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

Radu Mateescu, Pascal Poizat, *Member, IEEE Computer Society,* and Gwen Salaün

Abstract—Reuse and composition are increasingly advocated and put into practice in modern software engineering. However, the software entities that are to be reused to build an application, *e.g.*, services, have seldom been developed to integrate and to cope with the application requirements. As a consequence, they present mismatch, which directly hampers on their reusability and the possibility to compose them. Software Adaptation has become a hot topic as a non-intrusive solution to work mismatch out using corrective pieces named adaptors. However, adaptation is a complex issue, especially when behavioral interfaces, or conversations, are taken into account. In this article, we present state-of-the-art techniques to generate adaptors given the description of reused entities' conversations and an abstract specification of the way mismatch can be solved. We use a process algebra to encode the adaptation problem, and propose on-the-fly exploration and reduction techniques to compute adaptor protocols. Our approach follows the model-driven engineering paradigm, applied to service-oriented computing as a representative field of composition-based software engineering. We take service description languages as inputs of the adaptation process and we implement adaptors as centralized service compositions, *i.e.*, orchestrations. Our approach is completely tool-supported.

Index Terms—Service composition, software adaptation, interfaces, protocols, mismatch, adaptation contracts, process algebra, on-the-fly generation, verification, tools.

1 Introduction

The reuse and composition of existing software pieces has always been a central issue in software engineering in order to reduce development time and cost. This trend has increased with the development of service-oriented architectures, that give the technical means for the composition of heterogeneous applications encapsulated as services. However, direct reuse and composition of existing services is in most cases impossible because their interfaces present incompatibilities or mismatches. Let us imagine a very simple application in which we have a service, C, acting as a client for another service, S. Let us also suppose these services have conversations (also known as protocols or behavioral interfaces) that describe the way messages are exchanged, *i.e.*, the order in which S operations are invoked by C and the order in which S requires one to invoke its operations. When services are developed by different third-parties or in different contexts, which is often the case, mismatches may appear:

- Name mismatch (or 1-1 mismatch): an operation provided by S and an operation required by C have the same semantics but have different names, *e.g.*, two one-way operations, sendToPrinter and print.
- Unanticipated reception (or 1-0 mismatch): C tries to send a message to S while it is not used or required by it, *e.g.*, C, in non-connected mode, tries to send a login/password before each request, while S, in connected-mode, requires it only once.
- Generalized mismatch (or n-m mismatch): C and S use a different number of operations/messages for some task, *e.g.*, C wants to add two numbers x and y invoking in turn operations setX (to set x), setY (to set y) and add (to compute the sum) while S has a single operation, addition (to provide operands and compute the sum at once). In this generalized mismatch, data may have to be aggregated (n-1) or split (1-n) amongst messages.
- Reordering: C and S have corresponding operations but different orderings, *e.g.*, C first sets up a file to operate on (setF) and then asks for an operation to be performed on it (perform), while S has first an operation for setting up the action (setA) and then an operation for doing the previously set action on a given file (run).

Let us illustrate these different kinds of mismatch on a concrete example, Figure 1, left, where we extend the reordering example above with new elements that will demonstrate how an adaptor works. C sends session identifiers (cid) with its messages to enable message follow-up. S also requires session identifiers to operate (sid), but these are generated by S and are different from the ones sent by C. As one can see from the figure, there is no chance that S can be used to fulfill the needs of C.

Software Adaptation [1], [2], [3] is a very promising solution for non-intrusive composition of black-box components or (Web) services whose functionality is as required for the new system, but that present interface

- *R. Mateescu is with the CONVECS project-team, INRIA, France.*
- *P. Poizat is with the University of Evry Val d'Essonne and LRI, France.*
- *G. Salaün is with Grenoble INP and INRIA, France.*

(c) 2012 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other users, including reprinting/ republishing this material for advertising or promotional purposes, creating new collective works for resale or redistribution to servers or lists, or reuse of any copyrighted components of this work in other works.

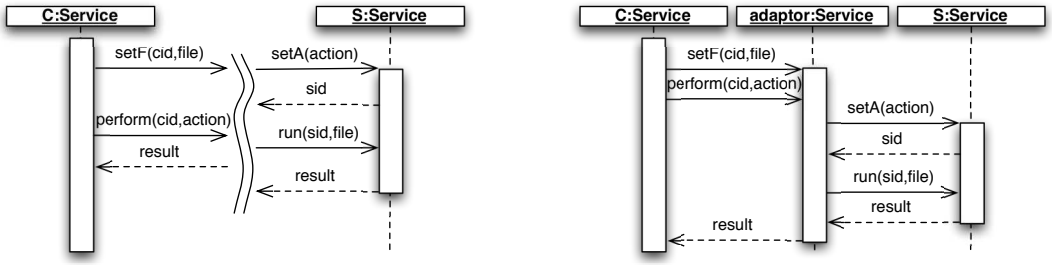

Fig. 1: Mismatch between services (left) and adaptor (right)

mismatches. Adaptation techniques aim to automatically generate 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. For instance, in the case of the name mismatch presented above, the adaptor would receive sendToPrinter from C and then would send print to S with the same data. If an unanticipated reception occurs, the solution is that the adaptor receives all client connection messages but only transmits the first one. In case of a generalized mismatch or a reordering issue, the adaptor would first receive from C all the necessary data and messages, and would interact with S only once all have been received. This is demonstrated on our example in Figure 1, right. In the sequel of this article we will see how such adaptors can be automatically generated.

Adaptation and service composition. The techniques we propose in this article can be used in different representative service composition contexts. A first context (Fig. 2, left) is when one wants a service composition to be automatically built for him/her. Given a set of services to be composed (Fig. 2, ❶), the description of the needs (Fig. 2, ❷), and an adaptation contract (Fig. 2, ❸), we can generate automatically an adaptor which is implemented as an orchestrator (Fig. 2, ❹). Since the used services and the user needs may present mismatch, adaptation supports the service composition process, yielding *adaptive composition*. The main use of it is either at design time (*e.g.*, when a service architect has to develop a new orchestration) or at deployment time (*e.g.*, with automatic end-user composition features within a cloud). A second context (Fig. 2, right) is when a service orchestration is already there (Fig. 2, ❺), but some services (here one, Fig. 2, ❻) fail or become unreachable. Using *corrective adaptation*, we may generate an adaptor between the broken partner links and replacement services (n-m replacement being possible, Fig. 2, ❼). Again, this adaptor is implemented as an orchestrator (Fig. 2, ❽) which mediates between the broken partner links and the replacement services. In the sequel, we will present the application of our technique to adaptive composition since corrective adaptation can be seen as a simpler case of it.

We support both simple and complex services, and services may be composite or not. *Simple services* have a signature interface (WSDL) and an implementation (*e.g.*, in Java). Additionally, to have a unified approach, we can associate to these services a basic conversation (Abstract BPEL, ABPEL for short) that represents the fact that the service operations may be called in any order. *Complex services* directly present such a conversation. We suppose it is also given in ABPEL. Finally, we have *composite services*, here implemented as orchestrations. These may be simple or complex. Orchestration implementation is usually done in (executable) BPEL. It can also be achieved using other languages such as Java, in which case we assume a description of the orchestration conversation is available.

Contributions. Model-based behavioral adaptation approaches are either restrictive or generative. *Restrictive approaches*, see for instance [4], [5], try to solve the problem by cutting off (pruning) the behaviors that may lead to mismatches, thus restricting the functionality of the services involved. *Generative approaches*, see for instance [6], [7], [8], 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 article, 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.

Automatic service composition approaches, see [9], [10] for comprehensive surveys, generally take the as-

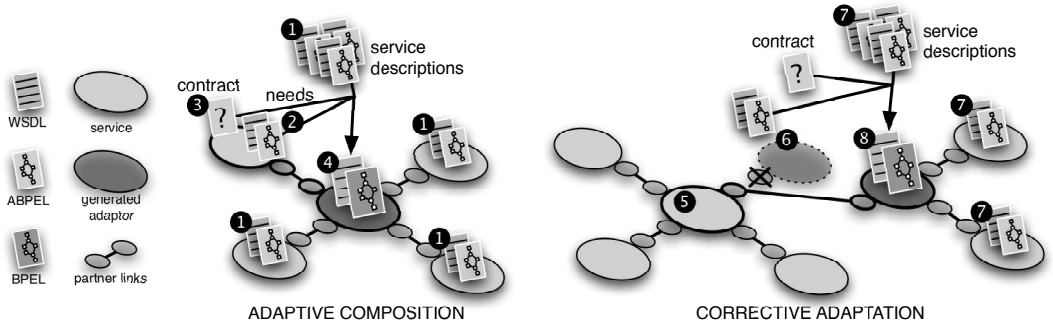

Fig. 2: Using adaptation to empower service composition

sumption that services, that are discovered and composed into an orchestration to fulfill some given requirements, perfectly match these requirements. In our approach we do not need to take such a strong assumption, and mismatching service/requirement conversations can be solved out using adaptation, hence yielding possible compositions where non-adaptive composition techniques would fail.

Let us emphasize that, following model-driven engineering principles, our adaptation techniques are platform-independent but they can easily be related to implementation languages as we will illustrate with (A)BPEL. Last but not least, our approach is fully tool supported.

Approach. In this article, we first present a model of services that makes it possible to describe signatures (operation names and types) and behaviors (interaction protocols or conversations). Protocols are essential [11] because erroneous executions or deadlock situations may occur if the designer does not take them into account while building composite services. In addition to considering messages exchanged in protocols, it is important to include value passing (parameters) with messages since this feature may raise composition issues too (unmatched number of parameters, different orderings, 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 and between message parameters. Then, from a set of service protocols and a contract, we present our approach to generate adaptor protocols which relies on an encoding into a process algebra together with on-the-fly exploration and reduction techniques. Verification of contracts is also possible by using CADP [12], a rich verification toolbox. Last but not least, we show how adaptors can be implemented in the BPEL service orchestration language. Our proposal is supported by tools (Fig. 3) that automate the generation of the encoding (Compositor), the efficient computation of the adaptor protocol from this encoding (Scrutator), and the model-based verification of the adapted system (here we reuse Evaluator and Bisimulator from the CADP toolbox). Relationship with BPEL is achieved with two tools that respectively extract abstract service models from XML descriptions of services (BPEL2STS) and support the generation of BPEL adaptors from adaptor models (STS2BPEL). The adaptation contract construction can be assisted and partially automated using recent results presented in [13], [14] and implemented in the Acide tool. BPEL2STS, Compositor, Scrutator, and STS2BPEL have been implemented for this work. The other tools we mention in this article (*e.g.*, Acide or CADP) have only been reused.

Outline. The remainder of this article is structured as follows. Section 2 presents our model of services and Section 3 introduces the contract notation which is used for adaptation purposes. Section 4 presents our encoding, and Section 5 our adaptor generation and verification techniques. Section 6 focuses on the relationships of our techniques with service frameworks and languages; we use (Abstract) BPEL to specify behavioral interfaces and generate the BPEL implementation of an adaptor from its protocol. Section 7 compares our approach to related work, and Section 8 ends the article with some concluding remarks.

2 Model of Services

In order to address service adaptation and composition, we are interested in service interfaces rather than in service implementations. This section focuses on the definition of the formal service model. Section 6 treats in detail the relationship between this model and service interface description languages. It explains how a formal model is derived from a service description, and how an orchestration implementation is generated from a formal model of an adaptor.

(Web) Services expose the operations they provide, using *signatures*. Moreover, complex services, including stateful ones, feature a *behavioral description* of the way operations should be called. Additional information, such as semantics or quality of service, can be used for service discovery and composition. We focus here on signatures and behaviors which are widely accepted and used as interface description language. Although our model is platform-independent, we will make some

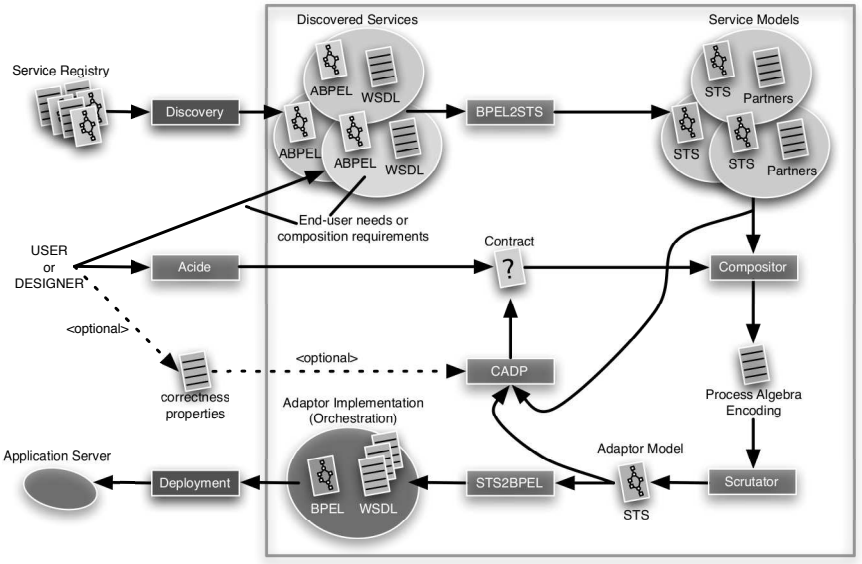

Fig. 3: Overview of our approach

explicit connections between models and implementation languages (WSDL and (A)BPEL in this article) to explain the different elements of our model.

Signatures. A signature is a combination of data structures (*e.g.,* defined using XML Schema), operations, and messages (*e.g.,* defined using WSDL). We model it as a tuple $\Sigma = (\mathcal{D}, \mathcal{O}, in, out)$. $\mathcal{D}$ is a set of domains that correspond to (XML) types, which can be either simple or complex. A type is associated with a name-space. For clarity, name-spaces will be omitted in this article when they are clear from the context. $\mathcal{O}$ is a set of (provided) operations that are either one-way or two-way. $in, out : \mathcal{O} \rightarrow \mathcal{D} \cup \{\perp\}$ denote respectively the input and output messages of an operation ($\perp$ when undefined, *e.g.,* $out(o) = \perp$ for any one-way operation o). In case of a complex service, several signatures are to be taken into account (one for each partner link). To support both simple and complex services in a unified way, we introduce partnerships. A partnership, ρ, is a set of signatures indexed by a set, PN, of partner names: $\rho = \{\Sigma_{i,i \in PN}\}$. For a simple service, ρ is a singleton that corresponds to the service signature.

Events. The semantics of a service conversation depends on message-based communication, which is modeled using *events* defined over the service partnership, ρ. An input event, $o?x$, with $o \in \mathcal{O}$ and x a variable whose domain is $in(o)$, corresponds to the reception of the input message of operation o. Similarly, we define output events $o!x$, the domain of x being $out(o)$. When messages have several parts, events are modified accordingly. Suppose an operation add with an input message having two parts, x and y, both of domain xsd:int. We may then have an event $add?x, y$ to denote the reception of a message to call operation add. $Ev^?$ (resp. $Ev^!$) is the set of input events (resp. output events). Additionally, the τ event is used to denote non-observable internal computations or conditions. Finally, we define events as $Ev = Ev^? \cup Ev^! \cup \{\tau\}$.

Conversations. Conversations, protocols, or service behavioral interfaces define the way one interacts with a service. Several models have been proposed to support behavioral service discovery, verification, testing, composition or adaptation (see [15], [16] for a survey of these models). They mainly differ in their formal grounding (Petri nets, transition systems, or process algebras), and the subset of service languages being supported. Since we are interested, for adaptation, only in abstract behavioral descriptions of service(s) interfaces (ABPEL) rather than full-fledged executable orchestration definitions (BPEL), we can use a state-transition model, with transitions labeled by events. Such labeled transition models are simple and graphical. Additionally, these models can be derived from existing service interface or implementation languages [17], [18], [19], [20], [21]. Since events carry symbolic information (*e.g.,* variables in input events), our model is a Symbolic Transition System (STS).

Definition 1 (STS): A Symbolic Transition System or STS is a tuple (A, S, I, F, T) where: A is an alphabet which corresponds to the set of labels associated with transitions, S is a set of states, $I \in S$ is the initial state, $F \subseteq S$ are final states, and $T \subseteq S \times A \times S$ is the transition relation.

Such models can be found with different names in the literature [22], [23], [24]. They have been originally introduced as Symbolic Transition Graphs (STG) in [22] to give a symbolic semantics to value-passing processes. In an STS, data being exchanged appear as variables (parameters) of events, *e.g.,* $add?x, y$ for the reception of two data values with message add. Hence, an STS

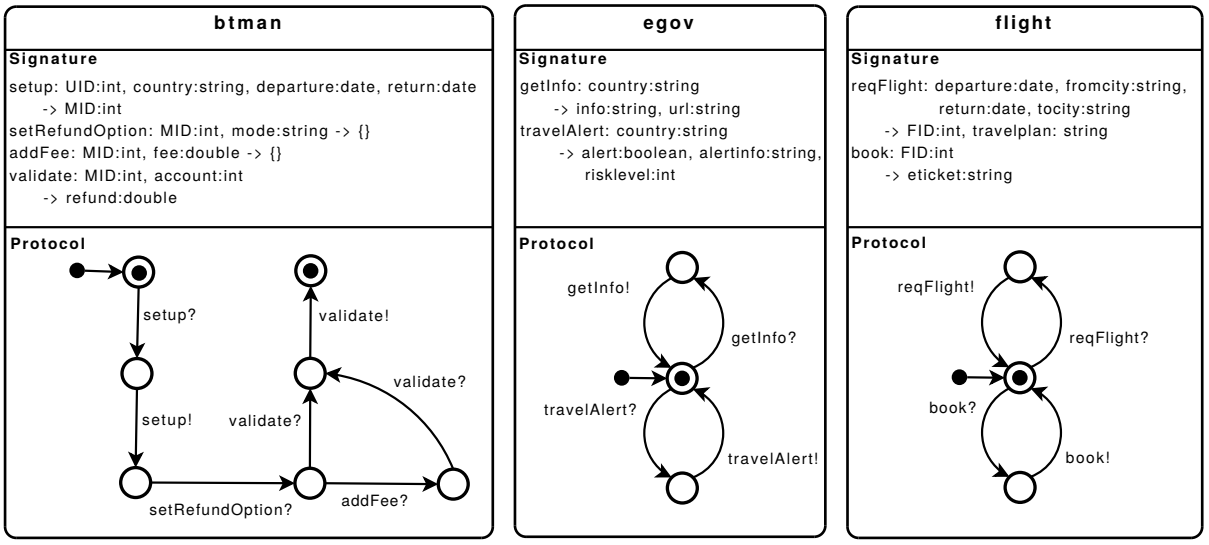

Fig. 4: eTrip – signatures and protocols: (left) business travel manager, (middle) e-government service, (right) flight service

enables us to keep a symbolic and finite representation for these data, therefore we do not suffer from state explosion problems in our approach. To the contrary, with an LTS one would either have (i) to discard completely data associated to service message exchanges, or (ii) rely on a finite set of constants for data and, accordingly replace any symbolic transition (*e.g.*, $add?x$) by all possible ground ones (add_0, add_1, add_2, etc.), yielding state explosion. The operational semantics of our STS model can be found in [25]. The scope of a variable received in a transition corresponds to all subsequent transitions. Moreover, if a variable is to be received again, then its former value is replaced by the new one. We have a simpler form of STS *wrt.* the ones that can be found, *e.g.*, in conformance testing or symbolic execution (see [26] for an example of this). Here we abstract guards that exist in these more expressive models (enabling or not a transition based on data values) by τ transitions. However, as far as adaptation is concerned, our techniques deal with any possible evolution. Avoiding abstraction can be supported with full-fledged STS, still, this would require a more complex machinery with, *e.g.*, symbolic execution and the use of SMT solvers.

Services. A service is a couple $(\rho, \mathcal{B})$ made up of a partnership, ρ, and a behavioral description of its protocol defined with an STS, $\mathcal{B}$, whose labels (A) correspond to events built on the partnership signatures.

Application. In this article, we use a business travel application as case-study (eTrip). This application is composed of three services: a business travel manager, btman, an e-government service, egov, and a flight information and booking system, flight. We also give an example of user requirements, user, representing the goal of the composition-to-be. Here, we present first the behavior of each service that can be reused through its public interface (see Fig. 4 for a graphical overview of the different interfaces involved in this example).

The business travel manager service is representative of services with a conversation. It is supposed to receive first a mission setup request (setup), and returns back a mission identifier. Then, the manager is informed about the refund mode (setRefundOption) which can be either fixed-amount or actual expenses, it optionally receives the information on travel expenses (addFee), and terminates with a validation interaction (validate). The e-government service supports citizen traveling information in two distinct ways: one may ask whether a country presents a risk (travelAlert), or ask for more general information about a country (getInfo). This service is representative of existing services provided, *e.g.*, by the US Department of State or by the EU Ministries of Foreign Affairs. The e-government service has no conversation (WSDL service). Such services are easily integrated into our framework by using a simple conversation for them enabling to invoke their operations in any ordering (using loops). Finally, the flight service provides information on flights (reqFlight). Given departure/return dates and cities, it returns a flight reference number and a travel plan. This service may also be used to actually reserve a seat on an identified flight and get a corresponding electronic ticket (book).

Now, let us present the user requirements we will use (Fig. 5). It corresponds to the goal of our composition. It could also guide the service discovery process that would propose the aforementioned services as possible candidates to our composition. However, since our topic is the composition and adaptation process, we will not address discovery here (see for instance [27] for more information). First, the user wants some information about the country (s)he plans to travel to (getInfo). This includes information on whether an alert exists or not for that country. If a risk exists, the user decides to cancel the

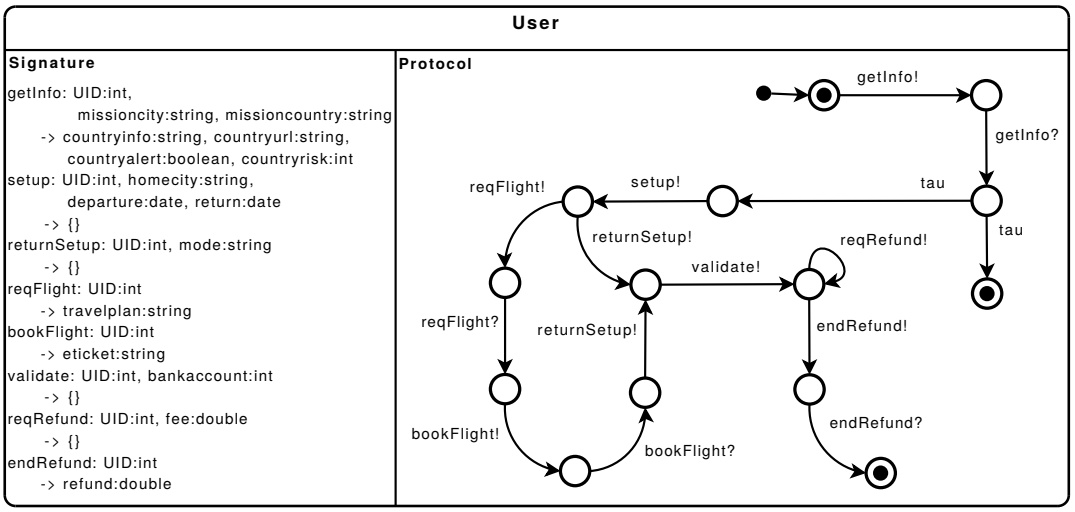

Fig. 5: eTrip – signatures and protocols: user requirements

trip (termination). Otherwise, the user starts the organization of the trip (setup), and possibly requests tickets (reqFlight) and books them (book). Upon returning from the travel, the user chooses a refund mode (returnSetup), and sends her/his bank account number (validate). In case of an actual expense refund mode, expenses are sent (reqRefund). Finally, the user wants to be informed on the refund amount (endRefund).

There are several interesting points in the user behavior. First, one can see that there are two different ways to express choice. After the getInfo? message, there is a branching of τ transitions. This typically corresponds to some condition in, *e.g.*, the code of a GUI client used by the user to access services (if there is a risk the user stops, else (s)he continues). At the level of the STS, this choice is specified as an internal choice and τ actions are used to encode this non-deterministic behavior[1]. Slightly later in the protocol, one can see another choice that is to be performed by the user, this time between asking for a flight ticket (reqFlight!) or directly performing actions related to return from traveling (returnSetup!). There, one does not have a τ branching because such a choice is not encoded in some piece of GUI code, it is rather dynamically performed by the user. One may note that the user can only seek one flight and has then to book it. This behavior has been chosen for the sake of simplicity.

Since the services have been designed independently, and without prior knowledge on our user needs, there are several mismatches that would, without adaptation, prevent one to fulfill the user needs with the services at hand. Let us present some of these mismatches.

- (1-1 mismatch) there are name mismatches between some operations required by the user and related operations in the services. For example, the operation that starts the refund process is called returnSetup in the user requirements while it is called setRefundOption in the travel manager. Similarly, operation addFee provided by the travel manager corresponds to reqRefund in the user requirements – an adaptor should transform messages when required.
- (1-0 mismatch) the endRefund message sending in the user requirements has no counterpart in any other service – an adaptor should not transmit unexpected messages.
- (n-m mismatch) the user is using a single operation, getInfo, to get information on a country, while two distinct operations in the e-government service, getInfo and travelAlert, should be used for this – an adaptor should invoke two operations, or more, when required, to fulfill the needs for a single one.
- (reordering) some messages are exchanged in different order by the user and the services, this is for instance the case between the validate! reqRefund! sequence in the user and the addFee? validate? sequence in the travel manager – an adaptor should reorder messages to avoid deadlock in interactions.
- (data mismatch) as far as the data transmitted along with messages is concerned, there are also some mismatches. For example, the travel alert information returned by the travel manager (travelAlert!) is not required by the user. Information used to seek a flight (reqFlight?departure,fromcity,return,tocity) with the flight service are not all provided by the corresponding call in the user. Indeed, it comes from different user calls – an adaptor should be able to remove some data parts in a message or, conversely, aggregate data from several messages.
- (correlation transparency) the user is not aware of correlation information used in the travel manager

1. A single choice semantics is available in STS, the external choice. Therefore, one needs to use this pattern to encode internal choices. In higher-level languages such as process algebras (*e.g.*, CSP [28]), these operators (internal and external choices) are distinguished in the syntax.

(the MID mission identifier). Instead, the user sends with each message her/his own identifier (UID) – an adaptor should store correlation information and use it transparently.

Note that our case study does not present all the possible cases of mismatch, a situation which rarely occurs in the real world. We can also work on services in which an interaction is shared by more than two participants (broadcast communication for instance).

In the next section, we present our adaptation contract notation. This is an abstract language which enables the designer to specify how mismatches can be solved and which is used in a second step as input to our adaptor generation algorithms.

3 Adaptation Contracts

An adaptation contract specifies how messages and data exchanged between services are related. Consequently, this specification indicates how some cases of mismatch can be solved (*e.g.,* making explicit that two messages with two different names correspond to an interaction). Some other cases (reordering of messages or data) will be worked out by our adaptor generation algorithms (see Section 4), which use as input an adaptation contract but also the service interfaces, and the description of the user requirements.

In this article, we use *vectors* and a *vector-Labelled Transition System* (vector LTS) as adaptation contract specification language [8]. A vector expresses 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. These correspondences are used by our adaptor generation techniques to solve data mismatches (unmatched parameters, different orderings, etc.). For instance, when two services exchange some parameters in a different order, the adaptor will successively interact with the sender and the receiver to solve the problem. However, in this work we assume no typing issues and we check that placeholders are used in vectors consistently *wrt.* data types (see Section 5).

Definition 2 (Vector): A *vector* v for a set of service STSs $(A_i, S_i, I_i, F_i, T_i), i \in [1, n]$, is an element of $(A_1 \cup \{\varepsilon\}) \times \ldots \times (A_n \cup \{\varepsilon\})$. Such a vector is denoted $\langle s_1 : l_1; \ldots; s_n : l_n \rangle$ where s_i are service identifiers, and l_i are labels built on the service alphabet A_i where message parameters are substituted by placeholders relating the arguments. If a service s_i is not involved in an interaction, its label is defined as ε, resulting in $s_i : \varepsilon$. For the sake of simplicity, the entry for s_i can also be removed from the vector.

Let us illustrate with a part of the example given in Figure 1 how vectors can be used to solve a mismatching interaction. Let us suppose that C has already sent setF and that it has been received by the adaptor. Now C is to send perform to S, which indeed has a setA operation. The two services present different message names, and the first one sends an additional parameter (an action, but also an identifier). The following vector specifies the interaction between both partners: $\langle$C : perform!CID, ACTION; S : setA?ACTION$\rangle$. This vector relates message perform with setA, but also parameters using placeholders. For instance, the parameters corresponding to the action in both labels are related in the vector using a fresh variable ACTION to indicate that these parameters match. In Figure 6, we show the corresponding part of the adaptor (its generation from an adaptation contract will be explained in details in the rest of the article). The adaptor synchronizes with the services using the same name of messages but the reversed directions, *e.g.,* communication between setA? in service S and setA! in the adaptor.

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 messages in another service, which requires application of 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.

Definition 3 (Adaptation Contract): An adaptation contract for a set of services STS_i, $i \in [1, n]$, is a couple $(V, VLTS)$ where V is a set of vectors for services STS_i, and $VLTS$ is a vector LTS built over V.

If only message names and data correspondences are necessary to build the adaptor-to-be (no ordering constraints in the application order of vectors), the vector LTS can leave the vector application order unconstrained using a single state and all vector transitions looping on it. In particular, this pattern can be used on specific parts of the contract for which the designer does not want to impose any ordering (see an example of this in Fig. 7).

In order to avoid the manual writing of adaptation contracts, a tool-based approach [14], called Acide, has been recently proposed and provides interactive support to guide the designer in the specification of adaptation contracts. Acide relies on graphical interfaces, hierarchical architectures, and compatibility degree measures. Acide makes the design of contracts much easier and user-friendly. It was developed as a standalone applica-

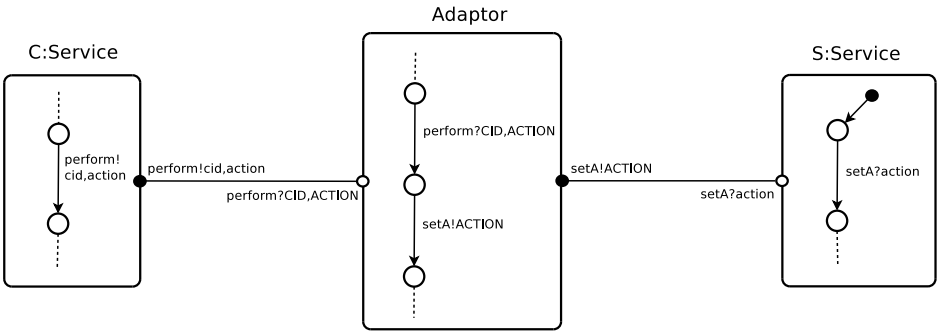

Fig. 6: Interaction between adaptor and services for a vector

tion fully dedicated to software adaptation. Still, it could be integrated as a plug-in in a more general IDE such as Eclipse.

Application. The very first step in the construction of an adaptation contract is to relate messages, *i.e.*, building the architecture of the composition-to-be. Next, since we are also considering value passing, data exchanged through messages have to be matched. In the rest of this section, we will focus on the textual version of this contract to explain how vectors help our adaptor generation algorithms to solve the mismatches existing for our running example. We will also present examples of vector LTSs that can be built for it. There are 15 vectors (V) in the contract of our example (Figure 7).

First of all, even if it is not the case here, some exchanged messages may match perfectly. Vectors would in this case contain pairs of matching messages. Notice that such vectors can be generated automatically from the service protocols. As regards 1-1 mismatch, vectors are defined with the different message names inside, see for example VbookRep that specifies the correspondence between the two different messages used for the response to a booking request, bookFlight in the user and book in the flight service. Unspecified reception is dealt with vectors with a standalone message, *e.g.*, VendRefundReq, which specifies that the endRefund message coming from the user should not be sent to any service.

The possible 1-2 correspondence between operation getInfo required by the user and operations getInfo and travelAlert in egov is specified using vectors VgetInfoReq1, VgetInfoReq2, VgetInfoRep1, and VgetInfoRep2. The first one expresses that whenever a getInfo message is received from the user, then a getInfo message should be sent to egov, still, removing some data that is not required. The second one specifies that a travelAlert request may be sent to egov as soon as some country information is known. This information is indeed received by the adaptor using the first vector, and stored in variable TOCOUNTRY. This schema is typical of a set of vectors to solve 1-n mismatch. Accordingly, the last two vectors support returning information back to the user. One can see that the return information is gathered from different messages, namely getInfo? and travelAlert?.

An important added-value of our adaptation algorithm is that one does not have to specify any specific solution for reordering. It is the adaptation process itself that will detect all possible application orderings of vectors, and accordingly possible message reorderings that enable interaction between services while avoiding deadlocks.

One can see the important role played by variables in vectors. Their scope is not limited to a vector, but rather corresponds to the whole set of vectors. Variables may therefore be used to gather information and also to make correlation with sub-services transparent. For example, all messages (but setup) can be sent to the travel manager as soon as the MID information required for correlation is available. This information is received using vector VsetupRep. To detect possible inconsistencies in the variables used in vector definitions, *e.g.*, a data needed in some message that may never be received before, one can use vector static analysis techniques we have implemented or check automatically placeholder occurrence properties (see Section 5.4).

Let us imagine now the same application but with some additional constraints. Using the set of vectors as-is, once the user has sent a getInfo request, the adaptor can send travelAlert requests to egov several times in a row. However, this repetition (looping behavior) is not desired because it increases the adaptor size and complicates its behavior unnecessarily. In order to make the adaptor send travelAlert requests only once, we can take into account in the adaptation contract the vector LTS represented in Figure 7 (above right). One can see there that vector VgetInfoReq2, enabling calls to travelAlert, is only possible once.

Let us go further with vector LTS-equipped adaptation contracts. A vector LTS may also be used to enforce that the user gives some expense information. This is represented in Figure 7 (below right), which builds on the previous contract. There, one can see that the two vectors handling the end of the user request (VendRefundReq and VendRefundRep) are forbidden until the vector for giving expenses (VaddFeeReq) is executed.

Such vector LTS contracts are very expressive means to specify adaptation constraints that go beyond solving

VgetInfoReq1 = ⟨user : getInfo!UID, TOCITY, TOCOUNTRY; egov : getInfo?TOCOUNTRY⟩
VgetInfoReq2 = ⟨egov : travelAlert?TOCOUNTRY⟩
VgetInfoRep1 = ⟨user : getInfo?CINFO, CURL, CALERT, CRISK; egov : getInfo!CINFO, CURL⟩
VgetInfoRep2 = ⟨egov : travelAlert!CALERT, CALERTINFO, CRISK⟩
VsetupReq = ⟨user : setup!UID, FROMCITY, DEPARTUREDATE, RETURNDATE; btman : setup?UID, TOCOUNTRY, DEPARTUREDATE, RETURNDATE⟩
VsetupRep = ⟨btman : setup!MID⟩
VreqFlightReq = ⟨user : reqFlight!UID; flight : reqFlight?DEPARTUREDATE, FROMCITY, RETURNDATE, TOCITY⟩
VreqFlightRep = ⟨user : reqFlight?TRAVELPLAN; flight : reqFlight!FID, TRAVELPLAN⟩
VbookReq = ⟨user : bookFlight!UID; flight : book?FID⟩
VbookRep = ⟨user : bookFlight?ETICKET; flight : book!ETICKET⟩
VreturnReq = ⟨user : returnSetup!UID, MODE; btman : setRefundOption?MID, MODE⟩
VvalidateReq = ⟨user : validate!UID, ACCOUNT; btman : validate?MID, ACCOUNT⟩
VaddFeeReq = ⟨user : reqRefund!UID, FEE; btman : addFee?MID, FEE⟩
VendRefundReq = ⟨user : endRefund!UID; ⟩
VendRefundRep = ⟨user : endRefund?REFUND; btman : validate!REFUND⟩

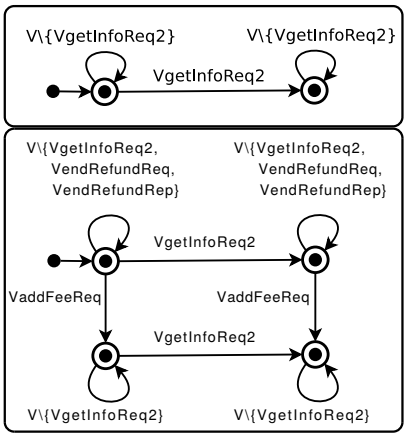

Fig. 7: eTrip – vectors and vector LTSs

mismatches. They can be used to specify rules specific to the service coordination.

4 Principles of the Encoding

An adaptor for a set of services is a process that orchestrates them in such a way that deadlock situations are avoided and that the ordering of messages specified by the adaptation contract is guaranteed. An adaptor plays a central role for the services it adapts, all exchanged messages pass through it, and it guides their execution. Building an adaptor means first to design its model. One has to take into account different constraints when designing this model. First, an adaptor has to act non-intrusively, *i.e.*, from the outside, without changing the services' code. Therefore, an adaptor should adhere to the service specifications, *i.e.*, the interactions between an adaptor and a given service correspond to this service's conversation. Each time the adaptor sends a message, the service should be ready to receive it. Further, each time the service sends a message (including replies to invocations), the adaptor should be ready to receive it. The adaptor has also to respect the system-level properties specified in the adaptation contract. All these constraints make the manual design of an adaptor a complicated and error-prone task. Automated adaptor generation and verification techniques are therefore of real interest.

The reader who is not interested in the details of the encoding may safely skip this section and can directly go to Section 5.

4.1 Motivating the Use of a Process Algebra

The adaptation constraints may be used to generate adaptors automatically, by encoding them into an executable formalism. This technique has recently been applied using LTSs [5]. Each constraint is encoded as an LTS and the conjunction of the constraints is achieved with a product of these LTSs. The adaptor model corresponds to the product where paths leading to deadlocks are pruned. In [8], Petri nets are used instead of LTSs. This enables to support message reordering. An important drawback in both cases is that the complete state space of the adapted system has to be computed before pruning can be achieved.

In our work, we chose to encode the adaptation constraints into a process algebra, namely LOTOS [29]. This choice is motivated by several reasons. First, the use of a standardized language (LOTOS being an ISO standard) enables one to take advantage of tools available for it and fosters adaptor model inter-operability between different phases of the adaptor engineering process (*i.e.*, generation, verification, implementation). Second, LOTOS is a very expressive behavioral specification language. It supports the specification of complex concurrent systems involving data in their exchanges. To the contrary, most languages and tools for automata or discrete event systems simply abstract from these data. This is clearly not possible when services are concerned. LOTOS is supported by CADP [12], a toolbox which implements optimized state space exploration techniques as well as verification tools. Finally, and more importantly, the use of a process algebra such as LOTOS enabled us to define an *on-the-fly* approach to adaptor computation, avoiding the complete construction of the adapted system state space that would correspond to the product in existing adaptation approaches. Thus, using LOTOS is not only a matter of expressiveness but also of efficiency.

In the following we present the principles of our encoding. We advocate that these principles could be applied to other process algebras as soon as they share with LOTOS the features presented above and the same semantics for the operators we use (see below, and [30] for the LOTOS semantics). Therefore, we will present the encoding in a language-independent way, with generic process algebraic operators: $proc\, P := def$ for process definition ($proc\, P = def\, endproc$ in LOTOS), $+$, generalized as Σ, for choice ([] in LOTOS), . for sequence (; and $>>$ in LOTOS), $if\, cond\, then\, ...$ for conditional ($[cond] \rightarrow$ in LOTOS), $||_A$ for concurrent processes communicating on A ($|[A]|$ in LOTOS), $|||$ for interleaved (non communicating) concurrent processes ($|||$ in LOTOS), $P \backslash A$ for

hiding a set of events A in a process P ($hide\ A\ in\ P$ in LOTOS), and 0 for termination ($stop$ in LOTOS). We also suppose that communication in the process algebra is synchronous, *i.e.*, two communicating processes interacting on some action a can only progress if both do a at the same time (in other words, this means a corresponds to the synchronizing of a constraint a in the two processes). Finally, we assume that symbols $\uparrow$ (emission) and $\downarrow$ (reception) in the process algebra support data transfer. It should be noted that these correspond respectively to symbols ! and ? in LOTOS. Since these are also used in service STS, a pre-processing of service STS and vectors should be done before using LOTOS for encoding (we use $_EM$ for ! and $_REC$ for ?), and a post-processing afterward (the reverse way). In Section 5, we develop on how adaptors and adapted systems can be verified. We also demonstrate the benefits of our *on-the-fly* generation techniques.

4.2 Setting Up Adaptation Constraints

Central to the way adaptation constraints are encoded as different processes (the basic atoms of a process algebraic specification) is the way these constraints are related. Due to the interactive nature of these relations we present them using a sequence diagram notation in Figure 8 where data are omitted for readability (however, they are taken into account in the encoding, below). Please note that with reference to the standard sequence diagram notation we introduce parallel blocks (par) to express that several interactions occur in parallel. This yields a more concise representation.

In our encoding, we have different kinds of processes:

- s_i:ServiceProtocol (one for each service): these processes encode the service conversations, *i.e.*, the ordering in which this service accepts or sends messages (accordingly, the ordering in which the adaptor may send or receive these messages);
- :Store (one): this process encodes data availability, *i.e.*, which data have already been received or not by the adaptor, hence which can be used in output messages;
- :VectorLTS (one): this process encodes the adaptation contract, *i.e.*, it imposes the order in which vectors can be successively applied;
- v:Vector (one for each vector): these processes encode the vectors, *i.e.*, the input and output messages for the vector, together with the fact that input messages should be received before output ones.

These processes synchronize to enforce ordering constraints over the reception/emission of messages. A specific action, FINAL, is used to denote correct termination of the processes. The fact that all processes end up synchronizing over FINAL denotes successful adaptation. In the following we present how each kind of process is encoded (which is denoted with $[|_|]$). We then focus on the description of their interaction.

Service conversation encoding. For each service s, its conversation $\mathcal{B} = (A, S, I, F, T)$ is encoded using a state-machine pattern. Each state q in S is encoded by $[|q|]$, the choice between the transitions outgoing from q, with an additional choice (using the FINAL action) if q is final:

$$proc\ q := \begin{cases} \Sigma_{(q,a,q')\in T}\,[|a|].q' + \mathsf{FINAL}.0 & \text{if } q \in F \\ \Sigma_{(q,a,q')\in T}\,[|a|].q' & \text{otherwise} \end{cases}$$

The process for s corresponds to the initial state of its conversation: $[|s|] = proc\ s := I$. One can easily see that for each transition in T, we generate exactly one transition in the corresponding process. We also add extra-transitions labeled with FINAL to make final states explicit since process algebras do not always distinguish deadlocks from correct final states.

In our context, the correct exchange of data will be ensured by the encoding of the adaptation constraints. Placeholders put the data received by the adaptor at some message(s) in relation with the one used by it to build sent message(s). Placeholders are the cornerstone of the adaptation constraints and are defined in vectors. Therefore, service labels are translated ($[|a|]$) as an input value-passing action (with $\downarrow$) with as many fresh variables as there are parameters coming with the message. It is the processes encoding vectors that will be in charge of setting the corresponding placeholders (with $\uparrow$). This is typical of a constraint-oriented style of using process algebras. Let $Var(V)$ be the set of all placeholders (names) used in the contract $(V, VLTS)$. The type for placeholder variables, PH, corresponds to an enumerated type over $Var(V)$.

Let us take the example in Figure 1 and the excerpt in Figure 6. We would have a transition labeled by C:perform!$\downarrow$ x_1:PH,x_2:PH in the encoding of C and a transition labeled by S:setA?$\downarrow$ x_1:PH in the encoding of S.

Store encoding. The $Store$ process is used to keep track of known data, or more precisely of which placeholders in vectors have already been set up. Looking at Figure 6, one can see that the setA message can be put in the adaptor model only after the perform one since the former uses a placeholder set up by the latter one, namely ACTION. From a constraint-oriented viewpoint, we do not need to know its value. It is only at run-time, when the adaptor implementation will receive and send messages, that values will be known and used. We may therefore associate a boolean value to each placeholder. Let $H : Var(V) \rightarrow \{0, 1\}$ be a function that associates a boolean value (initially false) to each var in $Var(V)$. Processes can interact with the store to set $H(var)$ to be true ($store_{var}$) or to check if $H(var)$ is true ($read_{var}$). Thanks to the process algebra synchronous communication, whenever a process (see Vector encoding, below) requires a placeholder that is not set, it will block on $read$ until $Store$ is informed that this placeholder is set up by

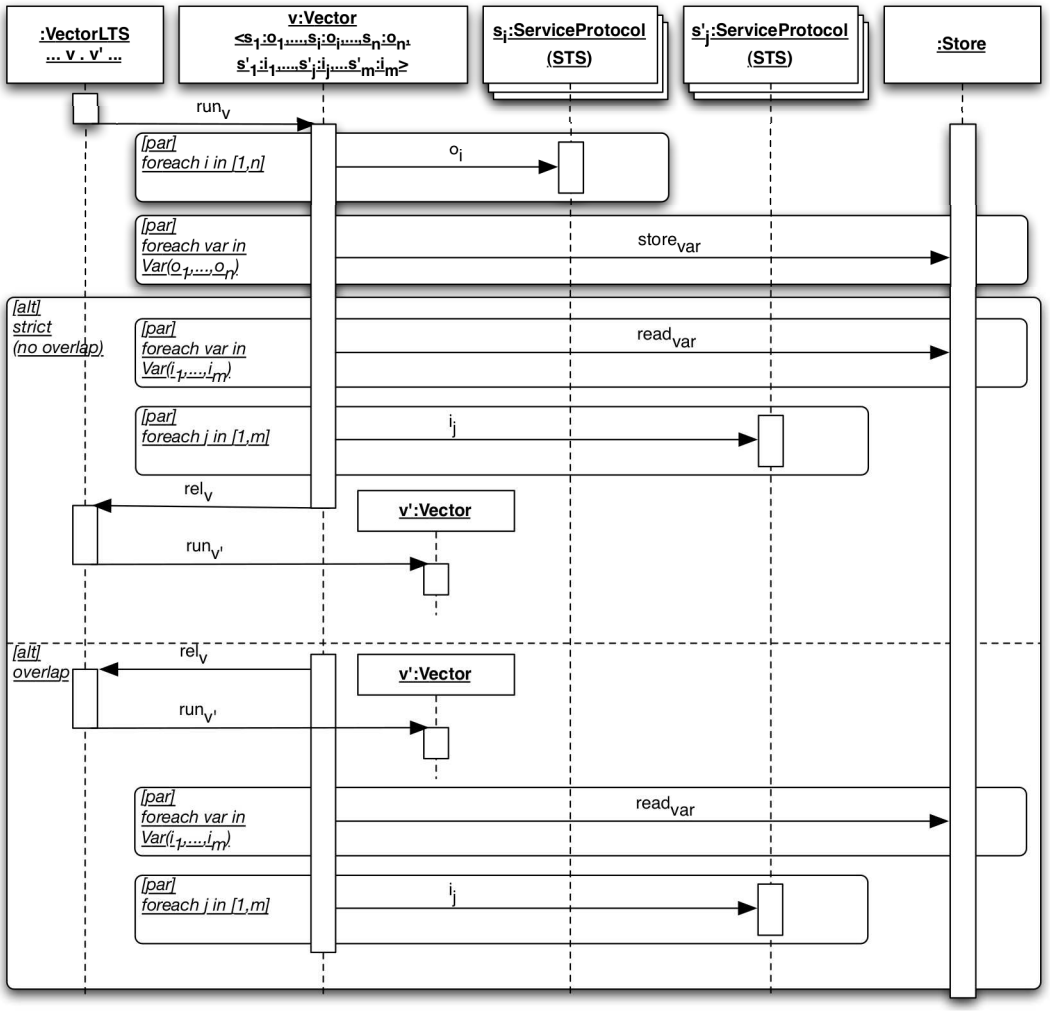

Fig. 8: Encoding pattern

another process. The encoding of the store, $[|Store|]$, is:

$$
\begin{array}{l}
proc\, Store(H) := \\
\Sigma_{var \in Var(V)} store_{var}.Store(H[var \mapsto 1]) \\
+\Sigma_{var \in Var(V)}(if\ H(var)\ then\ read_{var}.Store(H)) \\
+\mathsf{FINAL}.0
\end{array}
$$

where $H[var \mapsto 1]$ is H where the value for var is set to true.

Vector LTS encoding. The correct ordering of vectors is ensured by the vector LTS. It is encoded in the same way as for service conversations, but for the fact that each transition with label v (vector) corresponds to two actions. The first action (run_v) activates the corresponding vector process and suspends the vector LTS process until the second action (rel_v) releases it. For a vector LTS $L = (A_C, S_C, I_C, F_C, T_C)$ we have $[|L|] = proc\, L := I_C$ and for each q_C in S_C, $[|q_C|]$ is defined as:

$$
proc\, q_C := \begin{cases} \Sigma_{(q_C, v, q'_C) \in T}\, run_v.rel_v.q'_C & \\ +\mathsf{FINAL}.0 & \text{if } q_C \in F_C \\ \Sigma_{(q_C, v, q'_C) \in T}\, run_v.rel_v.q'_C & \text{otherwise} \end{cases}
$$

Vector encoding and overall encoding. We choose to present at the same time the vector encoding and the overall encoding since they are closely related (Fig. 8). We suppose for the sake of presentation that we are in the context of a vector LTS that prescribes the $v.v'$ sequence (v can be enabled and then v' will). Further, let v be a vector with n service outputs (*i.e.*, n adaptor inputs), $O = \{s_i : o_i\}$, $i \in \{1, \ldots, n\}$, and m service inputs (*i.e.*, m adaptor outputs), $I = \{s'_j : i_j\}$, $i \in \{1, \ldots, m\}$, where each o_i is of the form $opname_i!PH_{i,1}, \ldots, PH_{i,l_i}$ and each i_j is of the form $opname_j?PH_{j,1}, \ldots, PH_{j,l_j}$. For readability purposes, placeholders are not made explicit in Figure 8, where each arrow between a service process and a vector process (the o_i and i_j arrows) should be understood as a synchronization between the two processes. For example, the o_i arrow corresponds to a synchronization between $s_i : opname_i! \uparrow PH_{i,1}, \ldots, PH_{i,l_i}$ in the process for v and $s_i : opname_i! \downarrow x_1, \ldots, x_{l_i}$ in the process for s_i. Accordingly, the i_j arrow corresponds to a synchronization between $s'_j : opname_j? \uparrow PH_{j,1}, \ldots, PH_{j,l_j}$ in

the process for v and $s'_j : opname_j? \downarrow x_1, \ldots, x_{l_j}$ in the process for s'_j. Our objective is to build the legal orderings in which such adaptor events can occur. For this we use the constraints of the adaptation problem as presented above. The main idea here is to use process algebra synchronized communication (represented with arrows in Fig. 8).

The vector v is first enabled through a run_v synchronizing with the vector LTS process, which gets suspended. Next, the vector tries to synchronize with all service outputs, O, since all the corresponding messages must have been received before the adaptor can send messages to the service inputs, I. Still, the messages for O may be received in different orderings. Since we want to allow any possible one, we define v to receive the messages for O in parallel (interleaving). Once all these messages are received, v sets the corresponding placeholders to true by communicating with the store process. As far as the messages for service inputs, I, are concerned, v checks first that all required placeholders have been set up (either by v or another vector). If not, v waits until another vector receives the service message(s) with the pending placeholders. Once all placeholders are available, v may execute the I actions, again in any possible order, by synchronizing with the corresponding services.

When a vector has been completely executed, it can be run again once enabled by the vector LTS. However, several strategies are possible for a vector v process to release a suspended vector (using rel_v). The first strategy is to wait for the complete processing of a vector before launching a new one (Fig. 8, *strict* alternative block). The second one is to execute the release action once the reception of service messages has been done and before doing the sending part (Fig. 8, *overlap* alternative block). The latter is particularly interesting since it makes the reordering of messages possible, a typical case of mismatch between services. Let us take the example in Figure 1 and suppose that we have vectors Vfile = $\langle$C : setF!CID, FILE; S : run?SID, FILE$\rangle$, Vaction = $\langle$C : perform!CID, ACTION; S : setA?ACTION$\rangle$, VgetSID = $\langle$S : setA!SID$\rangle$, and that in the vector LTS we have the enabling sequence Vfile.Vaction.VgetSID. Without overlap, the adaptor would block once setF is received since to send run it needs SID to be set up, while it would not have been. With overlap, the adaptor may do the setF reception (setting up CID and FILE) corresponding to vector Vfile and suspend this vector outputs. Then the adaptor may do the perform reception (setting up CID and ACTION) and the setA emission corresponding to vector Vaction, and the setA result reception (setting up SID) corresponding to vector VgetSID. Finally, the vector Vfile can be un-suspended and the adaptor may do the run emission. This sequence is the beginning of the adaptor in Figure 1. However, there is no silver bullet: overlap yields larger state spaces when computing the adaptor models.

We give here the definition of $[|v|]$ for the overlap mode. The other one can be inferred from it and Figure 8.

$$
\begin{array}{l}
proc\, v := \\
(run_v \,.\, |||_{i\in[1,n]}[|s_i : o_i|] \,.\, |||_{var\in Var(o_1,\ldots,o_n)} store_{var} \\
\;.\, rel_v \,.\, |||_{var\in Var(i_1,\ldots,i_m)} read_{var} \,.\, |||_{j\in[1,m]}[|s'_j : i_j|] \,.\, v) \\
+ \mathsf{FINAL}.0
\end{array}
$$

The encoding of actions in vectors ($[|s_i : o_i|]$ and $[|s'_j : i_j|]$, above) corresponds to an output value-passing action ($\uparrow$) accompanied with the list of placeholders appearing in the corresponding action. This is essential to make the exchange of data work correctly between the services on the one hand and the adaptor on the other.

In order to complete the encoding, we have to define a process for the overall encoding. Since it corresponds to the conjunction of all constraints (service conversations, store, vector LTS, vectors), it is encoded as the synchronous parallel composition ($||$) of all processes defined above. Its process algebraic definition can be directly inferred from the message sequence chart in Figure 8: each time there is an arrow labeled with an action a between two processes x and y (*e.g.*, run_v between the vector LTS process and the vector v process), then x and y have to synchronize on a. Moreover, all processes synchronize on FINAL. Any action that does not correspond to some message appearing in the services (*e.g.*, "$run_$" and "$rel_$" actions, and all interactions with the $Store$ process) is hidden it represents internal actions of the adaptor. They will be removed by reduction steps when generating the adaptor from the process algebraic encoding. Given a set of n service STS $(A_i, S_i, I_i, F_i, T_i)$, $i \in \{1, \ldots, n\}$, and a vector LTS $(A_C, S_C, I_C, F_C, T_C)$ defined over a set $V = \{v_1 \ldots, v_m\}$ of m vectors, the overall encoding is:

$$
\begin{array}{rl}
proc\, Ad := & (\, Services \,||_{A^{\mathrm{S}}\cup\surd} \\
& (I_C \,||_{A^{\mathrm{V}}\cup\surd} Vectors \,||_{A^{\mathrm{X}}\cup\surd} Store)\,) \\
& \backslash (A^{\mathrm{V}} \cup A^{\mathrm{X}})
\end{array}
$$

where

$$
\begin{array}{l}
proc\, Services := (I_1 \,||_{\surd} \ldots \,||_{\surd} I_n) \\
proc\, Vectors := (v_1 \,||_{\surd} \ldots \,||_{\surd} v_m)
\end{array}
$$

with $\surd = \{\mathsf{FINAL}\}$, $A^{\mathrm{S}} = \bigcup_i A_i$, $A^{\mathrm{V}} = \bigcup_{v\in V}\{run_v, rel_v\}$, $A^{\mathrm{X}} = \bigcup_{v\in V, var\in Var(v)}\{store_{var}, read_{var}\}$. Once an adaptor has been computed from this encoding (see the next Section), communicating actions are relabeled removing $\uparrow$ and $\downarrow$ symbols since they are encoding-related only. Further, communicating actions are mirrored (*i.e.* ? and ! are reversed) since an adaptor mirrors service events.

Let us take again the example in Figure 1 and the excerpt in Figure 6. Vector Vaction would be encoded as:

$$
\begin{array}{rl}
proc\, v := & (run_v \,.\, \mathsf{C : perform!} \uparrow \mathsf{CID, ACTION} \\
& \;.\, (store_{\mathsf{CID}} \,.\, 0 \,|||\, store_{\mathsf{ACTION}} \,.\, 0) \\
& \;.\, rel_v \,.\, read_{\mathsf{ACTION}} \,.\, \mathsf{S : setA?} \uparrow \mathsf{ACTION} \,.\, v) \\
& + \mathsf{FINAL} \,.\, 0
\end{array}
$$

In the Ad encoding, v may synchronize with the labels for services, *i.e.*, C : perform! $\downarrow$ x_1, x_2 and S : setA? $\downarrow$

x_3, yielding the (observable) sequence C : perform! ↑ CID, ACTION. S : setA? ↑ ACTION. This is transformed into the adaptor sequence C : perform?CID, ACTION. S : setA!ACTION which achieves adaptation and data transfer when the adaptor is implemented and run.

4.3 Tool Support

The encoding (applied on the LOTOS process algebra) is fully automated by Compositor, a tool we have implemented. Compositor supports different modes (*strict*, *overlap*). Supported inputs are STSs in either XML format or AUT format (the textual format for representing automata in CADP) and contracts in XML format.

5 Adaptor Generation and Verification

This section is devoted to our methodology for automatic generation and verification of adaptors. We first present the principle of our on-the-fly adaptor generation. Second, we show its application on the running case-study and on several other examples of service compositions from our database. Last, we illustrate how the generated adaptors can be verified by model checking in order to assess their adequacy *w.r.t.* the desired behavior expected from the adapted system.

5.1 Principle of On-the-Fly Adaptor Generation

An adaptor can be obtained from the state space of the whole system (services, store, and adaptation contract) by keeping only the correct behaviors, which amounts to cutting the execution sequences leading to deadlock states. In the adaptation techniques that support deadlock elimination [5], [31], 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.

5.1.1 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 (see Fig. 9). 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 [32] formula $\langle\ true^*\ .\ \mathsf{FINAL}\ \rangle\ true$, which states the existence of a sequence going out of the current state and leading (after 0 or more transitions) to a transition labeled by the

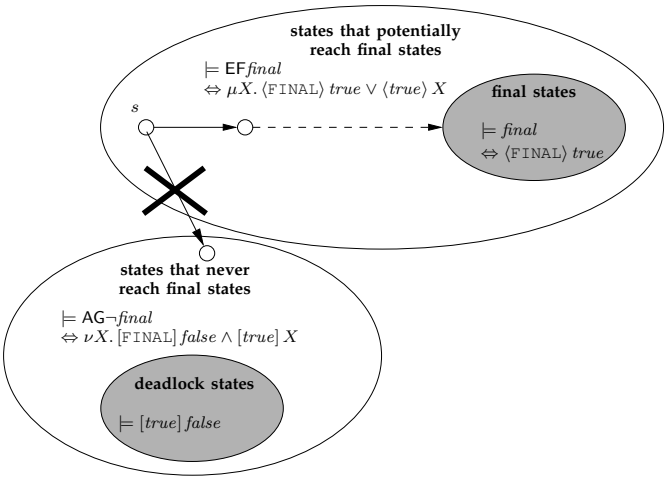

Fig. 9: Pruning deadlocks on-the-fly

FINAL action. This formula can be checked on-the-fly using the Evaluator [33] model checker of CADP. 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 obtained by translating the evaluation of this formula into the resolution of an equivalent boolean equation system (BES) [34]. This is done by first translating the formula into the equivalent modal μ-calculus [35] formula $\mu X.\ \langle\ \mathsf{FINAL}\ \rangle\ true \vee \langle\ true\ \rangle\ X$, by applying classical identities of PDL [32]. The propositional variable X defined by the minimal fixed point operator μ characterizes the states from which a transition labeled by the FINAL action can be reached: these are the states having either an immediate successor transition labeled by the FINAL action (modality $\langle\ \mathsf{FINAL}\ \rangle\ true$), or having at least one successor transition leading to a state satisfying X (modality $\langle\ true\ \rangle\ X$). The evaluation of this μ-calculus formula on a state space can be further expanded in terms of the resolution of a BES, following the classical translation of μ-calculus model checking into BES resolution [34], [36]. Basically, this translation builds a product between the formula and the state space, yielding the BES below:

$$\left\{ X_s =_\mu \bigvee_{s \xrightarrow{\mathsf{FINAL}} s'} true \vee \bigvee_{s \to s''} X_{s''} \right\}_{s \in S}$$

This BES consists of a set of minimal fixed point equations, each one defining in its left-hand side a boolean variable X_s indexed by a state $s \in S$ and having in its right-hand side a disjunctive boolean formula expressing whether the state s satisfies the modal formula $\langle\ \mathsf{FINAL}\ \rangle\ true \vee \langle\ true\ \rangle\ X$ or not. 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 Cæsar_Solve library [37] of CADP. The algorithm A3 is optimized for solving disjunctive BESs

in a memory-efficient way, by storing only the boolean variables and not the operators present in the right-hand sides of equations (and hence, it stores only the states and not the transitions of the state space). Moreover, the implementation of Cæsar_Solve ensures that a sequence of resolutions performed during a forward exploration of the state space has a linear-time accumulated complexity.

The use of BES as intermediate formalism to encode deadlock elimination allows to perform this task on-the-fly during a forward exploration of the state space, and to store only states in memory. This is nearly optimal from the on-the-fly perspective, since in the worst case (*e.g.*, when the system contains no deadlocks) one has to explore the whole state space in order to generate the adaptor.

5.1.2 Behavioral Reduction

Second, the adaptor STS obtained by pruning can be reduced on-the-fly (simultaneously with its generation) modulo an appropriate equivalence relation in order to get rid of the internal actions (τ) and obtain an adaptor as small as possible. These internal actions correspond here to the encoding of the system adaptation constraints, *e.g.*, hiding of run and rel actions. These τ actions must be removed to ensure that the generated adaptor does not contain non-controllable behaviors. Such non-controllability issues can still come from service interfaces, but we provide techniques to detect them in Section 5.4.3. Internal actions can be eliminated by applying on-the-fly reductions on the adaptor STS simultaneously with its generation.

We use the algorithms presented in [38] to implement on-the-fly reductions modulo τ-confluence (a form of partial order reduction that preserves branching bisimulation), and the $\tau^*.a$ and weak trace equivalences, both of which eliminate internal transitions and (for weak trace) determinize the adaptor STS. Among these reductions, τ-confluence preserves most of the branching behavior of the adaptor, by deleting only the redundant interleavings of independent actions; this may however reduce the size of the adaptor by several orders of magnitude, as observed in [38]. The worst-case complexity of these reductions depends on the relation adopted: τ-confluence is performed in linear-time *w.r.t.* the number of states, but in quadratic-time *w.r.t.* the branching factor of the state space (*i.e.*, the maximal number of transitions going out of a state), $\tau^*.a$ is carried out in quadratic-time, and weak trace in exponential-time for non-deterministic LTSs. However, the examples we encounter in practice appear to be quite far from the worst-case situations, and on-the-fly reductions exhibit a good performance.

5.2 Application

Using the methodology described above, we generated the adaptor protocols with respect to the first and second vector LTSs introduced in Figure 7. The complete protocol of the first adaptor (64 states and 103 transitions) is given in Appendix A. Here, we show in Figure 10 an excerpt of the second adaptor, which is slightly smaller (61 states and 95 transitions) due to the more restrictive nature of the second vector LTS. For the sake of clarity, we have cut all interleavings going out from states 11, 12, and 15, keeping however the essential parts of the adaptor in charge of solving mismatches. As explained in Section 4, the messages involved in the adaptor description are reversed to make synchronizations with the services possible. The adaptor starts by interacting with the user, receives the city and country concerned by this request (user:getInfo?UID, TOCITY, TOCOUNTRY), then submits either a request for information to the e-government service (egov:getInfo!TOCOUNTRY) or a request regarding a travel risk (egov:travelAlert!TOCOUNTRY). When the user has received from the adaptor all the requested information (user:getInfo!CINFO, CURL, CALERT, CRISK), (s)he arrives at state 9 and can decide to stop or start the trip organization (user:setup?UID, FROMCITY, DEPARTUREDATE, RETURNDATE), and so on.

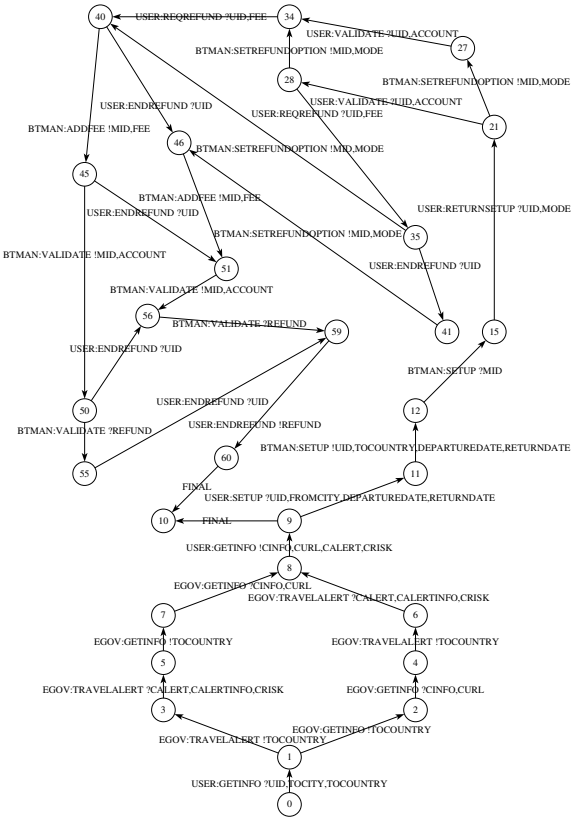

Fig. 10: eTrip – excerpt of the adaptor protocol using the second vector LTS from Fig. 7

In this adaptor, reordering of messages is required. Thus, if we focus on the upper left part of Figure 10, we can see that the adaptor interacts with the user on two messages concerning refund (user:validate?UID,

ACCOUNT and user:reqRefund?UID, FEE) before sending these information to the travel manager (btman:addFee!MID, FEE and btman:validate!MID, ACCOUNT). Indeed, the adaptor cannot interact on addFee with the travel manager before having received fees from the user.

5.3 Experiments

Table 1 shows experimental measures on 15 examples from our database, which contains about 250 examples that we used to validate our tools. For each experiment, the table gives the size of the "raw" adaptor STS generated from the LOTOS specification by keeping the states potentially leading to FINAL actions (columns 2, 3) and of the adaptor reduced on-the-fly modulo weak trace equivalence combined with τ-confluence (columns 4, 5). All the adaptors were generated using the first strategy mentioned in Section 4.2, *i.e.*, without overlapped execution of vectors. The largest example given in Table 1 ("eMuseum-016") took about one minute of computation (including the generation of both "raw" and reduced adaptors) on a standard desktop computer running Linux. In most cases, the size of the reduced adaptors is quite small, making possible their assessment by visual checking using the Bcg_Edit tool of CADP.

The rightmost columns of the table indicate the portion of the state space actually explored on-the-fly during the generation of the reduced adaptor, which can be significantly smaller than the whole state space (down to 51% of the states and 18% of the transitions). For the examples in Table 1, which have small and medium sizes, these gains have a limited impact; however, they can lead to significant memory savings for larger systems. This illustrates the benefits of on-the-fly adaptation *w.r.t.* the approach based on explicit construction of state spaces employed in [5], [6], [31]. Moreover, we observe that the ratio concerning the number of transitions explored is smaller than the ratio concerning the states. This aspect — which becomes more important with the increase of the branching factor of the state space, proportional to the number of services in the adapted system — further penalizes the explicit approach *w.r.t.* the on-the-fly approach, because the former requires to store all transitions in memory in order to compute the adaptor, whereas the latter allows to store only states.

5.4 Adaptor Verification

The adaptor generated by our approach respects its adaptation contract and is free of deadlocks. However adaptation contracts are an input of the adaptation process, and their writing requires human intervention. Since contracts are specified by the designer, they can contain errors that will also appear at the level of the adaptor. Indeed, this specification may not correspond exactly to what the designer expects from the system-to-be. Our verification techniques aim at analysing the resulting adaptor to be sure that the designer has not made any mistake while writing down the adaptation contract. An adaptation contract is an abstract specification of how existing mismatches can be worked out, but does not give all the execution scenarios corresponding to the adaptor protocol. To be sure to avoid erroneous executions of the adaptor, we need to analyse its whole state space, and not only the adaptation contract.

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. There are three groups of checks, determining respectively whether (i) labels are correctly used in vectors *w.r.t.* their definitions in service interfaces (labels defined in interfaces, labels used with the correct number of parameters, etc.), (ii) vectors and vector LTS are structurally consistent (vectors defined only once, all vectors used in the vector LTS defined beforehand, all states defined, no states or vectors unused, at least one reachable final state, etc.), (iii) placeholders have a consistent scope and type (same variable not received more than once in one vector, no variable sent before being received, variables relating message arguments with same type, etc.).

These static analysis features are very useful for detecting the simple errors that one can make while writing a contract manually. Nonetheless, this is not sufficient 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. Given that the behavior of the adaptor accurately reflects the behavior of the system after adaptation (consisting of the service and adaptor STSs running concurrently), to ensure proper functioning of the adapted system it is sufficient to verify the adaptor only (except for the controllability property 5.4.3). We illustrate below the verification using Evaluator of certain typical temporal properties on the adaptor generated for our running example shown in Figure 10.

5.4.1 General Properties

These are related to the adaptor structure, and should be satisfied by any adaptor generated using our approach. The goal is to check that the designer has not made any mistake while relating placeholders and messages in the adaptation contract. If such a mistake is introduced in the contract, our adaptation algorithms might be unable to generate the corresponding part of the adaptor protocol (*e.g.*, it is impossible to send a data that has not been received beforehand). The techniques we propose below aim at detecting this kind of issues in adaptation contracts. We distinguish two such properties:

- *Placeholder occurrence* means that any placeholder present in the adaptation contract must also occur in the adaptor. An example of such property (written as a PDL modality in the syntax of Evaluator) for

TABLE 1: Examples of adaptor generation

Application	Adaptor STS				State space portion explored for reduced adaptor generation			
	raw		reduced					
	states	trans.	states	trans.	states	%	trans.	%
eMuseum-016	21418	48692	978	2382	29026	72.8	17075	18.7
pervasive-music-system-010	1720	4368	49	60	14805	85.9	32923	74.5
sql-server-012	1720	4264	22	26	2337	57.1	3427	32.9
flower-014	764	1561	33	64	926	95.2	1690	84.1
sql-server-015	534	1133	20	23	691	92.8	1101	72.9
sql-server-011	488	995	22	26	2337	57.1	3427	32.9
mail-system-007	418	1059	418	1059	13630	99.7	23946	70.1
double-contract-005	284	539	20	24	2800	83.1	6798	75.7
pc-store-001	253	472	16	16	782	88.2	1208	66.8
rate-service-003	241	483	28	32	400	52.6	675	37.2
cs-loop-011	199	325	20	24	1592	95.3	3303	89.2
video-on-demand-016	149	231	17	22	251	97.6	260	63.5
batchsql-009	137	239	31	43	429	67.1	276	21.6
restau-booking-001	94	108	33	37	264	99.6	280	83.1
pc-store-004	17	17	17	17	237	91.5	249	64.3

the adaptor in Figure 10 is:

$$\langle\ true^*\ .\ \text{'.*!UID,TOCOUNTRY,DEPARTUREDATE RETURNDATE'}\ \rangle true$$

Here '.*!UID, TOCOUNTRY, DEPARTUREDATE, RETURNDATE' is a predicate (regular expression on character string) matching all actions containing an emission of the placeholders UID, TOCOUNTRY, DEPARTUREDATE, and RETURNDATE. These properties are extracted mechanically from all the placeholders contained in the adaptation contract. A placeholder present in some synchronization vector but not in the adaptor denotes a value-passing problem in the adaptation contract, which means that the execution sequences involving that placeholder have been cut off by the adaptation process.

- *Service action preserving* means that every action present in some service STS and occurring in some synchronization vector of the adaptation contract should also occur (modulo mirroring of emissions and receptions) in the adaptor STS. An example of such property for our adaptor is the following:

$$\langle\ true^*\ .\ \text{"flight:book ?ETICKET"}\ \rangle\ true$$

Every such property that fails on the adaptor STS may denote a problem in the adaptation contract, which should try to keep as many service actions as possible in the final adapted system. Again, properties of this kind are generated automatically from the service STSs.

As regards our running example, these two kinds of temporal properties were successfully verified on the two adaptor STSs (shown in Appendix A and Fig. 10) using the Evaluator model checker of CADP.

5.4.2 Specific Properties

These are related to the adaptor protocol, which differs from one adaptor to another. We consider here two classic types of properties for reactive systems:

- *Safety properties* express that "something bad never happens" during the execution of the adaptor. An example of safety property of our adaptor, expressed as a PDL modality, is that the user cannot be reimbursed before providing the necessary fee justifications:

$$[\ true^*\ .\ \text{"user:setup ?UID,FROMCITY, DEPARTUREDATE,RETURNDATE"}.\ (\neg\text{"user:reqrefund ?UID,FEE"})^*\ .\ \text{"user:endrefund !REFUND"}\ ]\ false$$

This property is satisfied by the adaptor STS shown on Figure 10. Notice that the same property may fail on adaptors generated from more permissive adaptation contracts (Fig. 7, above right), for instance if the requirement that users must submit expenses receipts upon coming back from traveling is dropped. In that case, the user has the possibility of choosing the fixed-amount option, which allows the reimbursement without requiring fee justifications. When checking the above property on the adaptor generated from the more permissive contract, Evaluator provided the following (shortest) counterexample sequence of 16 transitions corresponding

precisely to this scenario:

”user:getInfo ?UID,TOCITY,TOCOUNTRY” .
”egov:travelAlert !TOCOUNTRY” .
”egov:travelAlert ?CALERT,CALERTINFO,CRISK” .
”egov:getInfo !TOCOUNTRY” .
”egov:getInfo ?CINFO,CURL” .
”user:getInfo !CINFO,CURL,CALERT,CRISK” .
”user:setup ?UID,FROMCITY,DEPARTUREDATE,
RETURNDATE” .
”user:returnSetup ?UID,MODE” .
”user:validate ?UID,ACCOUNT” .
”user:endRefund ?UID” .
”btman:setup !UID,TOCOUNTRY,
DEPARTUREDATE,RETURNDATE” .
”btman:setup ?MID” .
”btman:setRefundOption !MID,MODE” .
”btman:validate !MID,ACCOUNT” .
”btman:validate ?REFUND” .
”user:endRefund !REFUND”

Such diagnostic sequences exhibited by the model checker as counterexamples for safety properties can be interactively replayed in the adaptor STS using, *e.g.*, the Ocis simulator of CADP, and are very helpful in identifying the cause of errors.

- *Liveness properties* express that “something good eventually happens” during the execution of the adaptor. A liveness property of our adaptor is that users will always be reimbursed once they set up a travel:

$$[\ true^*\ .\ \text{”user:setup ?UID,FROMCITY, DEPARTUREDATE,RETURNDATE”}\ ]\ \mu X.(\langle\ true\ \rangle\ true \wedge [\ \neg\text{”user:endrefund !REFUND”}\ ]\ X)$$

The minimal fixed point subformula μX specifies the inevitable reachability of a ”user:endrefund !REFUND” action. The above property holds on the adaptor STS shown on Figure 10.

5.4.3 Adaptability Property

A successful checking of the general and specific properties shown above on the adaptor STS guarantees that the adaptation contract accurately reflects the designer’s wishes and the adaptor protocol satisfies the desired correctness properties. Nevertheless, the system composed of the adaptor and service STSs may still contain deadlocks caused by non-controllable internal choices present in some service STSs, which prevent the adaptor from adequately guiding their execution. Suppose an adaptor built from the following vector $\langle \mathsf{c} : \mathsf{update!}; \mathsf{s} : \mathsf{update?} \rangle$ which first interacts with a client submitting an update! request (Fig. 11). In this example, we can see that the client starts his/her behavior with an internal choice. Given the contract above, this system is non-adaptable because the adaptor cannot prevent the client to enter the right-hand side branch of the choice. In such a case, the client deadlocks because the adaptor has not been built to accept this interaction (no corresponding vector in the contract). The adaptability property aims at detecting this kind of problem.

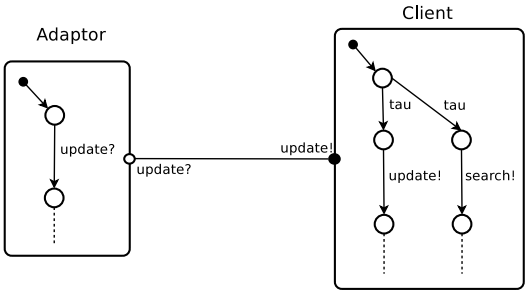

Fig. 11: Internal choice

Note that we cannot check this property while generating the adaptor because we are not able to differentiate between deadlocks that must be pruned (see Section 5.1) and deadlocks due to this non-controllability issue. Consequently, we first build the adaptor (generation, deadlock pruning, reduction), and in a second step, we check if there are still deadlocks when we compose the resulting adaptor with the involved services. The presence of these deadlocks can be detected by checking the following property on the composed system:

$$[\ true^*\ ]\ \langle\ true\ \rangle\ true$$

The system composed of the adaptor STS shown in Figure 10 and the service STSs is deadlock-free. However, a more constrained adaptation contract could lead to deadlocks in the adapted system. This can happen for instance if the adaptation contract forces the user to set up a travel departure whatever happens. In this case, the generated adaptor will not contain the FINAL action following user:getInfo!CINFO, CURL, CALERT, CRISK anymore (Fig. 10, state 9), which will cause the composed system to contain an internal action leading to a deadlock, corresponding to the user’s decision to not travel (see Fig. 5).

The presence of such deadlocks in the adapted system denotes the non-adaptability of the services via the proposed adaptation contract. In such a case, we can generate an adaptor, but this adaptor will not be able to make services respect the contract. This can be corrected by modifying the adaptation contract in order to appropriately take into account the branches of internal choices contained in the services, *i.e.*, all the actions appearing after each τ transition involved in the choice must be captured by a corresponding vector in the contract.

5.5 Tool Support: Scrutator and CADP

The on-the-fly adaptor generation is implemented by the Scrutator tool that we developed using the Open/Cæsar [39] environment for graph manipulation

provided by CADP. 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. Besides the on-the-fly reductions currently offered by Scrutator (τ-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 minimized off-line (since the on-the-fly reductions do not necessarily yield a minimal adaptor) modulo strong or branching bisimulation using the Bcg_Min tool of CADP.

To automate the whole adaptation process, Compositor generates an SVL script [40] 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" the adaptor actions (*i.e.*, reversing emissions and receptions, ! and ?) as the adaptor acts as an orchestrator in-the-middle of the services; and pretty-printing the adaptor STS by translating its actions from a LOTOS-like syntax to a more user-friendly syntax.

6 From (A)BPEL to STS and from STS to BPEL

In order to foster the applicability of our approach to different service frameworks (BPEL, Windows Workflow Framework, SCA components, etc.), we follow a model-driven approach. The algorithms presented in the previous sections work at the model level. Our approach can be applied to a specific framework using, on the one hand, a model generation algorithm retrieving service models from service interface descriptions, and, on the other hand, a model transformation algorithm, generating an adaptor implementation from an STS adaptor model.

In this section, we present the application of our approach to the BPEL orchestration language. Preliminary experiments have been undertaken using the Windows Workflow Framework [20]. Adaptors are implemented here with a centralized point of view (orchestration). However, we advocate that distributed adaptation can be achieved by applying orchestrator or adaptor distribution algorithms [41], [42], [43], [44] before BPEL generation.

An important issue with BPEL is that the standard document [45] only draws an informal semantics for it. Further, BPEL engines play an important role in what would be "the" BPEL semantics. In the last years several semantics for BPEL have been proposed [15], but there is no commonly accepted one. However, automated techniques for orchestration generation (either automatic service composition or service adaptation) can be complemented with model-based conformance testing [46] in order to check that implementations are conform to their models.

6.1 (A)BPEL to STS Transformation

Web services are described using a WSDL signature (data types and operations). Additionally, stateful services feature a protocol, also called conversation or behavioral interface, which describes the ordering of legal operation calls. This can be described using different languages. Here, we rely on Abstract BPEL (ABPEL).

6.1.1 A Short Presentation of (A)BPEL

The BPEL language is a workflow language based on an expressive set of activities enabling the orchestration of Web services. While the BPEL objective is to define executable orchestrations (*i.e.*, running on some execution engine), ABPEL has been defined to describe non-executable abstract processes. ABPEL is based on the same set of activities as BPEL. In this work we support the main ones. Time, fault and termination handlers can be supported through extension, *e.g.*, following [26]. However, we advocate that the subset we support is the one that is to be found in (A)BPEL behavioral interfaces.

Communication activities specify the communications between service partners. receive(o,$(v_1,\ldots,v_n)$) denotes the reception by a service of a message request for operation o with received data stored in variables v_i. reply(o,$(v_1,\ldots,v_n)$) denotes the corresponding reply by the service with output data taken from variables v_i. Accordingly, a service can call a partner service operation o using invoke(o,$(v_1,\ldots,v_n)$,$(v'_1,\ldots,v'_m)$) where sent data are stored in variables v_i and returned values to be stored in variables v'_j. When the operation is one-way (no return) we simply write invoke(o,$(v_1,\ldots,v_n)$). Note that we only represent the operation name (o) in communication activities for simplicity. Taking into account both partner link p and operation name o, can be done through prefixing ($p:o$). *Assignment* (:=) supports data computation. exit denotes an empty (*e.g.*, terminated) process. In addition to these five *basic activities*, BPEL defines workflow-based *structuring activities*: sequence (sequence($P_1,\ldots,P_n$)) (without loss of generality we will assume n is 2, *e.g.*, sequence(P_1,P_2,P_3) can be also written sequence(P_1,sequence(P_2,P_3))), conditional activities (if(cond,P_{then},P_{else})) and loops (while(cond,P)). BPEL also supports multiple event processing (pick({onMessage $o_i,(v_{1_i},\ldots,v_{n_i})$: P_i}[,onAlarm P_{alarm}])) where evolution of the process is triggered depending on either reception of a given message (onMessage $o_i,(v_{1_i},\ldots,v_{n_i}) : P_i$ yields P_i will be executed when message requests for operation o_i are received) or on a timeout (onAlarm P_{alarm}). For simplicity reasons we assume here there is at most one alarm in a pick.

6.1.2 Transforming (A)BPEL Behavioral Interfaces into STS

With reference to the model presented above, we focus on the behavioral part because the retrieval of the partnership information (related to the WSDL files) is straightforward. The rules for transforming (A)BPEL to

STS are defined inductively on the process structure (Appendix B).

We take inspiration from our previous work on WF [20]. These rules are also closely related to the recent work in [9]. Intuitively, the meaning of our rules is the following:

- exit does nothing hence is restricted to a final state.
- assignments are internal hence are transformed into internal transitions, *i.e.*, labeled with τ.
- receive and reply correspond to a single transition, labeled respectively with a reception ($o?\ldots$) and an emission ($o!\ldots$).
- invoke for one-way operations is transformed like reply, while invoke for two-way operations corresponds to a sequence of two transitions like reply and then receive.
- sequence is transformed to respect the ordering of actions in the processes, therefore, it corresponds to unifying the end states of the former with the initial state of the latter.
- if corresponds to an internal choice, hence it corresponds to a new state and two τ branches representing the two possibilities and respectively going to the initial states of the two sub-processes.
- while corresponds to a looping behavior, transformed with a new state in which the condition is internally tested and two branches going either to the initial state of the sub-process (a new loop) or to a new final state (end of the loops).
- pick is transformed to a new state with as many outgoing transitions as there are onMessage possibilities. Additionally, a timeout for onAlarm is an internal event represented with a τ transition. In each case, the transition goes to the initial state of the respective sub-process.

6.2 STS to BPEL Transformation

Due to the generic nature of the model-driven approach, an adaptor model may contain parts which are not implementable in a given language (here, BPEL). Therefore, adaptor implementation is achieved in two steps. First, the adaptor model is filtered in order to remove all non-implementable parts. Then, the filtered adaptor model can be transformed into a BPEL orchestration. In the remaining of the section, we detail these two steps.

6.2.1 Filtering Adaptors to Enable Implementation

The role of filtering is to remove parts which could not be implemented in BPEL. Before presenting the filtering rules, we may define what can be implemented. First, even if in an adaptor model it is possible that several transitions outgoing from a state are emissions, this cannot be implemented since we would need some condition to select which message to send first. Second, it is not possible that a process receives a message and sends one at the same time. Finally, the message emission (operation call) and the message reception (result retrieval) of invocations to two-way operations should be done atomically (in sequence). In order to achieve implementability of adaptors, we use three rules.

Rule 1 (emission determinism). Whenever an adaptor has, in some state, a possibility for several emissions, then only one emission is kept.

Rule 2 (emissions or receptions). Whenever an adaptor has, in some state, a possibility for both (one or several) emissions and (one or several) receptions, either the emissions or the receptions are kept.

Rule 3 (invocation atomicity). The only thing the adaptor can do after the initial emission of an invocation to a two-way operation (*i.e.*, an emission not targeted at the user partner link) is the corresponding reception. Formally, this means that for every two-way operation o of some S_i, $S_i \neq$user, for every $t = s \longrightarrow_{S_i:o!\ldots} s'$, we remove all transitions outgoing from s' but for $s' \longrightarrow_{S_i:o?\ldots} s''$. In a case where such a second transition is not available, we also remove t.

Once the filtering rules have been applied, it may be necessary to clean the model up to ensure that all states are still both reachable from the initial state, and co-reachable from a final state (deadlocks should not be introduced by filtering):

- remove any state $s \in S$ for which there does not exist a path $I \longrightarrow^* s$
  consequently remove any transition outgoing from s
- remove any state $s \in S$ for which there does not exist a path $s \longrightarrow^* f$, with $f \in F$
  consequently remove any transition going to s

Application. The adaptor protocol obtained from the one in Figure 10 after applying rules 1-2-3 is shown in Figure 12. It contains 33 states and 34 transitions. Note that, in the filtering process, the state identifiers (not relevant for the approach) have been changed by our tool to respect continuous identifiers (*i.e.*, from 0 to 32).

Filtering rules can be applied automatically (the predefined strategy is to apply them in order 1-2-3), semi-automatically (with ad-hoc application strategies) or interacting with a service architect (designer). Filtering removes parts of the adaptor models. If deadlock-freeness is preserved, some properties of the adaptors may change. It is therefore interesting to verify the filtered adaptors, which can be done using the same techniques as presented in Section 5. In our example, we have been able to verify that the temporal properties given in Section 5.4 hold, with the exception of the safety property concerning the absence of refund without providing expenses, which is false for the adaptor in Figure 12 because of the travel scenario based on the fixed amount option. This scenario is exhibited by Evaluator as the counterexample sequence shown in Section 5.4.

6.2.2 Transforming Adaptors into BPEL Orchestrations

Filtered adaptor models can be implemented with a service orchestration language, such as BPEL. The main part

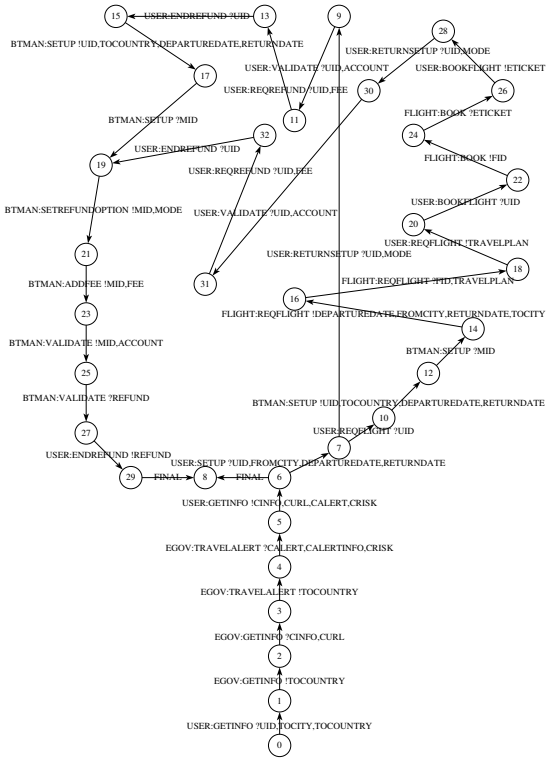

Fig. 12: eTrip – filtered version of the adaptor protocol for the first vector LTS given in Figure 7.

of the transformation is concerned with the encoding of the state and transition structure of the STS model. We rely on a specific design pattern, namely the state machine pattern. Additionally, BPEL specificities related to the orchestration environment (WSDL interfaces import and partnership definition), to communication (based on the receive, reply, invoke, and pick activities), or to internals (variables and assignments) are to be taken into account.

Partner links and variables. A partner link is created for each service, plus one for the composite itself (user). Accordingly, WSDL interfaces for all used services and for the orchestration itself (partner link user) are imported. Global variables are created for the vector variables, *e.g.*, TOCOUNTRY, and for each received/emitted message or related part depending on the variable passing scheme, *e.g.*, user_getInfoIn or user_UID, user_country, etc. 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:o!x1,...,xn transition (c not being user) followed by a c:o?y1,...,yn transition is encoded as a synchronous invoke(c, o, (x1,...,xn), (y1,...,yn)) activity, where c is the partner link, o is an operation defined in the service bound to partner link c, and xi and yj are respectively the input and the output variables (if there are several they correspond to message parts).

A user:o?x1,...,xn transition corresponds to the interaction with the environment, it is encoded as a receive(user, o, (x1,...,xn)) activity, where user is the partner link, o is an operation provided by the adaptor, and xi are input variables (or message parts). If there are several such transitions in some state, they are encoded using a pick activity with as many onMessage parts as there are reception transitions. Moreover, if the source state of the transition is a final one, then the pick activity is used and an onAlarm part is created to support a possible termination of the orchestrator, setting FINAL to true.

Finally, a user:o!y1,...,yn transition also corresponds to an interaction with the environment, and is encoded as a reply(user, o, (y1,...,yn)) activity, where user is the partner link, o is an operation provided by the adaptor, and yj are output variables (or message parts).

BPEL engines use part of the data transmitted along with incoming messages to route these messages to specific process instances. This sets up a form of communication session, called correlation set, between a process instance and its partners. Each correlation set is supported by one or several data values called its properties (it is usually a single one, *e.g.*, a user identifier, but it can be several, *e.g.*, a first name and a surname). In order to be able to retrieve in a message the parts corresponding to the properties of a correlation set, BPEL engines use correspondences (called property aliases) between correlation set properties and message data. Accordingly, we support this as follows in our transformation. All communication activities with the user partner link are related using a correlation set named USER_CS and we require that associated properties and property aliases have been defined in the user signature (in its WSDL file).

Assignments. Some adaptor variables come from vectors, while others are related to messages. To relate 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. if/elseif statements are used inside it to encode the adaptor states. The if/elseif body of a state i encodes its outgoing transition(s). This corresponds to invoke, receive (or pick), and reply activities. The STATE variable is updated accordingly to the transition of interest, *e.g.*, the encoding of a transition going to a state S_x will end up setting STATE to S_x. For the final state we only set FINAL to true.

Application. The result of the BPEL encoding of the filtered adaptor in Figure 12 is given in Figure 13. On the bottom, we present the overall BPEL process where one can see the result of the state machine pattern

encoding with a while loop and if/elseif branches for the different states in the filtered adaptor model. Note that intermediate states of invocations are not encoded since both reception and emission transitions will be encoded with a single invoke (*e.g.*, the transitions between states 1 and 3, or 3 and 5, in the zoom). On the zoom (top right part), one may see the use of a pick for state 6 with either the reception of a message from the user to set up the mission (onMessage) or the use of a timer to stop (onAlarm, set to 10 minutes by default). There is another pick in the encoding, in state 7, where the user may chose to ask for a flight or directly indicate expenses (two onMessage branches). Finally one may see on the zoom examples of message response (reply) and invocation (invoke), with related assignments before and after them.

6.3 Tool Support

The relation between real languages and the formal models used in our approach are automated by two prototype tools we have implemented. BPEL2STS transforms service interface descriptions (WSDL and (A)BPEL) into STS. STS2BPEL deals with adaptor implementation, *i.e.*, both filtering and BPEL encoding. As far as filtering rules are concerned, it supports both pre-defined strategies or designer/end-user ad-hoc ones. For the time being, once a BPEL orchestrator is obtained, it has to be deployed by hand. Our experiments have been achieved using the NetBeans IDE and the GlassFish BPEL Engine.

7 Related Work

Several surveys have recently been done on existing works which proposed solutions in the software adaptation area [1], [2], [3]. In this section, we focus on the proposals the most related to ours.

In the last ten years, most adaptation proposals focused on solving behavioral mismatches between abstract descriptions of *software components*, see for instance [4], [5], [6], [8], [47]. Brogi *et al.* (BBC) [6] presents 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 mismatches. This approach has recently been used to obtain adaptor implementations for services [48] (see below). Inverardi *et al.* (IT) [5] address the enforcement of behavioral properties out of a set of components. Starting from the specification with MSCs of the components to be assembled 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. In [8], we have proposed an automated adaptation approach that was both generative and restrictive, and supported 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 [6] 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 by use of process algebra encodings and on-the-fly generation techniques.

With the advent of *Web services* a few years ago, many people started to work on the adaptation of services. Adapting services is even more crucial than adapting components since, although the black-box assumption is sometimes criticized in the software components area [49], [50], [51], this hypothesis cannot be avoided in the Web services area, and rich interface description languages are necessary. In the following, we first present existing works which are not related to any programming platform, and second those which are.

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 (flow, gather, scatter, collapse, burst, and hide) for establishing the basic relation patterns between the message names used in the services 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 services. Their approach is similar to ours in the sense that these basic operations correspond to the different relations (1-1, 1-0, 1-n, n-m, etc.) between message names that can be defined by means of our vectors. 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. Moreover, their mappings require one to relate the messages at the behavioral level (*i.e.*, matching messages directly from the service protocol specifications), while our adaptation contracts are more abstract, since the mapping is performed at the signature level (*i.e.*, between the messages declared in the service interfaces) from which we automatically obtain an adaptor solving the mismatch at the behavioral level. Finally, their approach is not able to perform message reordering when it is required for solving the problem.

In [52], Van der Aalst *et al.* propose a solution to behavioral adaptation based on open nets, a variant of Petri nets. Their generation algorithm produces an adaptor which is obtained through several steps. First, a message transformation net, called engine, is generated from a set of message transformation rules. Then, a behavioral controller (a transition system) is synthesized for the product net of the services and the engine. This approach is supported by tools that automate the adaptor model

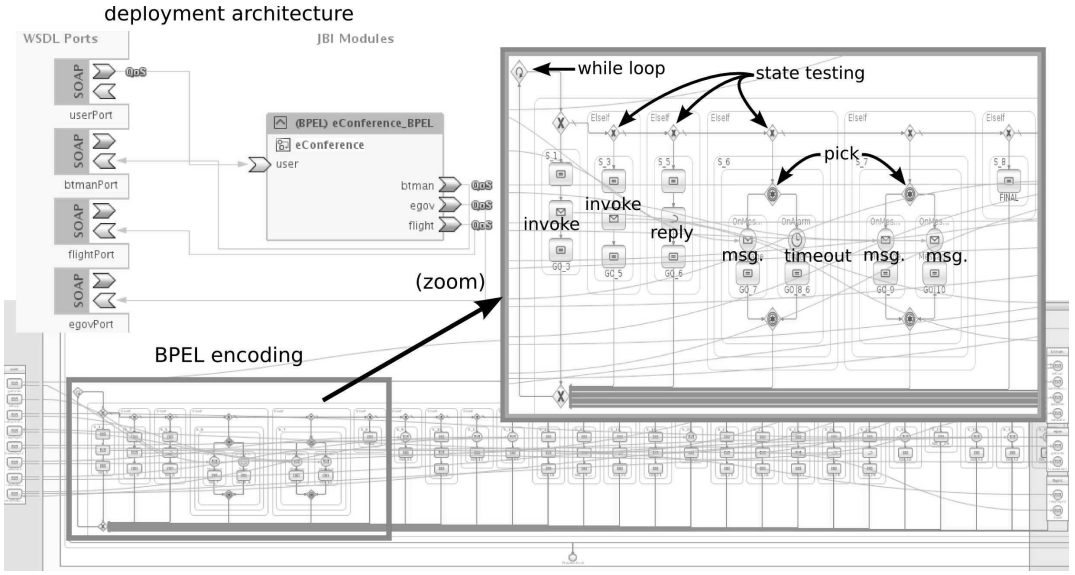

Fig. 13: eTrip – screenshot of the adaptor implemented as a BPEL orchestrator

computation. Adaptors may be implemented in BPEL using first the transformation of an adaptor model into an open net, then the transformation of this net into BPEL. The main difference with our work is that they address deadlock-freeness of adapted systems, while we may use vector LTS also to express more expressive adaptation scenarios.

In [53], the authors present some techniques to identify all mismatches when composing two services. They also classify these incompatibilities into patterns. These patterns come with some solutions to solve each kind of incompatibility. As far as adaptation is concerned, the authors rely on CEP technology which is a platform for running event-based applications that can catch event streams and trigger a specific action when these events occur. In that respect, this approach is quite close to run-time adaptation techniques such as [54], [55] which are able to dynamically compensate mismatches in a running system. However, these solutions are quite restricted because they cannot correct some subtle mismatches such as message or data reordering. In addition, this work only considers two services as input and this is a quite strong assumption, indeed many complex systems involve several services, and not only two.

Recent approaches [48], [56], [57] which focus on existing programming languages and platforms, such as BPEL or SCA services, suggest manual or at most semi-automated techniques for solving behavioral mismatches. First, [48] 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 service behavioral interfaces, and they use lock analysis techniques to detect behavioral mismatch. Motahari-Nezhad *et al.* [56] present some techniques in order to provide semi-automatic support for the identification and resolution of mismatches between Web services at their signature and protocol levels. First, the authors describe some techniques for signature matching based on XML schema matching. After applying interface matching techniques, the authors use the protocol definitions expressed using Finite State Machines to find all mismatch situations at the protocol level. While unspecified receptions are dealt with automatically, deadlock resolution is tackled through the generation of mismatch trees, which present to the developer potential execution scenarios where the services deadlock. This approach deals with some kinds of mismatch automatically, but requires user input to overcome others. The situations which can be adapted are quite limited. In particular, correspondences between operations are static, and 1-0 correspondences (operations with no match on the counterpart interface) are not supported. This work was extended in [58] to support one-to-many correspondences. Furthermore, they improve the protocol matching proposed in [56] using depth-based comparison and flooding-based techniques, similarly to the approach given in [59], in order to compute best matches between messages. Since their primary objective is to support service replacement, both [48] and [56] focus on adaptation between two services, while we are able to perform adaptation between any number of services. In [57], the authors deal with the monitoring and adaptation of BPEL services at runtime according to Quality of Services attributes (different focus compared to ours). Their approach also proposes replacement of partner services based on various strategies, either syntactic or semantic.

Software adaptation shares some objectives with controller synthesis [60], [61], which focuses on the genera-

tion of controllers with respect to a given system (called plant) designed as a finite automaton and properties to be ensured. However, both approaches are fundamentally different because controller synthesis tries to influence (when possible) the behavior of the controlled system, while adaptation works on black-box components or services, and should be non-intrusive. In addition, controller synthesis is a restrictive approach while some adaptation approaches, like ours, are also generative. Finally, adaptation can use buffering and tackles data mismatch, which controller synthesis does not.

Behavioral contracts [62], [63] provide an alternative and concise way to specify component or service behavioral interfaces. They are of particular interest for service discovery and selection (*i.e.*, selecting the best service in a registry that relates to some behavioral requirement – both services and requirements being described with contracts). In [64], Padovani presents a theory based on behavioral contracts that may be used to generate orchestrators between two services related by a subtyping (namely, sub-contract) relation. This can be used to generate an adaptor between a client of some service s and a service replacing s. An interesting feature of this approach is its expressiveness as far as behavioral descriptions are concerned, with support for recursive behaviors and buffered orchestrators. However, as far as adaptation is concerned, this work is restricted to pure behavioral adaptation since name mismatch is not supported. Further, the use for contracts of a process algebra without value-passing means adaptation related to data is also not directly supported.

A preliminary version of this work has been presented in [65]. It is extended here in several aspects: (i) an in-depth introduction and motivation to software adaptation and its application to Service-Oriented Computing, (ii) the use of a new realistic case study, (iii) a new presentation of the adaptation constraints encoded into process algebra, (iv) a detailed description of proposed adaptor verification techniques, including their extension with techniques for checking non-controllability, (v) the presentation of the rules for retrieving STS models from (A)BPEL protocols, and (vi) an updated review and comparison with related work.

8 Concluding Remarks

Software adaptation is a promising solution for composing, in a non-intrusive way, black-box services that contain incompatibilities in their interfaces. In this article, we have presented our tool-supported techniques for generating 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 approach is based on an encoding into the LOTOS process algebra, we take advantage of the CADP toolbox for LOTOS to verify the correctness of the contract. We have also shown with BPEL how our adaptors can be implemented.

A first perspective of our work is to develop an on-the-fly solution to the non-controllability issue. Our idea is not only to detect this issue, as presented in Section 5, but also to be able to deal with it directly while computing adaptor models, by pruning the branches of these adaptors that cannot be controlled. A second perspective is related to service selection. In our work we suppose that a set of services have previously been selected before we apply our adaptation algorithms. Defining an adaptability notion, and integrating it in service discovery, would foster efficiency of the overall service composition process by selecting services that present a better potential for being adapted. Our last perspective is related to adaptation contracts. We have seen in Section 3 that adaptation contracts based on vectors and vector LTSs enable one not only to solve mismatches between services, but also to enforce some constraints over service coordination. Extending our contract notation to take into account temporal properties, such as those presented in Section 5, would increase the expressiveness of our adaptation algorithms that would be able to enforce these temporal properties by construction. This extension would also allow us to use our adaptation techniques in other application areas, such as controller synthesis [60], [61] where, to the best of our knowledge, no tool support exists to generate controllers using on-the-fly exploration techniques.

Acknowledgments

The authors would like to thank the TSE Associate Editor and the anonymous reviewers for their very useful comments during the review process, José Antonio Martín for his participation in the implementation of the BPEL2STS and STS2BPEL tools, and Javier Cámara for his help with the Acide tool. We are also grateful to Christine McKinty, who proofread a former version of this manuscript. This work has been partially supported by projects "PERvasive Service cOmposition" (PERSO, ANR-07-JCJC-0155-01), and "Personal Information Management through Internet" (PIMI, ANR-10-VERS-0014-03) funded by the French National Agency for Research.

References

[1] 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*, ser. LNCS, vol. 3938. Springer-Verlag, 2006, pp. 193–215.

[2] C. Canal, J. M. Murillo, and P. Poizat, "Software Adaptation," *L'Objet*, vol. 12, no. 1, pp. 9–31, 2006.

[3] R. Seguel, R. Eshuis, and P. Grefen, "An Overview on Protocol Adaptors for Service Component Integration," Eindhoven University of Technology, BETA Working Paper Series WP 277, 2009.

[4] R. H. Reussner, "Automatic Component Protocol Adaptation with the CoConut/J Tool Suite," *Future Generation Computer Systems*, vol. 19, no. 1, pp. 627–639, 2003.

[5] M. Tivoli and P. Inverardi, "Failure-free Coordinators Synthesis for Component-based Architectures," *Science of Computer Programming*, vol. 71, no. 3, pp. 181–212, 2008.

[6] A. Bracciali, A. Brogi, and C. Canal, "A Formal Approach to Component Adaptation," *Journal of Systems and Software*, vol. 74, no. 1, pp. 45–54, 2005.

[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*, ser. LNCS, vol. 4102. Springer-Verlag, 2006, pp. 65–80.

[8] C. Canal, P. Poizat, and G. Salaün, "Model-Based Adaptation of Behavioural Mismatching Components," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 546–563, 2008.

[9] A. Marconi and M. Pistore, "Synthesis and Composition of Web Services," in *Proc. of Formal Methods for Web Services, 9th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM'2009)*, ser. Lecture Notes in Computer Science, vol. 5569, 2009, pp. 89–157.

[10] P. Bertoli, M. Pistore, and P. Traverso, "Automated composition of Web services via planning in asynchronous domains," *Artificial Intelligence*, no. 174, pp. 316–361, 2010.

[11] F. Plasil and S. Visnovsky, "Behavior Protocols for Software Components," *IEEE Transactions on Software Engineering*, vol. 28, no. 11, pp. 1056–1076, 2002.

[12] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes," in *Proc. of TACAS'11*, ser. LNCS, vol. 6605. Springer-Verlag, 2011, pp. 372–387.

[13] J. A. Martín and E. Pimentel, "Automatic Generation of Adaptation Contracts," in *Proc. of FOCLASA'08*, ser. ENTCS, vol. 229, no. 2, July 2009, pp. 115–131.

[14] J. Cámara, G. Salaün, C. Canal, and M. Ouederni, "Interactive Specification and Verification of Behavioural Adaptation Contracts," in *Proc. of QSIC'09*. IEEE Computer Society, 2009, pp. 65–75.

[15] M. H. ter Beek, A. Bucchiarone, and S. Gnesi, "Formal Methods for Service Composition," *Annals of Mathematics, Computing & Teleinformatics*, vol. 1, no. 5, pp. 1–10, 2007.

[16] A. Bucchiarone, H. Melgratti, and F. Severoni, "Testing Service Composition," in *Proc. of ASSE'07*, 2007.

[17] X. Fu, T. Bultan, and J. Su, "Analysis of Interacting BPEL Web Services," in *Proc. of WWW'04*. ACM Press, 2004, pp. 621–630.

[18] G. Salaün, L. Bordeaux, and M. Schaerf, "Describing and Reasoning on Web Services using Process Algebra," *International Journal of Business Process Integration and Management*, vol. 1, no. 2, pp. 116–128, 2006.

[19] 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*. ACM Press, 2006, pp. 771–774.

[20] 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*, ser. ENTCS, vol. 215. Elsevier, 2007, pp. 39–55.

[21] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli, "Automatic Synthesis of Behavior Protocols for Composable Web-Services," in *Proc. of ESEC/SIGSOFT FSE'09*. ACM, 2009, pp. 141–150.

[22] M. Hennessy and H. Lin, "Symbolic Bisimulations," *Theor. Comput. Sci.*, vol. 138, no. 2, pp. 353–389, 1995.

[23] P. Poizat and J.-C. Royer, "A Formal Architectural Description Language based on Symbolic Transition Systems and Modal Logic," *Journal of Universal Computer Science*, vol. 12, no. 12, pp. 1741–1782, 2006.

[24] C. Attiogbé, P. Poizat, and G. Salaün, "A Formal and Tool-Equipped Approach for the Integration of State Diagrams and Formal Datatypes," *IEEE Transactions on Software Engineering*, vol. 33, no. 3, pp. 157–170, 2007.

[25] F. Durán, M. Ouederni, and G. Salaün, "Checking Protocol Compatibility using Maude," in *Proc. of FOCLASA'09*, ser. ENTCS, vol. 255. Elsevier, 2009, pp. 65–81.

[26] L. Bentakouk, P. Poizat, and F. Zaïdi, "A Formal Framework for Service Orchestration Testing based on Symbolic Transition Systems," in *Proc. of TESTCOM'09*, ser. LNCS, vol. 5826. Springer, 2009, pp. 16–32.

[27] A. Zisman, G. Spanoudakis, and J. Dooley, "A Framework for Dynamic Service Discovery," in *Proc. of ASE'08*. ACM Press, 2008, pp. 158–167.

[28] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice Hall, 1985.

[29] ISO/IEC, "LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour," ISO, International Standard 8807, 1989.

[30] T. Bolognesi and E. Brinksma, "Introduction to the ISO Specification Language LOTOS," *Computer Networks and ISDN Systems*, vol. 14, no. 1, pp. 25–59, 1988.

[31] C. Canal, P. Poizat, and G. Salaün, "Synchronizing Behavioural Mismatch in Software Composition," in *Proc. of FMOODS'06*, ser. LNCS, vol. 4037. Springer-Verlag, 2006, pp. 63–77.

[32] M. J. Fischer and R. E. Ladner, "Propositional dynamic logic of regular programs," *Journal of Computer and System Sciences*, vol. 18, no. 2, pp. 194–211, September 1979.

[33] R. Mateescu and M. Sighireanu, "Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus," *Science of Computer Programming*, vol. 46, no. 3, pp. 255–281, 2003.

[34] H. R. Andersen, "Model checking and boolean graphs," *Theoretical Computer Science*, vol. 126, no. 1, pp. 3–30, Apr. 1994.

[35] D. Kozen, "Results on the propositional μ-calculus," *Theoretical Computer Science*, vol. 27, pp. 333–354, 1983.

[36] R. Cleaveland and B. Steffen, "A linear-time model-checking algorithm for the alternation-free modal mu-calculus," *Formal Methods in System Design*, vol. 2, no. 2, pp. 121–147, April 1993.

[37] R. Mateescu, "Caesar_solve: A generic library for on-the-fly resolution of alternation-free boolean equation systems," *Springer International Journal on Software Tools for Technology Transfer (STTT)*, vol. 8, no. 1, pp. 37–56, February 2006, full version available as INRIA Research Report RR-5948, July 2006.

[38] ——, "On-the-fly state space reductions for weak equivalences," in *Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems FMICS'05 (Lisbon, Portugal)*, T. Margaria and M. Massink, Eds., ERCIM. ACM Computer Society Press, September 2005, pp. 80–89.

[39] H. Garavel, "OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing," in *Proc. of TACAS'98*, ser. LNCS, vol. 1384. Springer-Verlag, 1998, pp. 68–84.

[40] H. Garavel and F. Lang, "SVL: A Scripting Language for Compositional Verification," in *Proc. FORTE'01*, IFIP. Kluwer Academic Publishers, 2001, pp. 377–392.

[41] M. Gowri Nanda, S. Chandra, and V. Sarkar, "Decentralizing Execution of Composite Web Services," in *Proc. of OOPSLA'04*, 2004.

[42] G. Chafle, S. Chandra, V. Mann, and M. Gowri Nanda, "Orchestrating Composite Web Services Under Data Flow Constraints," in *Proc. of ICWS'05*, 2005.

[43] M. Autili, L. Mostarda, A. Navarra, and M. Tivoli, "Synthesis of decentralized and concurrent adaptors for correctly assembling distributed component-based systems," *Journal of Systems and Software*, vol. 81, no. 12, pp. 2210–2236, 2008.

[44] G. Salaün, "Generation of Service Wrapper Protocols from Choreography Specifications," in *Proc. of SEFM'2008*, 2008.

[45] OASIS, "Web Services Business Process Execution Language (WS-BPEL) Version 2.0," OASIS, Tech. Rep., April 2007.

[46] M. Bozkurt, M. Harman, and Y. Hassoun, "Testing Web Services: A Survey," King's College London, Tech. Rep., 2010.

[47] D. M. Yellin and R. E. Strom, "Protocol Specifications and Components Adaptors," *ACM Trans. on Programming Languages and Systems*, vol. 19, no. 2, pp. 292–333, 1997.

[48] A. Brogi and R. Popescu, "Automated Generation of BPEL Adapters," in *Proc. of ICSOC'06*, ser. LNCS, vol. 4294. Springer-Verlag, 2006, pp. 27–39.

[49] M. Buchi and W. Weck, "The Greybox Approach: When Blackbox Specifications Hide Too Much," Turku Center for Computer Science, Tech. Rep. 297, 1999.

[50] J. Henriksson, F. Heidenreich, J. Johannes, S. Zschaler, and U. Aßmann, "How Dark Should a Component Black-box Be? The Reuseware Answer," in *Proc. of the 12th International Workshop on Component-Oriented Programming (WCOP'07)*, 2007.

[51] F. Puntigam, "Black & White, Never Grey: On Interfaces, Synchronization, Pragmatics, and Responsibilities," in *Proc. of the 12th International Workshop on Component-Oriented Programming (WCOP'07)*, 2007.

[52] W. M. P. van der Aalst, A. J. Mooij, C. Stahl, and K. Wolf, "Service Interaction: Patterns, Formalization, and Analysis," in *Proc. of Formal Methods for Web Services, 9th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM'2009)*, ser. Lecture Notes in Computer Science, vol. 5569, 2009, pp. 42–88.

[53] Y. Taher, A. Aït-Bachir, M.-C. Fauvet, and D. Benslimane, "Diagnosing Incompatibilities in Web Service Interactions for Automatic Generation of Adapters," in *Proc. of AINA'09*. IEEE Computer Society, 2009, pp. 652–659.
[54] J. Cámara, G. Salaün, and C. Canal, "Composition and Run-time Adaptation of Mismatching Behavioural Interfaces," *Journal of Universal Computer Science*, vol. 14, no. 13, pp. 2182–2211, 2008.
[55] K. Wang, M. Dumas, C. Ouyang, and J. Vayssière, "The Service Adaptation Machine," in *Proc. of ECOWS'08*. IEEE Computer Society, 2008, pp. 145–154.
[56] H. R. Motahari-Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati, "Semi-Automated Adaptation of Service Interactions," in *Proc. of WWW'07*, 2007, pp. 993–1002.
[57] O. Moser, F. Rosenberg, and S. Dustdar, "Non-Intrusive Monitoring and Adaptation for WS-BPEL," in *Proc. of WWW'08*, 2008, pp. 815–824.
[58] H. R. M. Nezhad, G. Y. Xu, and B. Benatallah, "Protocol-Aware Matching of Web Service Interfaces for Adapter Development," in *Proc. of WWW'10*. ACM, 2010, pp. 731–740.
[59] S. Nejati, M. Sabetzadeh, M. Chechik, S. M. Easterbrook, and P. Zave, "Matching and Merging of Statecharts Specifications," in *Proc. of ICSE'07*. IEEE Computer Society, 2007, pp. 54–64.
[60] W. M. Wonham and P. J. Ramadge, "On the Supremal Controllable Sublanguage of a Given Language," *SIAM Journal on Control and Optimization*, vol. 25, no. 3, pp. 637–659, 1987.
[61] P. J. G. Ramadge and W. M. Wonham, "The Control of Discrete Event Systems," *Proc. of the IEEE*, vol. 77, no. 1, pp. 81–98, 1989.
[62] A. Vallecillo, V. T. Vasconcelos, and A. Ravara, "Typing the Behavior of Objects and Component Using Session Types," *Electr. Notes Theor. Comput. Sci.*, vol. 68, no. 3, 2003.
[63] A. Brogi, C. Canal, and E. Pimentel, "Behavioural Types and Component Adaptation," in *Proc. of AMAST'04*, ser. LNCS, vol. 3116. Springer, 2004, pp. 42–56.
[64] L. Padovani, "Contract-Based Discovery and Adaptation of Web Services," in *Proc. of SFM'09*, ser. LNCS, vol. 5569. Springer, 2009, pp. 213–260.
[65] 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*, ser. LNCS, vol. 5364. Springer-Verlag, 2008, pp. 84–99.