra and On-the-Fly Reduction Techniques. 6th International Conference on Service Oriented Computing ICSOC'2008, Dec 2008, Sydney, Australia. pp.84-99, 10.1007/978-3-540-89652-4_10 . inria-00341598

HAL Id: inria-00341598

https://inria.hal.science/inria-00341598

Submitted on 25 Nov 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Adaptation of Service Protocols using Process Algebra and On-the-Fly Reduction Techniques

Radu Mateescu[1], Pascal Poizat[2,3], and Gwen Salaün[4]

[1] INRIA/VASY project-team, aile de l'Ingénieur, bât. LE2I, Dijon, France
radu.mateescu@inria.fr
[2] INRIA/ARLES project-team, France
pascal.poizat@inria.fr
[3] IBISC FRE 3910 CNRS – Université d'Évry Val d'Essonne, France
[4] University of Málaga, Spain
salaun@lcc.uma.es

Abstract. Software Adaptation is a hot topic in Software Engineering since it is the only way to compose non-intrusively black-box components or services with mismatching interfaces. However, adaptation is a complex issue especially when behavioral descriptions of services are considered. This paper presents optimised techniques to generate adaptor protocols, being given a set of service interfaces involved in a composition and an adaptation contract. In this work, interfaces are described using a signature, and a protocol that takes value passing into account. Our proposal is completely supported by tools that automate the generation and the verification of the adaptor protocols. Last, we show how our adaptation techniques are implemented into BPEL.

1 Introduction

Service composition is a central issue in Service Oriented Computing. Reuse of existing entities is mandatory not to implement again the same blocks of software, and then help developers to reduce development time, respect delays, and have their companies save money by diminishing software design costs. However, direct reuse and composition of existing services is in most of cases impossible because their interfaces present some incompatibilities. *Software Adaptation* [3] is a very promising solution to compose in a non-intrusive way black-box components or (Web) services whose functionality is as required for the new system, although they present interface mismatches. Adaptation techniques aim at automatically generating new components called *adaptors*, and usually rely on an *adaptation contract* which is an abstract description of how mismatches can be worked out. All the messages pass through the adaptor which acts as an orchestrator, and makes the involved services work correctly together by compensating mismatches.

Contributions. Model-based behavioral adaptation approaches are either restrictive or generative. *Restrictive approaches* [5, 2] try to solve the problem by

cutting off (pruning) the behaviors that may lead to mismatch, thus restricting the functionality of the services involved. *Generative approaches* [4, 7] try to accommodate the protocols without restricting the behavior of the services, by generating adaptors that act as mediators, remembering and reordering events and data when necessary. In the current state of the art, restrictive approaches are fully automated and are directly related to programming languages, but they do not support advanced adaptation scenarios. On the other hand, generative approaches suffer from the computational complexity of generating adaptors, often lack of tool support, and are not related to implementation languages. In this paper, we propose model-based adaptation techniques that are both generative and restrictive since we support complex adaptation scenarios (such as message reordering), while removing incorrect behaviors. We also diminish the computational complexity of adaptor generation by using on-the-fly exploration and reduction techniques to avoid the generation of the full state space of the adaptor under construction. Last, let us emphasize that our approach is fully supported by tools we implemented, and adaptors are finally implemented using service implementation languages.

Approach. In this paper, we first present a model of services that makes it possible to describe signatures (operation names and types) and behaviors (interaction protocols). Protocols are essential because erroneous executions or deadlock situations may occur if the designer does not take them into account while building composite services. More than only considering messages exchanged in protocols, it is important to include value passing (parameters) coming with messages since this feature may raise composition issues too (unmatching number of parameters, different ordering, etc). Next, we introduce the contract notation that is used to describe how mismatches appearing in signatures and protocols can be worked out by defining correspondences between messages but also between message parameters. Then, from a set of service protocols and a contract, we present our approach to generate adaptor protocols which relies on (i) encodings into the LOTOS process algebra [14], and (ii) on-the-fly exploration and reduction techniques. Verification of contracts is also possible by using CADP [13] a rich verification toolbox for LOTOS. Last but not least, we show how adaptors can be implemented in the WS-BPEL (BPEL for short) service orchestration language. Our proposal is supported by tools (Fig. 1) that automate the extraction of abstract interfaces from XML description of services (BPEL2STS), the generation of the LOTOS encoding (Compositor), the efficient computation of the adaptor protocol from the LOTOS specification (Scrutator), the verification of the adapted system (Evaluator), and the generation of BPEL from adaptor models (STS2BPEL). The only step of our approach which requires manual intervention is the adaptation contract construction.

Outline. The remainder of this paper is structured as follows. Section 2 presents our model of services. Section 3 introduces the contract notation which is used for adaptation purposes. In Section 4, we present the adaptor generation and verification techniques. Section 5 focuses on adaptor implementation. Section 6

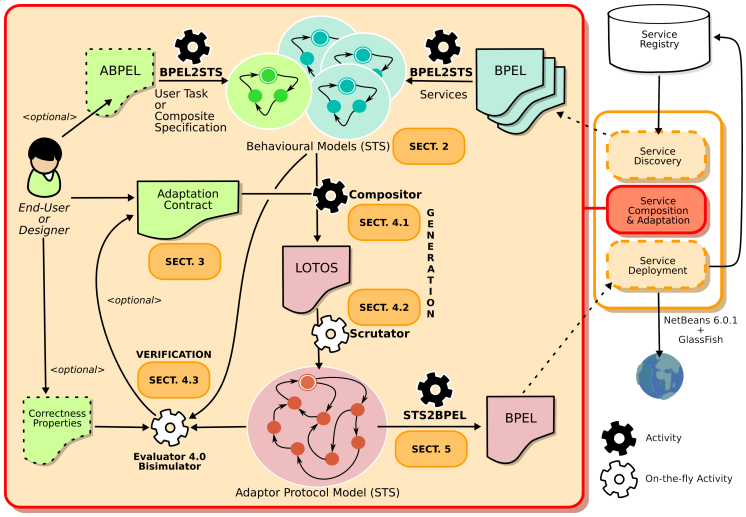

Fig. 1. Overview of our approach

compares our approach to related work, and Section 7 ends the paper with some concluding remarks.

2 Service Model

In this section we present our service interface model. We assume that service interfaces are given using both a signature and a protocol. *Signatures* correspond to operation profiles described using WSDL, *i.e.*, operation names associated with argument and return types relative to the messages and data being exchanged when the operation is called. Additionally, we propose that protocols are represented by means of *Symbolic Transition Systems* (STSs) which are Labelled Transition Systems (LTSs) extended with value passing (data parameters coming with messages). Communication between services is represented using *events* relative to the emission (denoted using !) and reception (denoted using ?) of *messages* corresponding to operation calls. Events may come with a set of data terms whose types respect the operation signatures. In our model, a *label* is either the internal action (tau) or a tuple (SI, M, D, PL) where SI is a service identifier, M is a message name, D stands for the direction (!,?), and PL is either a list of data terms if the message corresponds to an emission, or a list of variables if the message is a reception.

An STS is a tuple (A, S, I, F, T) where: A is an alphabet that corresponds to message events relative to the service provided and required operations, S is a set of states, $I \in S$ is the initial state, $F \in S$ are final states, and $T \in S \times A \times S$ is the transition function. This formal model has been chosen because it is simple, graphical, and it can be easily derived from existing implementation platforms'

languages, see for instance [10, 21, 9] where such abstractions for Web services were used for verification, composition or adaptation purposes. For space reasons, in the rest of the paper, we will describe service interfaces only with their STSs. Signatures will be left implicit, yet they can be inferred from the typing of arguments (made explicit here) in STS labels.

Example. We will use throughout this paper an on-line restaurant booking system as a running example. First of all, let us present the three existing services we reuse to build this new system (Fig. 2). Service YellowPages can receive a search request, and returns an address and a map. Service EasyRestaurant can receive and answer availability requests to check if a restaurant has room for a given date and number of people. After these interactions, this service can receive a booking message and send an acknowledgement back. Service eTaxi receives booking requests with address and date. In addition, we give the system end-user requirements (USER). The user can first look for a place. Then, (s)he can search again, quit, or reserve a restaurant found in the former step. If reservation is possible, the user can accept and book a taxi if necessary. The tau transitions in the user protocol stand for internal decisions taken by her/him.

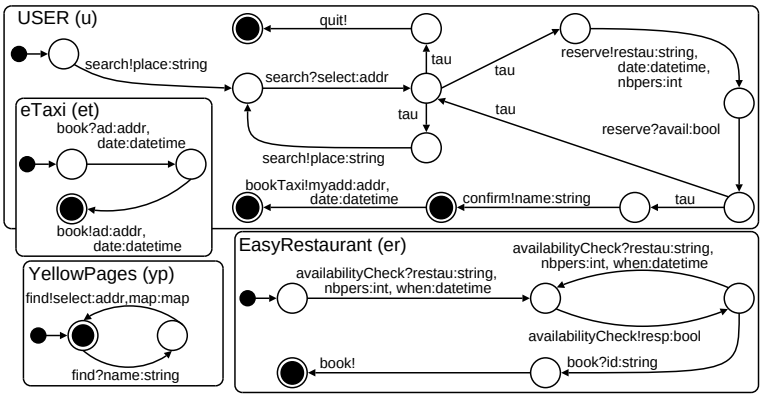

Fig. 2. Example – service protocols

3 Adaptation Contracts

In this section, we present the adaptation contract notation that allows us to specify interactions and to work mismatch situations out. We rely on *synchronization vectors* [1] (or vectors for short). They express correspondences between messages, like bindings between ports or connectors in architectural descriptions. Each event appearing in a vector is executed by one service and the overall result corresponds to an interaction between all the involved services. A vector may involve any number of services and does not require interactions occuring on the

same names of events. Furthermore, variables are used in events as placeholders for message parameters. The same variable name appearing in different events (possibly in different vectors) enables one to relate sent and received message parameters. Vectors can be either written by hand or obtained from a graphical description of the architecture built by the designer (Fig. 3).

However, vectors are not sufficient to support more advanced adaptation scenarios such as contextual rules, choice between vectors or, more generally, ordering (*e.g.*, when one message in some service corresponds to several in another service, which requires to apply several vectors in sequence). The ordering in which vectors have to be applied can be specified using different notations such as regular expressions, LTSs, or (Hierarchical) Message Sequence Charts. Due to their readability and user-friendliness, we chose to specify adaptation contracts using *vector LTSs*, that is, LTSs whose labels are vectors. In addition, vector LTSs ease the development of adaptation algorithms since they provide an explicit description of the adaptation contract set of states. An adaptation contract for a set of service STSs is a couple (V, L) where V is a set of vectors, and L is a vector LTS built over V. If only message name correspondences are necessary to solve mismatches between services, the vector LTS may leave the vector application order unconstrained using a single state and all vector transitions looping on it. In particular, this pattern may be used on specific parts of the contract for which the designer does not want to impose any ordering.

Example. The very first step in the construction of an adaptation contract is to relate messages, and then building the architecture of the system-to-be. The graphical architecture of our booking system is shown in Figure 3 (left) where for instance the search messages in the user requirements correspond to the find ones in the YellowPages service. A specific notation is used to denote an unsynchronized message, *i.e.*, a message with no correspondence (see quit in the user requirements for example).

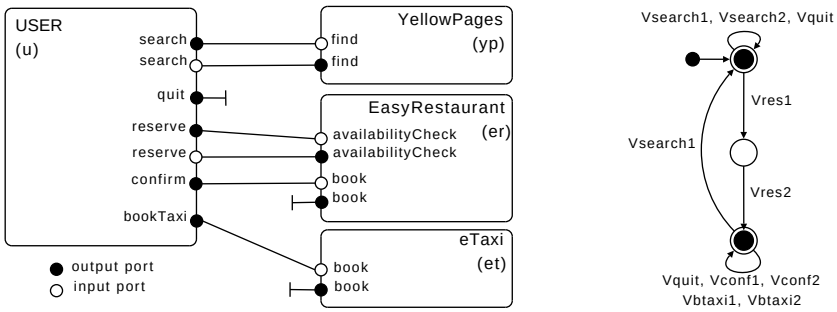

Fig. 3. Example – (left) system architecture, (right) vector LTS

However, such an architecture is not sufficient because we are considering value passing too, and data exchanged through messages (Fig. 2) have to be

matched as well. We give below the vectors that are first deduced automatically from the architecture and complemented with data mappings. As an example the search request emitted by the user comes with a parameter (place) whose counterpart is the argument coming with the reception of find in the YellowPages service (vector Vsearch1). Next, in vector Vsearch2, we can see that answer sent by the YellowPages service comes with two parameters, one of which is matched (select) but the other one (map) is received by the adaptor yet never used afterwards in any other vector. An example of data reordering exists in vector Vres1. Note that the variable scope is not limited to one vector, and a data received in a vector can be used (sent) in another. We have implemented analysis techniques to check possible scope inconsistencies (see Section 4.3).

$\mathsf{Vsearch1} = \langle \mathsf{u} : \mathsf{search!place}; \mathsf{yp} : \mathsf{find?place} \rangle$
$\mathsf{Vsearch2} = \langle \mathsf{u} : \mathsf{search?select}; \mathsf{yp} : \mathsf{find!select}, \mathsf{map} \rangle$
$\mathsf{Vquit} = \langle \mathsf{u} : \mathsf{quit!} \rangle$
$\mathsf{Vres1} = \langle \mathsf{u} : \mathsf{reserve!restau}, \mathsf{date}, \mathsf{nbpers}; \mathsf{er} : \mathsf{availabilityCheck?restau}, \mathsf{nbpers}, \mathsf{date} \rangle$
$\mathsf{Vres2} = \langle \mathsf{u} : \mathsf{reserve?resp}; \mathsf{er} : \mathsf{availabilityCheck!resp} \rangle$
$\mathsf{Vconf1} = \langle \mathsf{u} : \mathsf{confirm!name}; \mathsf{er} : \mathsf{book?name} \rangle$
$\mathsf{Vconf2} = \langle \mathsf{er} : \mathsf{book!} \rangle$
$\mathsf{Vbtaxi1} = \langle \mathsf{u} : \mathsf{bookTaxi!address}, \mathsf{date}; \mathsf{et} : \mathsf{book?address}, \mathsf{date} \rangle$
$\mathsf{Vbtaxi2} = \langle \mathsf{et} : \mathsf{book!address}, \mathsf{date} \rangle$

Being given this set of interactions, the user would be able to submit infinitely availability requests to EasyRestaurant for the same restaurant, which is useless. Accordingly, a vector LTS is defined (Fig. 3, right) in order to impose a single interaction between the user and the EasyRestaurant service every time the user is eager to check for place availability at some restaurant.

4 Adaptor Generation and Verification

An adaptor model for a set of services is an STS running in parallel with the service STSs and guiding their execution (all exchanged messages pass through the adaptor) in such a way that mismatches are avoided and the ordering of messages imposed by the adaptation contract is guaranteed. Generating adaptor protocols is a complicated task since the adaptor has to respect the adaptation contract taking into consideration behavioral constraints of services formalised into their interfaces (STSs). In addition, protocols may generate many interleaved interactions that we want to preserve to accept all the possible message execution orders.

In this work, we chose to encode the adaptation constraints (service interfaces and adaptation contract) into the LOTOS process algebra [14]. LOTOS relies on a rich notation that allows to specify complex concurrent systems possibly involving data types. Our goal is first to generate LOTOS code for service interfaces and their interaction constraints as specified in the contract. In a second step, the LOTOS encoding allows the automatic generation of adaptor protocols whose traces represent all possible (correct) interactions between services. To do so, we rely on CADP [13] a toolbox for LOTOS which implements optimised state space exploration techniques. In particular, we employ *on-the-fly* algorithms to

increase, *w.r.t.* existing approaches, the efficiency of the adaptator generation and reduction process by avoiding the generation of the full state space. The LOTOS encoding also enables the adaptor protocol verification by using model checking tools available in CADP. Techniques and tools presented in this section have been validated on more than 200 examples.

4.1 Principles of the Encoding into LOTOS

This approach aims at successively encoding: the services' STSs, the abstract requirements for composition and adaptation (*i.e.*, the adaptation contract), and the desired system architecture that formalises how the services interact guided by the contract.

Service STS encoding. Each service STS *sv* is encoded using several LOTOS processes. Each LOTOS process corresponds to one state *s* of the STS, and its behavior is a choice containing as many branches as there are transitions outgoing from *s*. Each branch encodes the label associated to the transition, and is followed by a call to the LOTOS process that encodes the target state of the transition being translated. An additional branch, using a specific `FINAL` action, models termination when *s* is final. STS labels are encoded into LOTOS following patterns presented in Figure 4. Sent (resp. received) messages are represented with a "`_EM`" (resp. "`_REC`") suffix. In addition, LOTOS symbols `!` and `?` are used to support data transfer (resp. emission and reception). In our context, the correct distribution will be ensured by the encoding of the adaptation constraints (see the next step in this section), therefore all service STS labels that involve value passing (emission or reception of parameters) are translated into LOTOS with a question mark followed by as many fresh variables as there are parameters coming with the message. Since these variables are placeholders, their LOTOS type can simply be an arbitrary one that we call `PH`. This type is defined beforehand using the LOTOS abstract datatype facilities with all the placeholder names appearing in the contract defined as type constructors.

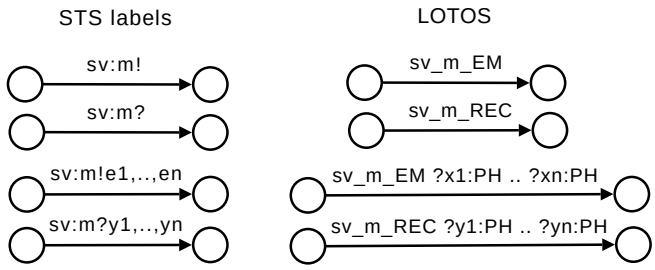

Fig. 4. Encoding patterns for STS labels

Adaptation contract encoding. An adaptation contract is encoded by generating (i) a process for the vector LTS, (ii) a process for each vector defined in

the contract, and (iii) the interleaving of all these vector processes. The correct ordering of vectors is ensured by the vector LTS process thanks to two actions for each vector v. A first one (`run_`v) activates the corresponding vector process. A second one (`rel_`v) releases the vector LTS and enables it to overlap vector applications. The vector LTS process (i) is encoded using the same pattern as service STSs, that is every state is encoded as a LOTOS process. For each transition with label v in the vector LTS, two LOTOS actions are generated in a sequence: `run_`v; `rel_`v.

Vector processes (ii) are first launched through a "`run_`" interaction with the vector LTS (Fig. 5, 1). Next, they communicate with services on all actions appearing in their vector definition. They have to receive all sent messages (Fig. 5, 2) before beginning to emit some (Fig. 5, 4b.2). There is no specific ordering between receptions (*resp.* between emissions) in a vector process. When a vector process executes a vector, it must be ready to interact with the service STSs on their emissions (Ox in Fig. 5). Then, several strategies are possible to release the vector (`rel_`v), and therefore to execute the services' receptions. A first option is to wait the complete processing of a vector before firing a new one (4a.2 done after 4b.1 and 4b.2). Another strategy is to execute the release action once all the emissions executed, that means that the execution of the receptions by the services (Iy in Fig. 5) can be postponed, and the vector LTS can launch another vector. This behavior makes the reordering of messages possible, a typical case of mismatch between services.

As regards value passing, an auxiliary LOTOS process `Store` is generated to store information about the availability of received values. Every time some values are sent by a service, they are received by one of the vector processes and stored by using the (global) process `Store`, which makes them available at the level of the adaptor. This availability is essential, because when service receptions in a vector are being run (emissions at the level of the adaptor), this firing is conditioned by the availability of the values to be emitted. Thus, every service emission in a vector is followed by an interaction with the process `Store` to set to *true* the availability of the received values (Fig. 5, 3), and every service reception in a vector is preceded by some interactions with the `Store` process to check that the required values have been received (Fig. 5, 4b.1). In the latter case, the vector process may have to wait the availability of the needed resources. Such an active waiting is encoded using a looping process that terminates once the data are available. If they are never available, this will generate a deadlock in the underlying state space that will be cut away in a second step by our reduction techniques (see Section 4.2).

Finally, vector processes (iii) are interleaved since they do not communicate together. All the vector processes may synchronize with the `Store` process to store new available values, or check the availability of some values to be sent. The collaboration diagram in Figure 5 summarizes the pattern for encoding vectors into LOTOS when vector overlapping is enabled.

System encoding. In this step, we generate a LOTOS expression corresponding to the whole system constraints from the LOTOS processes encoding the service

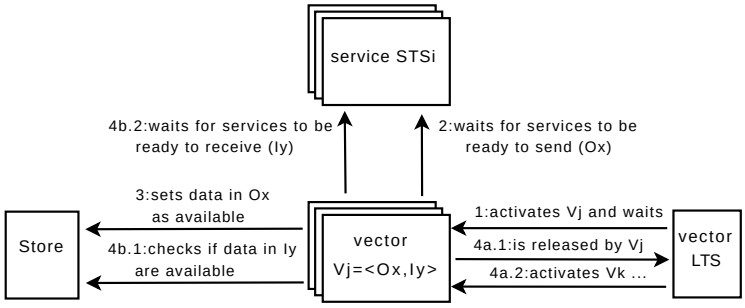

Fig. 5. Encoding pattern for vectors

STSs, the ones encoding the adaptation contract, and respecting the desired system architecture (adaptor in-the-middle, intercepting all messages). This means that the service STSs only interact together on `FINAL` (correct termination is when all services terminate) while they interact with vectors on actions used in their alphabets. The synchronizing between vector processes and vector LTS has been described earlier on (using "`run_`" and "`rel_`" actions). In addition, all actions that are not messages of the system, *i.e.*, messages appearing in the involved services, are hidden as they represent internal actions of the adaptor we are building (*e.g.*, "`run_`" and "`rel_`" actions, or all interactions with the `Store` process). They are the "mechanics" of adaptation and are not relevant for implementation. They will be removed by reduction steps of the adaptor generation process (see Section 4.2).

Tool support: Compositor. The LOTOS encoding is fully automated by Compositor, a tool we have implemented. Supported inputs are XML STSs and the `aut` LTS textual format extended with value passing for service interfaces, and XML for contract specifications. Strategies to implement the different ways of releasing vectors have been implemented as an option.

4.2 On-the-Fly Adaptor Generation

An adaptor can be obtained from the state space of the whole system (services and adaptation contract) by keeping only the correct behaviors, which amounts to cut the execution sequences leading to deadlock states. In the adaptation techniques that support deadlock elimination [6, 2], the computation of the deadlock-free behaviors is done by performing a backward exploration of the explicit, *entirely constructed*, state space by starting at the deadlock states and cutting all the transitions whose target state leads to a deadlock. To increase efficiency, we avoid the entire construction of the state space and instead we explore it forwards in order to generate the adaptor *on-the-fly* by carrying out deadlock elimination and behavioral reduction simultaneously.

Deadlock elimination. First, the execution sequences leading to deadlocks must be pruned. We do this by keeping, for each state encountered, only its

successor states that potentially reach a successful termination state, which is source of a transition labeled with the action `FINAL`. Besides avoiding deadlocks (sink states reached by actions other than `FINAL`), this also avoids livelocks, *i.e.*, portions of the state space where some services get "trapped" and cannot reach their final states anymore. The desired successor states satisfy the PDL [8] formula $\langle true^*.\texttt{FINAL}\rangle\, true$, which can be checked on-the-fly using the Evaluator [18] model checker. However, this scheme is not efficient since each invocation of Evaluator has a linear complexity *w.r.t.* the size of the state space and therefore a sequence of invocations (in the worst case, one for each state) may have a quadratic complexity. An efficient solution is to translate the evaluation of the formula into the resolution of the boolean equation system (BES) $\{X_s =_\mu \bigvee_{s \xrightarrow{\texttt{FINAL}} s'} true \vee \bigvee_{s \rightarrow s''} X_{s''}\}$, where a boolean variable X_s is true iff state s satisfies the propositional variable X corresponding to the PDL formula. A state s potentially leading to a successful termination is detected by solving on-the-fly the variable X_s of this BES using the algorithm A3 of the Caesar_Solve library [16]. In this way, a sequence of resolutions performed during a forward exploration of the state space has a linear-time overall complexity and does not store transitions, but only states in memory.

Behavioral reduction. Second, the adaptor STS obtained by pruning can be reduced on-the-fly modulo an appropriate equivalence relation in order to get rid of the internal actions and obtain an adaptor as small as possible. These internal actions correspond here to the encoding of the system adaptation constraints, *e.g.*, "`run_`" and "`rel_`" actions. Such internal actions are not relevant for adaptor implementation but are usually inherent to adaptation processes, as they model internal computations done by adaptors [6]. The algorithms presented in [15] can be used to implement on-the-fly reductions modulo tau-confluence (a form of partial order reduction preserving branching bisimulation) and the tau$^*.a$ and weak trace equivalences, both of which eliminate internal transitions and (for weak trace) determinize adaptor STSs.

Tool support: Scrutator and CADP. The on-the-fly adaptor generation is implemented by the Scrutator tool that we have developed using the Open/Caesar [11] environment for graph manipulation provided by the CADP verification toolbox. Two kinds of pruning are implemented by the tool: the first one deletes the states leading eventually to deadlocks and the second one keeps only the states leading (potentially or eventually) to transitions labeled by a given action (here, `FINAL`). Besides the on-the-fly reductions currently offered by Scrutator (tau-confluence, tau$^*.a$, and weak trace equivalence), we plan to implement reductions modulo other equivalences, such as branching bisimulation; for the time being, the adaptors generated by Scrutator can be reduced off-line modulo strong or branching bisimulation using the Bcg_Min tool of CADP.

To automate the whole adaptation process, Compositor generates an SVL script [12] in charge of the following activities: building and reducing the adaptor on-the-fly by invoking Scrutator on the LOTOS specification of the system; "mirroring" of the adaptor actions (reversing emissions and receptions, "`_EM`" and "`_REC`") as the adaptor acts as an orchestrator in-the-middle of the ser-

vices; and pretty-printing of the adaptor STS by translating its actions from a LOTOS-like to a more user-friendly syntax.

The reduced adaptor protocol for our running example is shown in Figure 6. The initial state is identified by `0`. In this state, the adaptor can interact with the user (message `SEARCH`) and receives as parameter the place (s)he is looking for. Next, the adaptor sends this place to the YellowPages service with the `FIND` message, etc.

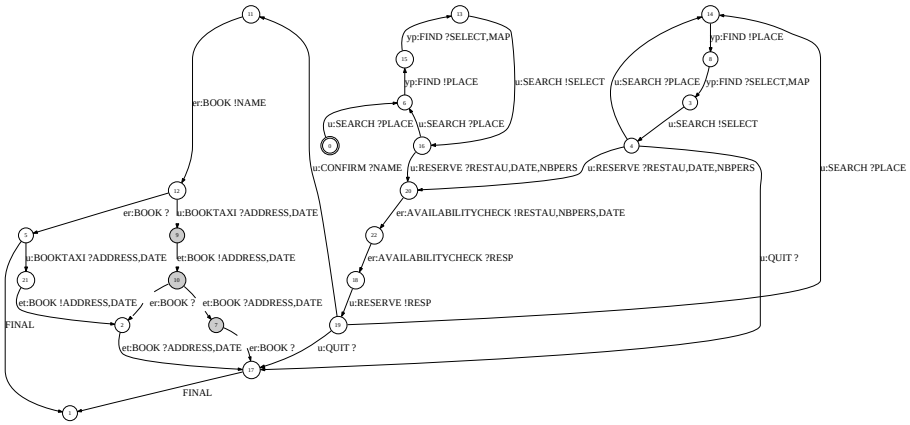

Fig. 6. Example – adaptor protocol

4.3 Adaptor Verification

In our approach, contracts are built by the designer. Therefore, they can contain errors that will also appear at the level of the adaptor. As a first step in the verification of the adaptor, we have implemented several static analysis checks to verify that the contract is correctly written (labels defined in interfaces correctly used in vectors, vectors and vector LTS structurally consistent, scope and type of placeholders, etc). These static analysis features are very useful for detecting the simple errors that one can make while writing out a contract manually. Nonetheless, this is not enough since protocols of interfaces and contracts (vector LTS) are not considered. Therefore, to complement static analysis checks, we propose more powerful verification techniques based on model checking tools (Evaluator). Two kinds of temporal properties are suitable for checking the behavior of adaptors: (i) *general properties* (placeholder occurrence, service action preserving, etc) related to the adaptor structure, which should be satisfied by any adaptor generated using our approach, (ii) *specific properties* (safety and liveness) related to the adaptor protocol, which differ from one adaptor to another.

5 Adaptor Implementation

In this section we present the final step of our approach, namely adaptor implementation. Due to lack of space, the initial step, generating STS models from (A)BPEL (using the rules defined in [21]) is not presented here. To generate a BPEL orchestrator from an adaptor model we proceed in two steps: (i) filtering the model, and (ii) encoding the filtered model into BPEL.

Adaptor filtering. The adaptor generation algorithm is a implementation independent model-based one whose objective is to be applied to different implementation platforms (BPEL, Windows Workflows, SCA components, etc.). To support implementation using the BPEL constructs, we have to apply first three simplification rules:

- whenever a state has both emission and reception outgoing transitions, we remove the reception transitions;
- whenever a state has more than one emission outgoing transition, we keep a single one;
- let o be a two-way operation, *i.e*, a receive-reply operation of a service to be invoked by the adaptor in a synchronous way; for every transition t with an emission corresponding to such an o, and targeting state s, we remove all transitions outgoing from s but for the transition with the corresponding reception (*i.e.*, we impose atomicity of the two events corresponding to invocations in the adaptor). In a case where such a second transition is not available, we also remove t.

We end by cleaning the adaptor model, *i.e.*, we remove states (and accordingly transitions) which are not reachable (from the initial state) or not coreachable (from a final state). Filtering is demonstrated on Figure 6 where the grey states and related transitions are removed. Filtering is compatible with adaptation; it just removes some of the interactions between the services which are not possible from a BPEL point of view. Verification techniques presented for adaptor models apply to filtered models too. Currently, we have been able to show that the important safety and liveness properties that yield for the Figure 6 adaptor (*e.g.*, that the client cannot be asked to confirm the reservation before the YellowPages service has found an appropriate place, or that the client cannot be asked to confirm the reservation before the YellowPages service has found an appropriate place) yield also after filtering.

BPEL implementation. Once models have been cleaned up as presented above, we automatically implement them in BPEL as follows.
Partnerlinks and variables. A partnerlink is created for each service, plus one for the composite itself (USER). Global variables are created for the vector variables and for each part of received or emitted message. Moreover, a STATE integer variable is used to represent the current state and a FINAL boolean variable to represent the termination of the adaptor.
Communication. A c:msg!x1,...,xn transition (c not being USER) followed by a c:msg?y1,...,yn transition is encoded as a synchronous invoke activity with message msg and partnerlink c. A USER:msg?x1,...,xn transition corresponds to the

interaction with the environment, it is encoded as a receive activity with message msg and partnerlink USER. Finally, a USER:msg!x1,...,xn transition corresponds to an interaction with the environment, it is encoded as a reply activity with message msg and partnerlink USER. All communication activities related to USER are linked using a correlation set named USER_CS with a property USER_PROP. Moreover, each of the operations in the USER interface has an additionnal part with a string identifier and a corresponding property alias making the link with USER_PROP. This machinery is required to ensure the correctness of the adaptor protocol *w.r.t.* its environment, *e.g.*, the user.
Assignments. Some adaptor variables (xi and yj above, *e.g.*, name in our example) come from vectors, while message parts in the communication activities (invoke, receive, reply) correspond to variables in service protocols (*e.g.*, id for message book in service EasyRestaurant in our example). To link them, before each invoke or reply activity, we add an assign activity assigning adaptor variables to message parts; accordingly, after each invoke or receive activity we add an assign activity assigning message parts to adaptor variables.
Process. We rely on the state machine pattern. Initially the STATE variable is set to the target state of the first transition in the adaptor. The main body of the process then corresponds to a while (not FINAL) activity. Cascaded if statements are used inside it to encode the adaptor states. The if body of a state i encodes its outgoing transition(s). For a single one we use communication encodings presented above. When there are several possible receptions we use a pick activity with an onMessage branch for each. If there is also a termination transition, we add an onEvent branch in the pick with a timer. In all cases, we terminate by updating the STATE variable accordingly to the transition(s) taken into account. For the final state we only set FINAL to true.

Tool support: BPEL2STS and STS2BPEL. The obtaining of STS from BPEL, the filtering of adaptor models, and the generation of BPEL adaptors from STS models, presented above, are automated by BPEL2STS and STS2BPEL, tools we have implemented. A part of our adaptor in BPEL is presented in Figure 7. Service deployment has been achieved using the NetBeans 6.0.1 IDE with the GlassFish BPEL Engine.

6 Related Work

Several adaptation proposals [4, 6, 2] focus on solving behavioral mismatch between abstract descriptions of components. Brogi et al. (BBC) [4] present a methodology for generative behavioral adaptation where component behaviors are specified with a subset of the π-calculus and composition specifications with name correspondences. An adaptor generation algorithm is used to refine the given specification into a concrete adaptor which is able to accommodate both message name and protocol mismatch. This approach has recently been used to obtain adaptor implementations for services [5] (see below). Autili et al. (IT) [2] address the enforcement of behavioral properties out of a set of components. Starting from the specification with MSCs of the components to be assembled

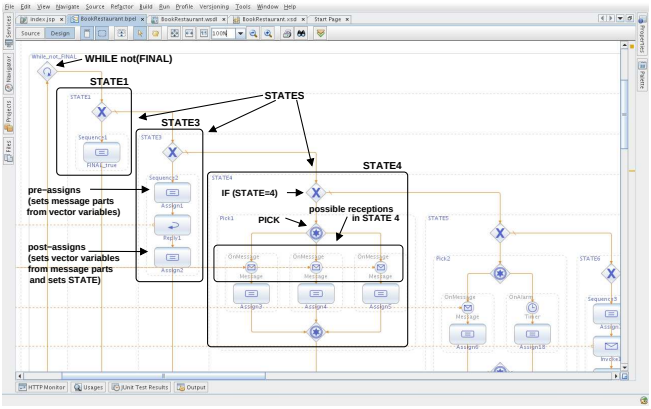

Fig. 7. BPEL Adaptor (part of) in the NetBeans IDE 6.0.1

and of LTL properties (liveness or safety) that the resulting system should verify, they automatically derive the adaptor glue code for the set of components in order to obtain a property-satisfying system. They follow a restrictive adaptation approach, hence they are not able for example to reorder messages when required. More recently, in [6], we have proposed an automated adaptation approach that is generative and supports adaptation policies and system properties described by means of regular expressions of vectors. It superseded both IT (as it supported message reordering) and BBC (which could generate dumb adaptors [4] and has no tool-support), yet it built on algorithms based on synchronous products and Petri nets encodings with a resulting exponential complexity for the computation of adaptors. Here, this is avoided thanks to process algebra encodings and on-the-fly generation techniques.

In their paper *Adapt or Perish* [7], Dumas and collaborators presented an approach to behavioral interface adaptation based on the definition of a set of adaptation operations for establishing the basic relation patterns between the messages names used in the components being adapted, and they defined a trace-based algebra for describing the transformations required to solve the adaptation problem. They also present a visual notation for describing a mapping between the behavioral interfaces of the components. However, their proposal does not present a solution for deriving an adaptor from the visual mappings, but just contains a preliminary (*i.e.*, non sufficient) condition for detecting deadlock scenarios in the behavioral interfaces.

Some recent approaches found in the literature [5, 20, 19] focus on existing programming languages and platforms, such as BPEL or SCA components, and suggest manual or at most semi-automated techniques for solving behavioral mismatch. In the context of Web services and BPEL, [5] outlines a methodology for the generation of adaptors capable of solving behavioral mismatches between BPEL processes. In their adaptation methodology, the authors use an intermediate workflow language for describing component behavioral interfaces, and they

use lock analysis techniques to detect behavioral mismatch. Similarly, [20] provides automated support for the identification of protocol-level mismatches, but is able to generate an adaptor only in the absence of deadlock. If deadlock may arise from the combination of the components, the authors propose a way to handle the situation by generating a tree for all mismatches that result in a deadlock, and suggesting some hints for assisting the designer in the manual implementation of the actual adaptor. In [19], the authors deal with the monitoring and adaptation of BPEL services at run-time according to Quality of Services attributes (different focus than us). Their approach also proposes replacement of partner services based on various strategies either syntactic or semantic.

Finally, compared to a preliminary version of this work [17], in the current paper, we have first extended the model of services with value passing. Consequently, the contract notation was enhanced as well to consider not only message matching but also correspondences between message arguments. New adaptation and verification techniques have been proposed to deal with these new models, and tool support extended in consequence. Last but not least, we have addressed adaptor implementation in BPEL.

7 Concluding Remarks

Software adaptation is a promising solution to compose in a non-intrusive way black-box services that contain incompatibilities in their interfaces. In this paper, we have presented our tool-supported techniques to generate adaptor protocols from interfaces of services described by signatures and protocols with value-passing, and an adaptation contract. Adaptor generation is completely automated and the resulting adaptor makes the whole system work correctly by solving protocol mismatches as well as value passing issues. Since our mechanisms are based on an encoding into the LOTOS process algebra, we take advantage of the existing CADP toolbox for LOTOS to verify the correctness of the contract. We have also shown with BPEL how our adaptors can be implemented. The main perspective of this work is to propose an assisted design approach to help and guide the architect in the construction of adaptation contracts.

Acknowledgements. This work has been partially supported by project "PERvasive Service cOmposition" (PERSO) funded by the French National Agency for Research (ANR-07-JCJC-0155-01), project TIN2008-05932 funded by the Spanish Ministry of Innovation and Science, and project P06-TIC2250 funded by the Andalusian local Government.

References

1. A. Arnold. *Finite Transition Systems.* International Series in Computer Science. Prentice-Hall, 1994.
2. 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.

3. S. Becker, A. Brogi, I. Gorton, S. Overhage, A. Romanovsky, and M. Tivoli. Towards an Engineering Approach to Component Adaptation. In *Architecting Systems with Trustworthy Components*, volume 3938 of *LNCS*, pages 193–215. Springer-Verlag, 2006.
4. A. Bracciali, A. Brogi, and C. Canal. A Formal Approach to Component Adaptation. *Journal of Systems and Software*, 74(1):45–54, 2005.
5. A. Brogi and R. Popescu. Automated Generation of BPEL Adapters. In *Proc. of ICSOC'06*, volume 4294 of *LNCS*, pages 27–39. Springer-Verlag, 2006.
6. C. Canal, P. Poizat, and G. Salaün. Model-Based Adaptation of Behavioural Mismatching Components. *IEEE Transactions on Software Engineering*, 34(4):546–563, 2008.
7. M. Dumas, K. W. S. Wang, and M. L. Spork. Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation. In *Proc. of BPM'06*, volume 4102 of *LNCS*, pages 65–80. Springer-Verlag, 2006.
8. M. J. Fischer and R. E. Ladner. Propositional Dynamic Logic of Regular Programs. *Journal of Computer and System Sciences*, 18(2):194–211, 1979.
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. H. Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In *Proc. of TACAS'98*, volume 1384 of *LNCS*, pages 68–84. Springer-Verlag, 1998.
12. H. Garavel and F. Lang. SVL: a Scripting Language for Compositional Verification. In *Proc. of FORTE'01*, pages 377–392. IFIP, Kluwer Academic, 2001.
13. H. Garavel, R. Mateescu, F. Lang, and W. Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In *Proc. of CAV'07*, volume 4590 of *LNCS*, pages 158–162. Springer-Verlag, 2007.
14. ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, ISO, 1989.
15. R. Mateescu. On-the-fly State Space Reductions for Weak Equivalences. In *Proc. of FMICS'05*, pages 80–89. ACM Computer Society Press, 2005.
16. R. Mateescu. CAESAR_SOLVE: A Generic Library for On-the-Fly Resolution of Alternation-Free Boolean Equation Systems. *STTT Journal*, 8(1):37–56, 2006.
17. R. Mateescu, P. Poizat, and G. Salaün. Behavioral Adaptation of Component Compositions based on Process Algebra Encodings. In *Proc. of ASE'07*, pages 385–388. IEEE Computer Society, 2007.
18. R. Mateescu and M. Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. *Science of Computer Programming*, 46(3):255–281, 2003.
19. O. Moser, F. Rosenberg, and S. Dustdar. Non-Intrusive Monitoring and Adaptation for WS-BPEL. In *Proc. of WWW'08*, pages 815–824, 2008.
20. H. R. Motahari-Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-Automated Adaptation of Service Interactions. In *Proc. of WWW'07*, pages 993–1002, 2007.
21. 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.