Interactive Specification and Verification of Behavioural Adaptation Contracts

Javier Cámara, Gwen Salaün, Carlos Canal, and Meriem Ouederni
Department of Computer Science, University of Málaga, Spain
{jcamara,salaun,canal,meriem}@lcc.uma.es

Abstract

Adaptation is a crucial issue when building new applications by reusing existing software services which were not initially designed to interoperate with each other. Adaptation contracts *describe composition constraints and adaptation requirements among these services. The writing of this specification by a designer is a difficult and error-prone task, especially when service protocol needs to be considered and service functionality accessed through behavioural interfaces. In this paper, we propose an interactive approach to support the contract design process, and more specifically: (i) a graphical notation to define port bindings, and an interface similarity measure to compare protocols and suggest some port connections to the designer, (ii) compositional and hierarchical techniques to facilitate the specification of adaptation contracts by building them incrementally, (iii) validation and verification techniques to check that the contract will make the involved services work correctly and as expected by the designer. Our approach is fully supported by a prototype tool we have implemented.*

1 Introduction

Services can be accessed and used to fulfill basic requirements, or can be composed with other services in order to build bigger systems which aim at working out complex tasks. They must be equipped with rich interfaces to ease their reuse and enable their automatic composition. Interface description languages distinguish several interoperability levels (*i.e.,* signature, protocol, quality of service, and semantics). Composition of services is seldom achieved seamlessly because mismatch may occur at the different interoperability levels and must be solved. *Software adaptation* [22, 7] is a recent discipline which aims at generating, as automatically as possible, adaptors used to solve mismatches among services in a non-intrusive way. So far, most adaptation approaches have assumed interfaces described by signatures (operation names and types) and behaviours

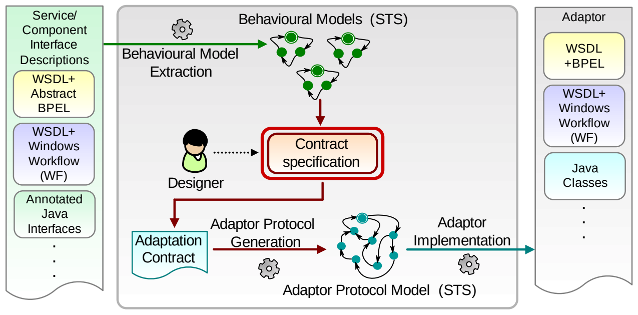

Service/ Component Interface Descriptions
WSDL+ Abstract BPEL
WSDL+ Windows Workflow (WF)
Annotated Java Interfaces
Behavioural Models (STS)
Behavioural Model Extraction
Designer
Contract specification
Adaptation Contract
Adaptor Protocol Generation
Adaptor Implementation
Adaptor Protocol Model (STS)
Adaptor
WSDL +BPEL
WSDL+ Windows Workflow (WF)
Java Classes

Figure 1. Generative adaptation process

(interaction protocols). Describing protocols in service interfaces is essential because erroneous executions or deadlock situations may occur if the designer does not consider them while building composite services.

A first class of existing works dedicated to model-based behavioural adaptation is referred to as *restrictive approaches*, see for instance [5, 3, 17]. They favour full automation of the process, and try to solve interoperability issues by pruning the behaviours that may lead to mismatch, thus restricting the functionality of the services involved. These techniques are limited since they are not able to fix subtle incompatibilities between service protocols by remembering and reordering events and data when necessary. A second class of solution is referred to as *generative approaches*, see for instance [4, 9, 7]. They avoid the arbitrary restriction of service behaviour, and support the specification of advanced adaptation scenarios. Generative approaches build adaptors automatically from an abstract specification, namely an *adaptation contract*, of how mismatch cases can be solved.

Although generative approaches result in a more general and satisfactory solution while composing and adapting services, writing the contract is a difficult and error-prone task. Incorrect correspondences between operations in service interfaces, or syntactic mistakes are common, especially in cases where the contract has to be specified using cumbersome textual notations [4]. Contracts should also describe in an abstract way the different execution scenarios of the system, which may not be easily envisioned by the designer. Moreover, contracts must avoid undesirable system behaviour such as deadlocks or incorrect order of the messages exchanged, and this is difficult when protocols are taken into account in interface descriptions.

In this paper, we advocate for interactive techniques to help the designer in the adaptation contract specification process (see Figure 1 for an overview of the whole adaptation process). To this purpose, we first propose a graphical notation to visualize service protocols and define port bindings. Our notation also integrates a measure of similarity between protocols that the designer can use to detect parts of service protocols which turn out to be similar, and then connect them. Second, we formalise compositional and hierarchical techniques in order to build the system incrementally and therefore simplify the process. Last, to check if the behaviour of the system complies with the designer's intentions, we propose validation and verification techniques which allow to simulate visually the execution of the system step-by-step, and find out which parts of the system lead to erroneous behaviour (deadlock, infinite loops, safety and liveness properties). Our approach is fully implemented in a prototype tool, ACIDE, which has been applied to many case studies.

The rest of this paper is structured as follows: Section 2 presents our service model. Section 3 introduces our contract specification language and overviews adaptation techniques that can be used to generate adaptor protocols from such contracts. Section 4 presents a compositional and hierarchical approach to ease the specification of adaptation contracts. Section 5 describes our graphical environment that supports contract design, as well as our similarity measure between service protocols. In Section 6, we propose verification techniques to check contracts. Section 7 introduces our prototype tool (ACIDE), and some experimental results. Finally, Section 8 compares our approach with related works, and Section 9 concludes the paper.

2 Interface Model

We assume that service interfaces are equipped both with a signature (set of required and provided operations), and a protocol represented by a *Symbolic Transition System* (STS). Communication between services is represented using *events* relative to the emission and reception of messages corresponding to operation calls. Events may come with a list of parameters whose types respect the operation signatures. In our model, a *label* is either the internal action τ or a tuple (M, D, PL) where M is the message name, D stands for the communication direction (! for emission, and ? for reception), and PL is either a list of data terms if the message corresponds to an emission, or a list of variables if the message is a reception.

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

Our STS are a simplified version of STG (Symbolic Transition Graphs) introduced in [12]. The only difference is that guards are abstracted as τ transitions, that correspond to internal (unobservable) activities of the service. The operational semantics of an STS ($\rightarrow_b$) is defined with three rules (Fig. 2) formalising the meaning of each kind of labels, namely τ (TAU), emissions (EM), and receptions (REC). Each couple $\langle s, E\rangle$ represents an active state $s \in S$ and a data environment E. A data environment is a set of couples $\langle x, v\rangle$ where x is a variable and v a ground value. We use a function *type* which returns the type of a variable, and we define the environment update "$\oslash$", and the evaluation function ev as follows:

$$E \oslash \langle x, v\rangle \triangleq E(x) = v$$
$$ev(E, x) \triangleq E(x)$$
$$ev(E, f(v_1, \ldots, v_n)) \triangleq f(ev(E, v_1), \ldots, ev(E, v_n))$$

The operational semantics of n STSs ($\rightarrow_c$) is defined with one rule (COM, see Fig. 2) that formalises a synchronous communication between two services. Value-passing and variable substitutions rely on a late binding semantics [16], and $\{as_1, \ldots, as_n\}$ is a set of couples $\langle s_i, E_i\rangle$.

The STS formal model has been chosen because it is simple, graphical, and it can be easily derived from existing implementation languages (see for instance [11, 21, 10, 8] where such abstractions for Web services were used for verification, composition or adaptation purposes). Due to page limitation, in the rest of the paper we will describe service interfaces only with their STSs. Signatures will be left implicit, yet they can be inferred from the typing of arguments (made explicit here) in STS labels.

Example. In this paper, we use as running example an online medical management system which handles patient appointments. As it can be observed in Figure 3, we reuse three services in this new system, and we give an example of user requirements implemented in a client. This Client can first log on to a server by sending respectively his/her user name (user!) and password (password!). Then, depending on his/her preferences (internal choice specified with τ transitions in the client protocol), the client can ask for an appointment either with a general practitioner (reqDoc!) or a specialist doctor (reqSpec!), and then receive an appointment identifier. Service Serverdoc first receives the client user name and password (login?). Next, this service receives a request for an appointment with a general

$$\frac{(s \xrightarrow{\tau} s') \in T}{\langle s, E\rangle \xrightarrow{\tau}_b \langle s', E\rangle} \qquad \text{(TAU)}$$

$$\frac{(s \xrightarrow{a!v} s') \in T \quad v' = ev(v, E)}{\langle s, E\rangle \xrightarrow{a!v'}_b \langle s', E\rangle} \quad \text{(EM)} \qquad \frac{(s \xrightarrow{a?x} s') \in T}{\langle s, E\rangle \xrightarrow{a?x}_b \langle s', E\rangle} \quad \text{(REC)}$$

$$\frac{\begin{array}{c} i, j \in \{1..n\} \quad i \neq j \\ \langle s_i, E_i\rangle \xrightarrow{a!v}_b \langle s'_i, E_i\rangle \quad \langle s_j, E_j\rangle \xrightarrow{a?x}_b \langle s'_j, E_j\rangle \\ type(x) = type(v) \quad E'_j = E_j \oslash \langle x, v\rangle \end{array}}{\{as_1, .., \langle s_i, E_i\rangle, .., \langle s_j, E_j\rangle, .., as_n\} \xrightarrow{a!v}_c \{as_1, .., \langle s'_i, E_i\rangle, .., \langle s'_j, E'_j\rangle, .., as_n\}} \qquad \text{(COM)}$$

Figure 2. Operational Semantics of STS

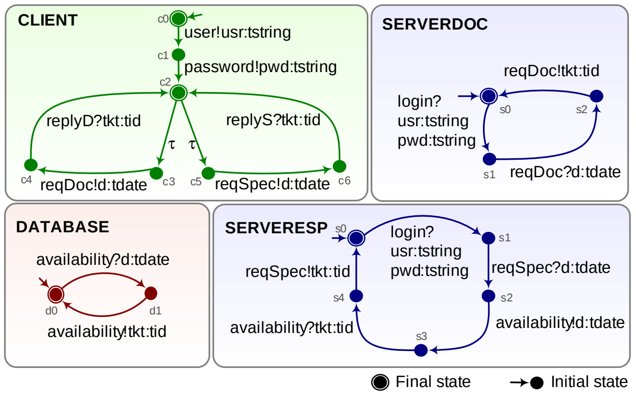


Figure 3. Client, server and database behavioural interfaces

practitioner (reqDoc?) and replies (reqDoc!). Service Serveresp first receives the client user name and password (login?), followed by a request for an appointment with a specialist doctor (reqSpec?). After checking doctor availability for the given date, an appointment identifier is returned (reqSpec!) to the client. Service Database can receive and reply requests for a specialist doctor's availability for a given date (availability?/availability!).

We intend to compose these services into a working system where the client can request an appointment with a general practitioner, and optionally request an appointment with a specialist doctor, provided that there is a previous appointment with the general practitioner (*i.e.*, the client cannot directly schedule an appointment with the specialist).

3 Contract Specification and Adaptor Generation

In this section, we first present our contract specification language that specifies how to work out mismatch situations, and briefly discuss the generation of adaptor protocols from such descriptions.

3.1 Contract Specification Language

While building a new system by reusing existing services, behavioural interfaces do not always fit one another, and these interoperability issues have to be faced and worked out. Mismatches may be caused by different message names, a message without counterpart (or with several ones) in the partner, message arguments differently ordered or distributed across different messages, etc. The presence of mismatch results in a deadlocking execution of several services [3, 7]. This is easily detected by exploring all the interactions of the set of service STSs obtained by application of the rule COM presented in Section 2.

Adaptors are automatically built from an abstract description, called *adaptation contract*, of how mismatch situations can be solved. In this paper, we use *vectors* and a *vector LTS* as adaptation contract specification language [19]. A vector contains a set of events (message, direction, set of parameters). Each event is executed by one service and the overall result corresponds to an interaction between all the involved services. Vectors express correspondences between messages, like bindings between ports, or connectors in architectural descriptions. In this paper, we consider a binary communication model, therefore vectors are always reduced to one event (when a service evolves independently) or two (when services communicate). Furthermore, variables are used as placeholders in message parameters. The same variable names appearing in different labels (possibly in different vectors) relates sent and received arguments in the messages. Finally, since we want our adaptation process to be hierarchical and compositional, a vector may be observable (prefix o) or not (prefix c). Such vectors can also be referred, respectively, as open and closed (this issue is discussed in further detail in Section 4).

Definition 2 ((Compositional) Vector) *A (compositional)* *vector v for a set of service STSs $(A_i, S_i, I_i, F_i, T_i), i \in \{1, \ldots, n\}$ is an element of $\{\{o, c\} \times A_j\} \cup \{\{c\} \times A_j \times A_k\}$ with $j, k \in \{1, \ldots, n\}$. Such a vector is noted $o : \langle s_j : l \rangle$, $c : \langle s_j : l \rangle$, or $c : \langle s_j : l, s_k : l' \rangle$ where s_j, s_k are service identifiers, and l, l' are labels on the alphabets of services $A_j, A_k, j \neq k$ where message parameters are substituted by placeholders relating the arguments.*

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

Definition 3 (Adaptation Contract) *An adaptation contract for a set of services STS_i, $i \in \{1, .., n\}$, is a couple (V, LTS_v) where V is a set of vectors for services STS_i, and LTS_v is a vector LTS.*

Example. Figure 4 displays the set of vectors used to solve mismatch among our interfaces. For illustration purposes, we focus on the initial part of the composition, where we want to connect the general practitioner server (Serverdoc) with the client, and make authentication work correctly. For this, we need two vectors, respectively v_{user} and $v_{NloginD}$, in which we solve existing mismatches by relating different message names (login and password), specifying the independent evolution of user!, and connecting correctly exchanged data parameters using placeholders U and P (please refer to Figure 5 to see how placeholders connect parameters). The rest of the vectors in the contract work in a similar fashion, relating the remaining parts of the interfaces.

Regarding the specification of additional constraints on the composition, we can observe in the bottom part of Figure 4 that the Vector LTS for the contract constrains the interaction of the Client, Serverdoc, and Serveresp interfaces by imposing the request for an appointment with a general practitioner (v_{NreqD}) always before the possible request of an appointment with a specialist doctor (v_{NreqS}). This is achieved by excluding v_{NreqS} from the possible transitions in state 0, and including the transition $(0, v_{NreqD}, 1)$. It is worth observing that by default, all vectors available in the contract (V) are executable in both states of the VLTS, and only specific vectors are removed in order to constrain the composition. Building the VLTS in such an abstract way simplifies its specification since transitions for all vectors do not have to be specified one by one.

$$
\begin{aligned}
V = \{ v_{user} &= c\!:\langle c\!:\mathsf{user!U}\rangle,\\
v_{NloginD} &= c\!:\langle sd\!:\mathsf{login?U,P};\, c\!:\mathsf{password!P}\rangle,\\
v_{NreqD} &= c\!:\langle c\!:\mathsf{reqDoc!D};\, sd\!:\mathsf{reqDoc?D}\rangle,\\
v_{NrespD} &= c\!:\langle c\!:\mathsf{replyD?T};\, sd\!:\mathsf{reqDoc!T}\rangle,\\
v_{NloginS} &= c\!:\langle se\!:\mathsf{login?U,P}\rangle,\\
v_{NreqS} &= c\!:\langle c\!:\mathsf{reqSpec!D2};\, se\!:\mathsf{reqSpec?D2}\rangle,\\
v_{NrespS} &= c\!:\langle c\!:\mathsf{replyS?T2};\, se\!:\mathsf{reqSpec!T2}\rangle,\\
v_{availReq} &= c\!:\langle se\!:\mathsf{availability!D3};\, d\!:\mathsf{availability?D3}\rangle,\\
v_{availResp} &= c\!:\langle se\!:\mathsf{availability?T3};\, d\!:\mathsf{availability!T3}\rangle \; \}
\end{aligned}
$$

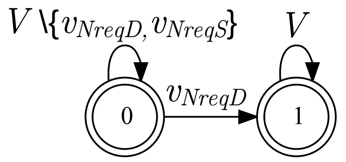


Figure 4. Adaptation contract for our example: vectors (top) and vector LTS (bottom)

3.2 Generation of Adaptor Protocols

Being given a set of service interfaces, and an adaptation contract, an adaptor protocol can be generated using techniques presented in [14]. An adaptor is a third-party component that is in charge of coordinating the services involved in the system with respect to the set of constraints defined in the contract. Consequently, all the services communicate through the adaptor, and this one is able to compensate mismatches by making required connections as specified in the contract. All protocols (adaptor and services) interact *wrt.* the rule COM presented in Fig. 2.

From adaptor protocols, either a central adaptor can be implemented, or several service wrappers can be generated to distribute the adaptation. In the former case, the implementation of executable adaptors from adaptor protocols can be achieved for instance using Pi4SOA technologies [1], or techniques presented in [14] and [8] for BPEL and Windows Workflow Foundation, respectively. In the latter case, each wrapper constrains the functionality of its service to make it respect the adaptation contract specification [20].

Example. Figure 5 shows a small portion of the adaptor protocol generated from the two vectors $v_{user} = c : \langle c : \mathsf{user!U}\rangle$ and $v_{NloginD} = c\!:\langle sd\!:\mathsf{login?U,P};\, c\!:\mathsf{password!P}\rangle$ presented above. This makes service Serverdoc and the Client interact correctly. We emphasize that the adaptor synchronizes with the services using the same name of messages but the reversed directions, *e.g.,* communication between login? in Serverdoc and login! in the adaptor. Furthermore, the adaptor always starts a set of interactions formalised in a vector with the receptions user? or pass-

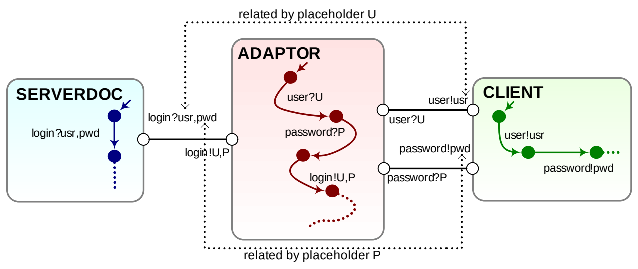


Figure 5. Example of adaptation for authentication mismatches

word? (which correspond to emissions on service interfaces), and next handles the emissions (login!).

The full adaptor protocol for our example contains 33 states and 42 transitions. It is worth observing that this adaptor has a moderate size and complexity since constraining the composition using a VLTS helps also to reduce the size of the final adaptor by imposing sequentiality on the different actions, thus reducing interleaving. In order to illustrate this fact, we can mention that if the composition of the different services in our example had not been constrained by the vector LTS, the adaptor generated using the same set of bindings without the VLTS contains 132 states and 199 transitions.

4 Hierarchical Service Composition and Adaptation

Real scenarios for service reuse and adaptation often involve several interacting services. This increases the complexity of adaptation, hindering the task of the developer. In this section, we present a *divide-and-conquer* approach that simplifies the adaptation process by building contracts incrementally. This approach is used as foundation for the graphical notation for service hierarchy and contracts presented in Section 5. Hence, in addition to being able to specify the system incrementally, the complexity of the approach described in this section is hidden to the designer since contracts and hierarchy are automatically obtained from their graphical description in our approach.

In particular, our incremental approach is based on the notion of *composite service*, which corresponds to a hierarchy of connected services. By encapsulating interactions through composite hierarchical services, the developer can focus on the construction of a contract for a particular adaptation sub-problem at a time. This encapsulation has important advantages in terms of design, development and debugging. In particular, service composites may be independently developed, tested, and modularly replaced by new elements as requirements change.

Definition 4 (Composite Service) *A composite service is a couple (SI, C) where:*

- *SI is a set of (composite or basic) service interfaces (i.e., an Id-indexed set of STSs $S_i, i \in Id$).*
- *$C = (V = V_{int} \cup V_{ext}, LTS_v)$ is an adaptation contract for the set of services in SI:*
  - *V_{int} is a set of vectors of the form $c : \langle l_l, l_r \rangle$. It represents internal bindings between the composite sub-services.*
  - *V_{ext} is a set of vectors of the form $o : \langle l \rangle$. It represents ports on the composite subservices which remain open to the environment and therefore are exposed through the composite public interface.*
  - *LTS_v is a vector LTS with its alphabet defined in V.*

Example. In our online medical management system, services Serverdoc, Serveresp, and Database are bundled within a composite Service, which interacts with the Client (Figure 6, left). In the remainder of this paper, we will informally refer in the context of our example to the scope of the Service composite as *bottom level* of the hierarchy, and the global scope of the system containing the Client and the Service as *top level*:

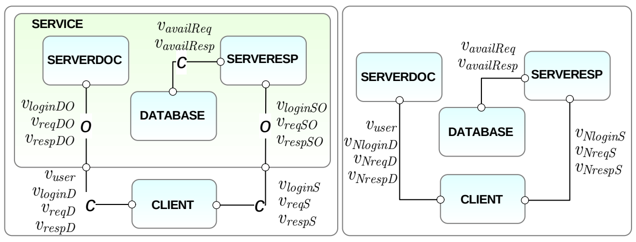


Figure 6. Service hierarchy and bindings (left) and flattened structure and bindings (right)

The bottom level contract bindings (V_{bot}, internal to the Service composite interface) allow the interaction of Serveresp and Database services (through closed vectors $v_{availReq}, v_{availResp}$), and enable us to export the rest of the ports in Serverdoc and Serveresp for external interaction with the client (open vectors). At the top level, we define the interaction of the Client with the Service composite interface. It is worth observing that the top level of a hierarchy consists of an implicit composite which contains all the interfaces on the global scope of the system and a contract relating them where all bindings are represented by closed vectors, since no ports have to be exported to an upper level. ■

Expressing hierarchical relationships among interfaces in composites is not enough to achieve composability. Particularly, if we want to replace a part of a service hierarchy (composite service) by a black-box service (thus making its implementation transparent to the rest of the system), we must provide:

1. An internal implementation for the composite service. This is obtained by generating an adaptor from c-vectors using the techniques referenced in Section 3.2. Adding this adaptor enables the involved services to interoperate while leaving ports corresponding to o-vectors open to the environment.

2. A behavioural interface for the composite service. An STS behavioural interface can be obtained for a composite service by generating the interleaving of the parts of service protocols in SI where labels correspond to open ports (those ports contained in o-vectors).

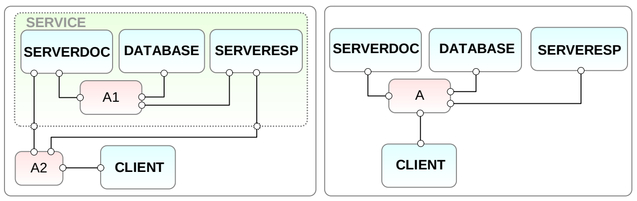


Figure 7. Alternative architectures

As an alternative to generating the implementation of composite services and composing them incrementally with the rest of the system, in some cases it is interesting to generate a centralized adaptor for a service hierarchy, since this reduces the number of adaptors (then components) in the system (Figure 7, right). In order to enable the generation of centralized adaptors, we propose an algorithm to automatically merge all the partial contracts at different levels of a service hierarchy, returning a single or *flat* adaptation contract which involves all the interacting services in the hierarchy. A hybrid approach can also be taken by applying the *flattening* process to a restricted part of a service hierarchy, reducing the overall number of adaptors in the system without compromising parallelism in parts where its preservation must be enforced.

Example. Figure 7 shows two alternative system architectures: (left) an adaptor which leaves o-vectors open to the environment (A1) is first generated for service composite Service, and another adaptor (A2) is generated in a second step to enable interoperability between Service and Client and; (right) a centralized adaptor enables the interaction of all the services after applying contract flattening. ■

The obtention of a flat adaptation contract is achieved by recursively merging contracts of adjacent levels n and $n+1$ in the service hierarchy (Algorithm 1). This contract merging process implements a depth-first traversal, since the contracts inside of any particular sub-composite of the hierarchy must be merged before proceeding to an upper level. The algorithm returns a single adaptation contract involving all the services in the hierarchy.

Algorithm 1 *flat_contract*

Returns a single contract for a composite service.
inputs Composite service $CI = (SI, C)$
output Flat adaptation contract FC

```
1: $FC = C$
2: for all $i \in SI$ do
3:     if $is\_composite(i)$ then
4:         $FC = merge\_contracts(flat\_contract(i), FC)$
5:     end if
6: end for
7: return $FC$
```

Function $is\_composite(i)$ returns $True$ if i is a composite service. Function $merge\_contracts$ merges two contracts C_{int} and C_{ext} of adjacent levels in the hierarchy, returning a single contract C for both levels:

$$merge\_contracts(C_{int} = (V_{int}, LTS_{vi}), C_{ext} = (V_{ext}, LTS_{ve})) = \\ (merge\_vectors(V_{int}, V_{ext}), free\_product(LTS_{vi}, LTS_{ve}))$$

Specifically, two contracts are merged by:

1. Merging the sets of vectors in the two contracts of levels n and $n+1$ in the hierarchy (Algorithm 2). This algorithm first adds to V all the c-vectors from V_{int} (bottom level), and in a second step, a set of vectors which results from merging o-vectors in V_{int} with vectors in V_{ext} (top level) which overlap in one (open or observable) label. Finally, the rest of the unmatched (not merged) vectors in V_{ext} are added to V.

2. The resulting VLTS for the merged contract is obtained by computing the free product [2] of the bottom and top level VLTSs (LTS_{vi} and LTS_{ve}, respectively), where transitions containing merged vectors (Algorithm 2, lines 7, 11) have been previously relabeled.

Now, we define more formally the different functions we use in Algorithm 2. First, we introduce function $id(l = e!(v_1 \ldots v_n)) = e!$, $id(l = r?(x_1 \ldots x_n)) = r?$, which returns a unique identifier for each label (by using its name and direction). We extend this function to obtain a set of unique label identifiers from a label set in function $ids(\{l_1, \ldots, l_n\}) = \{id(l_1)\} \cup \cdots \cup \{id(l_n)\}$. Function $obs(v = e : \langle l_l, l_r \rangle) = e$ determines if a vector is observable from outside the scope of its composite.

Example. After applying the aforedescribed contract merging process to the service hierarchy in our example, we obtain the flat contract described in Section 3 (Figure 4). All

Algorithm 2 *merge_vectors*

Merges two set of vectors of adjacent hierarchical levels.
inputs Bottom level vector set V_{int}, Top level vector set V_{ext}
output Vector set V

```
1: $Observable := \{v \in V_{int} | obs(v) \neq c\}$
2: $V := V_{int} \backslash Observable$
3: $Vaux_{ext} := V_{ext}$
4: for all $v_o = o : \langle s_o : l_o \rangle \in Observable$ do
5:     if $\exists v_{ext} = c : \langle s_{ext1} : l_{ext1}, s_{ext2} : l_{ext2} \rangle \in V_{ext}, id(l_o) \in ids(\{l_{ext1}, l_{ext2}\})$ then
6:         $(l_n, s_n) := (l, s) \in \{(s_{ext1}, l_{ext1}), (s_{ext2}, l_{ext2})\}, id(l) \neq id(l_o)$
7:         $v_n := c : \langle s_o : l_o, s_n : l_n \rangle$
8:         $Vaux_{ext} := Vaux_{ext} \backslash \{v_{ext}\}$
9:         $V := V \cup \{v_n\}$
10:    else if $\exists v_{ext} = o : \langle s_{ext} : l_{ext} \rangle \in V_{ext}, id(l_o) = id(l_{ext})$ then
11:        $v_n := o : \langle s_o : l_o \rangle$
12:        $Vaux_{ext} := Vaux_{ext} \backslash \{v_{ext}\}$
13:        $V := V \cup \{v_n\}$
14:    end if
15: end for
16: $V := V \cup Vaux_{ext}$
17: return $V$
```

bindings in a flat contract are always represented by closed vectors. Moreover, the bindings resulting from an actual merge of vectors which overlap in the non-flat initial contracts are labeled with an N (*e.g.*, $v_{NloginD}$, v_{NreqS}, etc.). Figure 6 (right) shows a simplified graphical representation of the bindings in the flat contract. Figure 4 also depicts the VLTS for the flat contract obtained by performing the free product of the two input VLTSs. It is worth observing that before this free product is performed, transitions on the input VLTSs are relabeled with the names of merged vectors (*e.g.*, $v_{loginD} \rightarrow v_{NloginD}$, etc.).

5 Interactive Contract Specification

In order to make the contract design as simple and user-friendly as possible, we advocate interactive specification techniques to support the designer through this process. To this purpose, we first propose a notation to graphically make explicit bindings between ports. We also overview a protocol similarity measure which is used to suggest some port connections to the designer.

Graphical notation. The graphical notation for a service interface includes a representation of its protocol (STS) and a collection of ports. Each label on the STS corresponds to a *port* in the graphical description. Ports include a *data port* for each parameter contained in the parameter list of the label. Correspondences between the different service interfaces are represented as *port bindings* (c-vectors) and *data port bindings* (solid and dashed lines, respectively). Starting from the graphical representation of the interfaces,

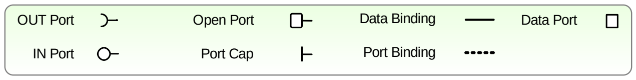


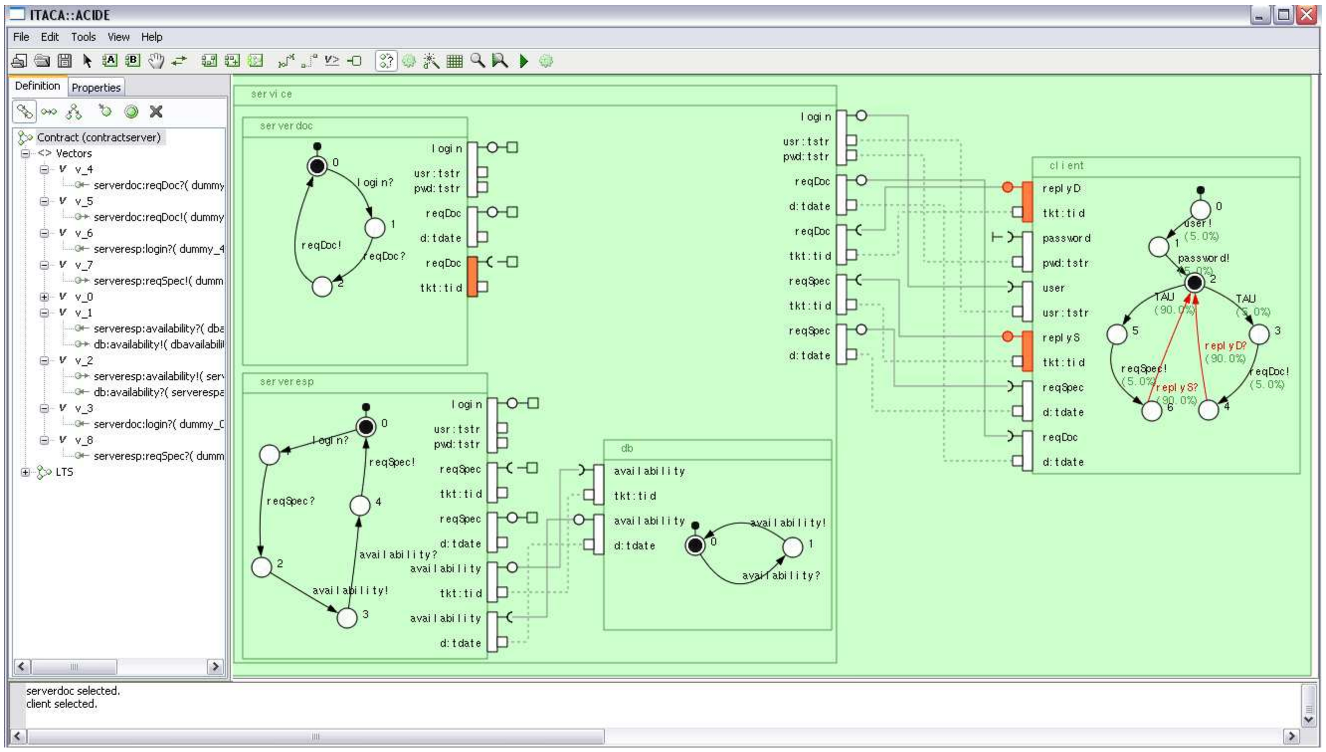
Figure 8. Graphical notation: ports and bindings

the designer builds a contract by successively connecting ports and data ports. This results in the creation of bindings which specify how the interactions should be carried out. It is also possible to add a *port cap* (c-vector with a single label) on a port in order to indicate that it does not have to be connected anywhere. Our graphical notation considers hierarchical relations among interfaces as well (see Section 4 for the underlying principles). Thus, a port can be *open* (o-vector), and it will appear in the external interface of the composite service to which it belongs. Figure 8 summarizes ports and bindings used in our notation.

Protocol similarity. Comparing two protocols helps to build adaptation contracts by suggesting the best possible interface matches to the user. To do so, we define a similarity measure which aims at pointing out mismatches between two protocols, but also at detecting parts of them which turn out to be similar. Our similarity measure accepts as input two service protocols described as STSs and results in a matrix where each entry corresponds to a similarity score for a couple of states (s_i, s_j) with s_i and s_j belonging respectively to the two STS sets of states. This score for each couple of states ranges between 0 and 1, and is computed from a set of four detailed similarity comparisons, namely for states, labels, depths and graphs. The whole similarity measuring process is not presented here for lack of space, but the reader may refer to [18] for more details.

Example. Let us focus on the graphical representation of the database service in our example (Figure 9 gives a graphical description in our ACIDE tool). It can be observed that it contains a port for the reception of the availability request with a data port attached for the date, and another port for the emission of the availability response with a data port attached for the ticket identifier issued for the given date. Moreover, the figure depicts the hierarchy of services in our example, where the Serverdoc, Serveresp and Database interfaces are placed inside a composite interface (Service) and interact on a set of bindings defined between their ports. It is worth noticing that the Serverdoc and Serveresp interfaces have several open ports connected to the external interface of Service.

As regards similarity measures, while connecting the external interface of the Service composite with the ports on the Client interface, ports user or password can be compared with the ports on Serverdoc. The best match is login (0.8) for both of them, therefore we choose binding pass-


ITACA::ACIDE
File Edit Tools View Help
Definition
Properties
Contract (contractserver)
Vectors
v_4 serverdoc:reqDoc?(dummy
v_5 serverdoc:reqDoc!(dummy
v_6 serveresp:login?(dummy_4
v_7 serveresp:reqSpec!(dumm
v_0
v_1 serveresp:availability?(dba
db:availability!(dbavailabili
v_2 serveresp:availability!(serv
db:availability?(serverespa
v_3 serverdoc:login?(dummy_C
v_8 serveresp:reqSpec?(dumm
LTS
service
serverdoc
login?
reqDoc!
reqDoc?
serveresp
login?
reqSpec?
reqSpec!
availability!
availability?
db
client
user!
password!
TAU
replyD?
replyS?
reqSpec!
reqDoc!
(90.0%)
(5.0%)
login
usr:tstr
pwd:tstr
reqDoc
d:tdate
tkt:tid
reqSpec
availability
replyD
password
user
replyS

serverdoc selected.
client selected.

Figure 9. Graphical contract specification for our running example in ACIDE

word with login, and then add a cap to user, but connect its data port usr to the user data port required by login. Similarity measures also allow to automatically generate port bindings for labels that perfectly match, saving time to the developer who would otherwise have to relate manually ports which are obviously similar. For instance, the two ports on the Database interface perfectly match with the availability ports on Serveresp, so they can be automatically bound together based on that information.

6 Validation and Verification of Adaptation Contracts

In this section, we propose a set of verification techniques to check that the adaptation contract makes the involved services work correctly. This helps the designer to understand if the behaviour of the system complies with his/her design intentions. These techniques are completely automated, and include four kinds of checks: (i) static checks on the contract *wrt.* STS service interfaces involved, (ii) simulation of the system execution, (iii) trace-checking to find potential deadlocking executions and infinite loops, and (iv) verification of temporal logic formulas using model-checking.

Static analysis. In the first place, our approach implements a set of static checks on the contract under specification. These include determining if all labels used in vectors are defined in service interfaces, finding out if all service identifiers appearing in vectors belong to one of the interfaces involved in the composition, checking if connected parameters have the same type, etc. Although these checks will detect all common errors that occur when a contract is manually written, they are not enough since they do not focus on the interactions between the services and the adaptor defined by the contract, missing out the behavioural issues that might be raised during execution.

Simulation. In order to solve this problem, our approach first includes a simulation algorithm, which extends the composition engine we presented in [6] with value-passing. This algorithm simulates the execution of the system step-by-step and determines how the different behavioural interfaces evolve as different vectors in the contract are executed. Simulation can be run in two different modes:

- Safe mode. Only *safe* vectors (*i.e.,* a vector for which a global termination state of the system exists after its execution) can be selected at each step of the simulation. To determine if a vector is safe, the simulation algorithm relies on a depth-first search, stopping either if a final state for the whole system (the vector is safe and the search ends), or a deadlock state is found (the search backtracks and tries another path). If no final state for the whole system has been found at the end of the search, the vector is not safe.

- Unsafe mode. All applicable vectors can be selected. Although this allows the application of vectors leading to deadlock states (*i.e.,* a vector for which a global termination state of the system does not exist after its execution), this possibility is interesting in order to observe and understand potential flaws in the contract under specification.

Moreover, we propose two different semantics for o-vectors:

- Closed. When an unbound open port (o-vector) is encountered by the simulation algorithm, the service cannot further evolve and the system deadlocks.
- Open. In this case, the service evolves independently on the open port, and simulation continues.

Trace-checking. We also propose some automated techniques to check traces generated using our simulation engine (unsafe, closed mode). In order to achieve this, many random execution traces are first generated, and then traversed to detect those leading to deadlock situations or infinite loops.

Model-checking. Last, our approach integrates process algebra encoding techniques presented in [14] to verify temporal logic properties on the system under construction. To do so, some LOTOS code is automatically generated for each STS service, and for the adaptation constraints specified in the contract. Finally, some properties can be specified by the designer in a modal μ-calculus and verified using the CADP model-checker [15].

7 Tool Support and Experimental Results

The different contributions we have presented before in this paper are fully implemented in a prototype tool named ACIDE (Adaptation Contract Interactive Design Environment – see Fig. 9). ACIDE has been validated on many examples, which range from small ones to experiment boundary cases, to real-world examples such as a travel agency, rate finder services, on-line computer material store, library management systems, a SQL server, and other systems.

Experimental study setup. With the assistance of twelve volunteers, we conducted a small experimental study which helped us to determine how our approach to adaptation contract specification behaves in terms of required development effort and accuracy, compared to manual contract specification. Specifically, tests were conducted by handing over to users adaptation problems which consisted of the graphical description of the behavioural interfaces to be reused in the composition, and a short specification in natural language of what was the intended functionality of the system. Users were asked to perform contract specification either by: (i) directly typing on a text file or writing down on a piece of paper the contract without further assistance, or (ii) making use of our interactive contract specification techniques (ACIDE).

For our study we used different adaptation problems that were either borrowed from research papers, or obtained from our own archive of adaptation problems. Table 1 summarizes the problems used for our study, which are organized according to increasing size and complexity. We also include the number of interfaces involved and ports to connect, as well as the overall size of the protocols as a total number of states and transitions. The table also includes the experimental results (time required to solve the problem and number of errors in the specified contract) for each of the examples using manual (M) and interactive (I) contract specification.

Experimental results. **(i) Efficiency**. Figure 10 shows the results of our experiments. As it can be observed on the top part of the figure, there is a remarkable difference in the amount of time required to solve the different problems between manual and interactive specification, which showed a reduction of 53% on the time required, compared to manual specification. **(ii) Accuracy**. We measure as errors the number of bindings created between ports which were either wrong or useless for the resulting contract. In the case of manual specification we also take into account syntactic errors (our tool-supported approach avoids them). In Figure 10 (bottom), it can be noticed that the number of errors in problem solutions is lower in our approach (a reduction of 59% in the number of errors compared to manual specification). This difference is negligible for small cases, but increases with the complexity of the problem. It is worth noticing that there is a small difference between the two approaches in the case of easyrest-005. This is explained by the low number of mismatches this problem presents relative to its size, something that makes the manual specification for this particular problem less prone to errors.

8 Related Work

Existing works dedicated to model-based behavioural adaptation are often classified in two families, namely restrictive and generative approaches. Restrictive approaches, *e.g.,* [5, 3, 17], are fully automated and try to solve interoperability issues by pruning the behaviours that may lead to mismatch. These techniques do not allow to fix subtle incompatibilities between service protocols. On the other

					Time (s)		Errors	
Problem	**Interf.**	**Ports**	**States**	**Trans.**	**M**	**I**	**M**	**I**
ftp-002	2	9	11	11	338	222	1.77	1.5
client-sup-002	2	12	15	16	480	248	0.33	0.5
which-004	2	17	16	19	486	146	2.95	0.75
online-med-003	3	15	16	17	531	189	5	0
easyrest-005	4	17	22	24	689	310	3	1.66
pda-001	6	46	37	48	2160	1152	27.6	10.66

Table 1. Problem size and experimental results for the two tested approaches.

hand, generative approaches, *e.g.,* [4, 9, 19, 7, 14], are also able to accommodate protocols, for instance by reordering messages and their parameters when required, or by supporting the specification of advanced adaptation scenarios. In the rest of this section, we compare our proposal with existing generative approaches.

Brogi et al. [4] present a methodology for generative behavioural adaptation where component behaviors are specified with a subset of the π-calculus and composition specifications with name correspondences. An adaptor generation algorithm is used to refine the given specification into a concrete adaptor which is able to accommodate both message name and protocol mismatch. More recently, [7, 14] proposed state-of-the-art adaptation approaches that are generative and support adaptation policies and system properties described by means of regular expressions or LTSs of vectors. However, in these works, no support is proposed to help the designer during the contract specification task, which is therefore achieved manually.

As regards interactive contract specification, [9] introduces an approach to service interface adaptation using a visual language based on an algebra over behavioural interfaces. A graphical editor taking as input pairs of behavioural interfaces allows to link them through interface transformation expressions. The output of this tool can be used as input for a service mediation engine which interprets the information in order to perform composition. Although this approach provides the means to define interface transformation expressions graphically, it does not support the incremental specification of adaptation since it only considers pairs of provided-required interfaces. Moreover, our approach provides systematic contract verification mechanisms and protocol similarity measures which help to guide the specification of adaptation using the graphical notation.

In [19], the authors focus on systems where components or services may enter and leave at any time, such as pervasive ones, and propose an incremental approach for the integration and adaptation of software components. This proposal simplifies the design process by building the system incrementally, and thus avoids the costly computation of global adaptors. Two algorithms are proposed respectively for the addition and suppression of a component. In the first case, a local adaptor is generated, and in the second case, some reconfigurations are applied to preserve the consistency of the system. This work shares some similarities with our proposal, such as the incremental process and the generation of local adaptors. However, [19] relies on a very simple model (LTS without value passing), and advocates for a manual writing of the adaptation contract.

To sum up, our solution to design graphically adaptation contracts goes far beyond existing related work, since we combine in a unique environment new protocol similarity results to guide the construction, hierachical structuring to divide the composition and adaptation in smaller pieces, and verification techniques to detect possible design errors. Last but not least, our proposal is completely supported by a prototype tool we implemented.

9 Concluding Remarks

Manual specification of adaptation contracts is a cumbersome and error-prone task which can be simplified by assisting the designer. In this work, we have presented an interactive approach which speeds up the contract specification process and reduces the risk of mistakes in the specification. Our solution relies on compositional and graphical notations, similarity measures which help the designer to identify similar parts of protocols, and some verification mechanisms which range from static checks in contracts to trace and model-checking techniques. Our approach is fully supported by a prototype tool we implemented and applied to many case studies.

As regards future work, we first plan to extend our solution to take goal-oriented adaptation into account. Our interactive environment would accept the graphical specification of temporal properties to be used next as a guide to the adaptation process. We will also propose some techniques to dynamically evaluate such properties. Thus, once a formula is specified, the user is informed about the satisfaction of this property during the contract construction. A second perspective aims at enhancing our approach with techniques dedicated to the automatic generation of adaptation contracts [13]. Although these approaches are often

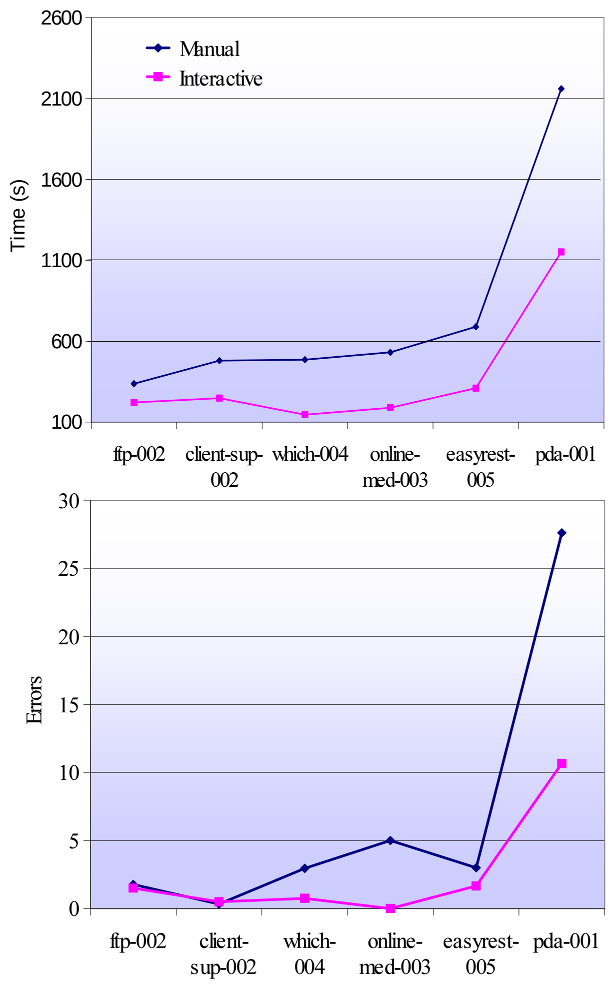


Figure 10. Experimental results: Time elapsed (top) and accuracy (bottom)

costly in terms of computational complexity, and do not permit generative adaptation, they can be helpful to automate the computation of parts of the contract which correct simple mismatch cases.

Acknowledgements. This work has been partially supported by the project TIN2008-05932 funded by the Spanish Ministry of Innovation and Science (MICINN).

References

[1] Pi4SOA Project. `www.pi4soa.org`.

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

[3] M. Autili, P. Inverardi, A. Navarra, and M. Tivoli. SYNTHESIS: A Tool for Automatically Assembling Correct and Distributed Component-based Systems. In *Proc. of ICSE'07*, pages 784–787. IEEE Computer Society, 2007.

[4] A. Bracciali, A. Brogi, and C. Canal. A Formal Approach to Component Adaptation. *Journal of Systems and Software*, 74(1):45–54, 2005.

[5] A. Brogi and R. Popescu. Automated Generation of BPEL Adapters. In *Proc. of ICSOC'06*, volume 4294 of *LNCS*, pages 27–39. Springer, 2006.

[6] J. Cámara, G. Salaün, and C. Canal. Composition and Run-time Adaptation of Mismatching Behavioural Interfaces. *Journal of Universal Computer Science*, 14(13):2182–2211, 2008.

[7] C. Canal, P. Poizat, and G. Salaün. Model-Based Adaptation of Behavioural Mismatching Components. *IEEE Transactions on Software Engineering*, 34(4):546–563, 2008.

[8] J. Cubo, G. Salaün, C. Canal, E. Pimentel, and P. Poizat. A Model-Based Approach to the Verification and Adaptation of WF/.NET Components. In *Proc. of FACS'07*, volume 215 of *ENTCS*, pages 39–55. Elsevier, 2007.

[9] M. Dumas, M. Spork, and K. Wang. Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation. In *In Proc. of BPM'06*, volume 4102 of *LNCS*, pages 65–80. Springer, 2006.

[10] H. Foster, S. Uchitel, and J. Kramer. LTSA-WS: A Tool for Model-based Verification of Web Service Compositions and Choreography. In *Proc. of ICSE'06*, pages 771–774. ACM Press, 2006.

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

[12] M. Hennessy and H. Lin. Symbolic Bisimulations. *Theor. Comput. Sci.*, 138(2):353–389, 1995.

[13] J. A. Martin and E. Pimentel. Automatic Generation of Adaptation Contracts. In *Proc. of FOCLASA'08*, ENTCS. Elsevier. To appear.

[14] 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*, volume 5364 of *LNCS*, pages 84–99. Springer, 2008.

[15] R. Mateescu and M. Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. *Science of Computer Progr.*, 46(3):255–281, 2003.

[16] R. Milner, J. Parrow, and D. Walker. Modal Logics for Mobile Processes. *Theor. Comput. Sci.*, 114(1):149–171, 1993.

[17] H. R. Motahari Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-Automated Adaptation of Service Interactions. In *Proc. of WWW'07*, pages 993–1002. ACM Press, 2007.

[18] M. Ouederni. Measuring Similarity of Service Protocols. Master Thesis, University of Málaga. Available on Meriem Ouederni's Webpage, September 2008.

[19] P. Poizat and G. Salaün. Adaptation of Open Component-based Systems. In *Proc. of FMOODS'07*, volume 4468 of *LNCS*, pages 141–156. Springer, 2007.

[20] G. Salaün. Generation of Service Wrapper Protocols from Choreography Specifications. In *Proc. of SEFM'08*, pages 313–322. IEEE Computer Society, 2008.

[21] G. Salaün, L. Bordeaux, and M. Schaerf. Describing and Reasoning on Web Services using Process Algebra. *International Journal of Business Process Integration and Management*, 1(2):116–128, 2006.

[22] D. M. Yellin and R. E. Strom. Protocol Specifications and Components Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.