Interactive Specification and Verification of Behavioural Adaptation Contracts

Javier Cámara

Department of Informatics Engineering, University of Coimbra, Portugal

Gwen Salaün

Grenoble INP-INRIA-LIG, France

Carlos Canal

Department of Computer Science, University of Málaga, Spain

Meriem Ouederni

Department of Computer Science, University of Málaga, Spain

Abstract

Context: 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 interaction protocols are considered in service interfaces.
Objective: In this article, we propose a tool-based, interactive approach to support the contract design process.
Method: Our approach includes: (i) a graphical notation to define port bindings, and an interface compatibility 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.
Results: Our results show a reduction both in the amount of effort that the designer has to put into building the contract, as well as in the number of errors present in the final result (noticeably higher in the case of manual specification).
Conclusion: We conclude that it is important to provide integrated tool support for the specification and verification of adaptation contracts, since their incorrect specification induces erroneous executions of the system. To the best of our knowledge, such tool support has not been provided by any other approach so far, and hence we consider the techniques described in this paper as an important contribution to the area of behavioural software adaptation.

Keywords:
formal specification, software adaptation, software reusability, interaction protocol

1. Introduction

Services can be accessed and used to fulfill basic requirements, or can be composed with other ser-

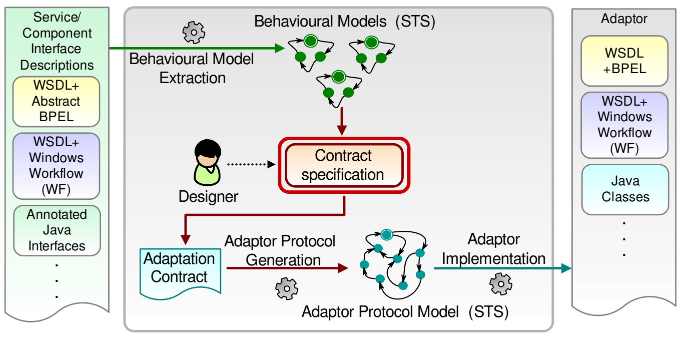

Service/ Component Interface Descriptions
WSDL+ Abstract BPEL
WSDL+ Windows Workflow (WF)
Annotated Java Interfaces
Behavioural Models (STS)
Behavioural Model Extraction
Contract specification
Designer
Adaptation Contract
Adaptor Protocol Generation
Adaptor Implementation
Adaptor Protocol Model (STS)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

Figure 1: Contract-based adaptation process

vices in order to build bigger systems which aim at working out complex tasks. These services must be equipped with rich interfaces to ease their reuse and enable their automatic composition. We can distinguish several interoperability levels for the description of service interfaces (*i.e.,* signature, interaction protocol/behaviour, quality of service, and semantics [1]). Composition of services is seldom achieved seamlessly because mismatch may occur at the different interoperability levels and must be solved. *Software adaptation* [2, 1] 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 (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 (see for instance [3, 4, 5]) are those which favour the full automation of the composition process, and try to solve interoperability issues by pruning behaviours that may lead to mismatch. This restricts in general the functionality of the services involved. A second class of solutions (see for instance [6, 7, 1]) aim at avoiding the restriction of service behaviour, supporting the specification of advanced adaptation scenarios. These solutions build adaptors automatically from an abstract specification, namely an *adaptation contract*, of how mismatch cases must be solved. However, this classification is not strict, since different approaches exhibit features that make their classification in one of the two categories difficult. Take for instance the case of [5], which solves automatically some kinds of mismatch, but requires user input to solve deadlock situations, or [4], which enables the restriction of service behaviour according to user-defined properties, which can be considered as a particular kind of adaptation contract.

Although approaches that use adaptation contracts 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 [6]. 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 interaction protocols are taken into account in interface descriptions.

In this article, we advocate interactive techniques to help the designer in the adaptation contract specification process (see Figure 1 for an overview of the whole adaptation process). For this purpose, we:

- propose a graphical notation to visualize service protocols and define port bindings;
- propose a measure of compatibility between protocols, integrated with our graphical notation, that the designer can use to detect parts of service protocols which turn out to be compatible, and then connect them;
- formalise compositional and hierarchical techniques in order to build the system incrementally, simplifying the process; and
- 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).

Contract construction is an incremental process where understanding the reasons behind incorrect behaviour of the composition is fundamental in order to obtain a correct result. Our choice of taking an interactive approach to adaptation contract construction aims at improving the development process, enabling the user to perform interactive simulation of the system behaviour in order to understand the problem, and correct the specification, all from within the same environment. This results in a more agile development cycle compared to alternatives such as encoding the contract and service protocols into the input language of an existing verification tool, which may result in a tiresome trial-and-error process.

Our approach is fully implemented in a prototype tool, ACIDE, which has been applied to many case studies.

A preliminary version of this work has been published in [8], and is improved here in several aspects. In this article, (i) we extend the textual and graphical contract language to consider data consumption and data synthesis, (ii) we extend the adaptor protocol synthesis techniques described in [9] to take these new contract operators into account, (iii) we present a new simulation algorithm which considers the aforementioned extensions and allows message reordering, (iv) we propose new techniques to measure the compatibility degree of two service interfaces using propagation algorithms, (v) we extend our contract verification techniques with trace-based checking of temporal logic formulas, (vi) we illustrate the different parts of our approach on a revised case study, and (vii) all along the article, we add detailed explanations and comments; for instance we present an extended discussion comparing our approach with related works.

The rest of this paper is structured as follows: Section 2 presents our service model. Section 3 introduces our contract specification language and provides an overview of 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 compatibility 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

This section describes the interface model that we use throughout our proposal and its relation with existing implementation platforms. In particular, since we intend to tackle mismatch both at the signature and behavioural levels, we assume that interfaces are equipped both with a signature (representing the set of required and provided operations), and a protocol specifying the interactive behaviour followed by the component or service in terms of communication actions. These actions correspond to the invocations of operations described in the signature. In our approach, protocols are represented by *Symbolic Transition Systems* (STS).

2.1. Signature

Definition 1 (Signature). *A signature Σ is a set of provided and required operation profiles. An operation profile is the name of an operation, along with its argument types (possibly none), and its return types (possibly none):*

$$op : ti_1 * \ldots * ti_n \rightarrow to_1 * \ldots * to_m$$

Provided operations implement the functionality of the component and are thus offered to other components in the environment for invocation, whereas required operations are those that the service needs to invoke in order to fulfill its purpose.

Signatures are usually described as a set of operation profiles in component-based frameworks (*e.g.,* J2EE) using an Interface Definition Language (IDL), or with WSDL descriptions in the field of Web services. Specifically, in the case of WSDL, services are defined as collections of ports. A port includes the set of operation profiles supported. Moreover, each operation may contain a specific set of input and output messages carrying the arguments and return values of the operation, respectively.

Example. Consider a simple Web service (named MedDB) within the context of a health care organization. MedDB receives requests for information about

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="MedDb">
    <types/>
    <message name="availabilityRequest">
        <part name="d" type="xsd:string"/>
        <part name="hash" type="xsd:string"/>
        <part name="correlation" type="xsd:token"/>
    </message>
    <message name="availabilityResponse">
        <part name="tkt" type="xsd:int"/>
        <part name="correlation" type="xsd:token"/>
    </message>
    <portType name="MedDbPortType">
        <operation name="availability">
            <input name="input1" message="tns:availabilityRequest"/>
            <output name="output1" message="tns:availabilityResponse"/>
        </operation>
    </portType>
</definitions>
```

Figure 2: WSDL specification for the MedDB service.

the availability of a specialist doctor for a given date (supplied in the request), and responds to them with a token or ticket identifier for an appointment (or an error code if there are no available specialist doctors for the given date). To be valid, the request also requires a hash parameter obtained from the information of a valid user (username and password). The specification of the single operation available on the interface, given as a WSDL description (Figure 2), corresponds to the following signature:

$$\text{availability} : \text{string}, \text{string} \rightarrow \text{int}$$

In this case, availability is the name of the operation, whereas string and int are the types of the inputs (date and a hash parameter) and output token of the operation, respectively. It is worth observing that at the signature level, we are only interested in which argument types are required as input, or returned by the operation as output. Argument names will be included at the behavioural interface level.

2.2. Protocol

In our approach, the protocol of a component or service is represented on its interface by a Symbolic Transition System (STS). In an 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* in a transition represents either the internal action τ, or a communication event.

Definition 2 (Communication Event). *A Communication Event is a tuple* (M, D, PL)*, where:*

- M *is a message name.*
- $D \in \{!, ?\}$ *stands for the communication direction* (! *for emission, and* ? *for reception).*
- $PL = [p_1, \ldots, p_n]$ *is either a list of expressions if the message corresponds to an emission, or a list of variables local to the service if the message is a reception.*

Definition 3 (STS). *A Symbolic Transition System is a tuple* (A, S, I, F, T) *where: given a set of communication events* C, $A = C \cup \{\tau\}$ *is an alphabet which corresponds to the set of labels 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 is a reduced version of STG (Symbolic Transition Graph) introduced in [10]. Here, guards are abstracted as τ transitions, which denote internal (unobservable) activities of the service.The operational semantics of a STS ($\rightarrow_b$) is defined in Figure 3. A couple $\langle s, E\rangle$ is composed by an active state $s \in S$ and a data environment E. A data environment is a set of pairs $\langle x, v\rangle$ where x is a variable and v a ground value. We use a function *type* which returns the type of a variable or a value, and we define the environment update "$\oslash$", and the evaluation function ev as follows:

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

Specifically, the three rules in Figure Figure 3 model: (i) transitions internal to the service (TAU); and (ii) transitions where the STS evolves through an emission or a reception (EM and REC, respectively). It is worth observing that no environment updates are included in these rules.

The operational semantics of n ($n > 1$) STSs ($\rightarrow_c$) is formalised using a synchronous communication rule (COM, Figure 4) in which value-passing and

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

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

Figure 3: Operational Semantics of one STS

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

$$\frac{i \in \{1..n\} \quad \langle s_i, E_i\rangle \xrightarrow{\tau}_b \langle s_i', E_i\rangle}{\{as_1, .., \langle s_i, E_i\rangle, .., as_n\} \xrightarrow{\tau}_c \{as_1, .., \langle s_i', E_i\rangle, .., as_n\}} \qquad \text{(INE}_\tau\text{)}$$

Figure 4: Operational Semantics of n STSs

```
<?xml version="1.0" encoding="UTF-8"?>
<process name="MedDB"/>
    <partnerLinks>
        <partnerLink name="DB" partnerLinkType="tns:MedDb"
         myRole="MedDbPortTypeRole"/>
    </partnerLinks>
    ...
    <sequence>
        <receive name="availability_REC_1" partnerLink="DB"
                 operation="availability"
                 portType="tns:MedDbPortType">
        </receive>
        ...
        <reply name="availability_INV_1" partnerLink="DB"
               operation="availability"
               portType="tns:MedDbPortType">
        </reply>
    </sequence>
</process>
```

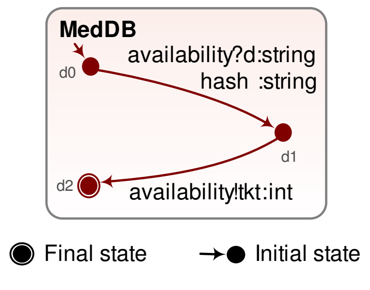


Figure 5: Simplified BPEL specification and STS for the MedDB service.

variable substitutions rely on a late binding semantics [11], and an independent evolution rule (INE$_\tau$, Figure 4).

Example. Figure 5 shows a simplified Abstract BPEL specification and its corresponding STS describing our MedDB service. The STS contains two labels: availability?d,hash and availability!tkt which receive the request for availability along with a date and a hash validation parameter, and respond with a token to the request, respectively.

For the sake of conciseness, in the rest of this paper we will describe service interfaces only with their STS, making explicit argument types in STS labels. □

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 [12, 13, 14, 15] where such abstractions for Web services were used for verification, composition or adaptation purposes). In particular, signature information can be obtained for our models using the information available in WSDL descriptions, whereas STS information is derived from the specification of a service expressed in a behavioural IDL such as Abstract BPEL or Abstract Windows Workflows (see Figure 6). Moreover, this formalism is very convenient for the development of algorithms that rely on the traversal of the different states of protocols, and the transition function available in STS descriptions makes the access to the set of states and their connections straightforward.

If we compare our model to other automata-based formalisms, we may emphasize that transitions in STSs are data-dependent, unlike in interface automata [16], or I/O automata [17], where transitions are labelled exclusively with actions with no data parameters. Data-dependent transitions are required to appropriately model stateful interaction and data exchange among services. Furthermore, other formalisms such as port automata [18] include data-dependent transitions, but similarly to the aforementioned I/O automata, they assume input enabledness, therefore the automaton is receptive towards every possible input action at every state and does not accept certain inputs under the assumption that the environment never generates them. On the contrary, constraint automata [19] are variants of Labeled Transition Systems featuring transitions which include data constraints and do not assume input-enabledness. Although constraint automata have also been used to formalize Web service composition [20] and provide a generic operational model that supports any kind of synchronous and asynchronous peer-to-peer communication, in this work we chose to use STSs, in line with other works in behavioural adaptation of services. In particular, this has allowed us to take advantage of existing implementations of BPEL-STS translation mechanisms.

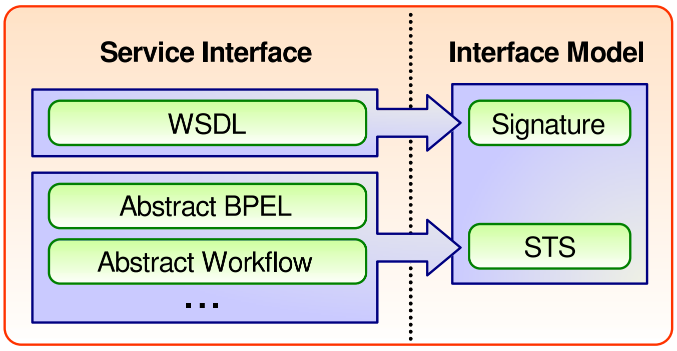
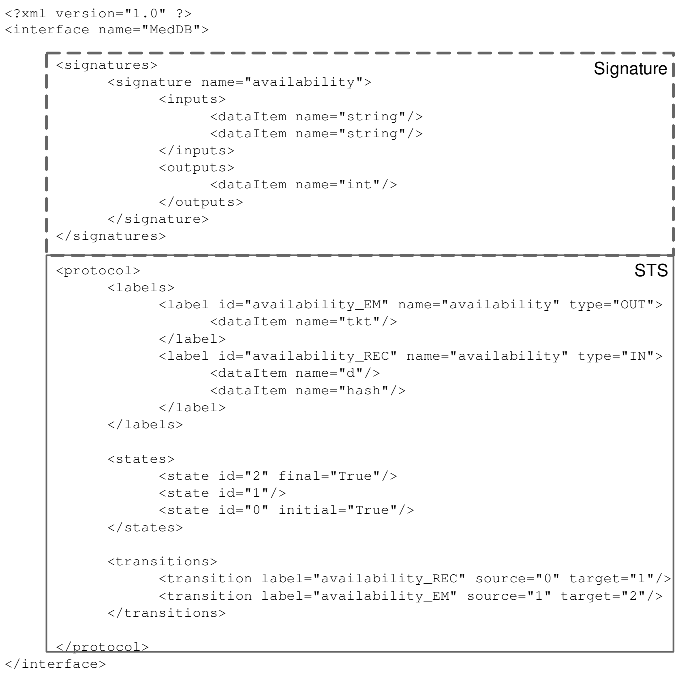


Figure 6: Interface model extraction from existing higher-level interface description languages.

Example. Figure 7 shows the XML description of MedDB's interface, containing a description of its signature (top) and protocol STS (bottom) including labels, states, and transitions. It is worth observing that labels on the STS and operation names can be related through the name attribute included in labels.


Signature
STS

```
<?xml version="1.0" ?>
<interface name="MedDB">
    <signatures>
        <signature name="availability">
            <inputs>
                <dataItem name="string"/>
                <dataItem name="string"/>
            </inputs>
            <outputs>
                <dataItem name="int"/>
            </outputs>
        </signature>
    </signatures>
    <protocol>
        <labels>
            <label id="availability_EM" name="availability" type="OUT">
                <dataItem name="tkt"/>
            </label>
            <label id="availability_REC" name="availability" type="IN">
                <dataItem name="d"/>
                <dataItem name="hash"/>
            </label>
        </labels>

        <states>
            <state id="2" final="True"/>
            <state id="1"/>
            <state id="0" initial="True"/>
        </states>

        <transitions>
            <transition label="availability_REC" source="0" target="1"/>
            <transition label="availability_EM" source="1" target="2"/>
        </transitions>

    </protocol>
</interface>
```

Figure 7: STS-based XML description for the MedDB service.

2.3. *Mismatch Detection*

Most of the time, services cannot be reused as they are because interactions among them would lead to an erroneous execution, namely a *mismatch*. In practice, mismatch situations may be caused when message names do not correspond, the assumed order of messages is not respected, a message in one service has no counterpart, or a message matches with several other messages. Furthermore, mismatch may also appear when the number and/or type of parameters do not match.

More formally, cases of mismatch may lead the whole system to a deadlock. A system deadlocks when all its constituent services are blocked because none of them meet the condition to move to a different state. Accordingly, mismatch detection is performed by exploring all the interactions of the set of service STSs obtained by application of the COM and INE$_\tau$ rules.

However, this method does not extract all the mismatch cases but only those that can be reached assuming that the involved services can interact using the same set of message names. This test can then be used as a first step to start the construction of the adaptation contract (presented in Section 3) that describes how mismatch situations are resolved. While building the adaptation contract, the designer can incrementally build and correct it in subsequent steps by applying the aforedescribed test until the composition is deadlock-free.

Example. In order to illustrate different mismatch situations that may arise, we describe an on-line medical management system which handles patient appointments within a health care institution, either with general practitioners, or specialist doctors.

As it can be observed in Figure 8, we reuse three services in this new system, and we give an example of user requirements implemented in a client:

- The Client can first log on to a server by sending respectively his/her user name (user!) and password (password!). Then, depending on his/her preferences (internal choice specified with τ transitions in the client protocol), the client can stop at this point, or ask for an appointment either with a general practitioner (reqDoc!) or a specialist doctor (reqSpec!), and then receive an appointment identifier.

- Service ServerDoc first receives the client user name and password (id?). Next, this service receives a request for an appointment with a general practitioner (reqDoc?) and replies (reqDoc!).

- Service ServerEsp first receives a request for an appointment with a specialist doctor (reqSpec?), followed by the client user name and password (id?). After checking doctor availability for the given date, an appointment identifier is returned (reqSpec!) to the client.

- Service MedDB can receive and reply to 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 either with a general practitioner or a specialist doctor. It is worth observing that in order to provide appointments with specialist doctors, service ServerEsp must check their availability using an external service (in this case, MedDB provides that functionality).

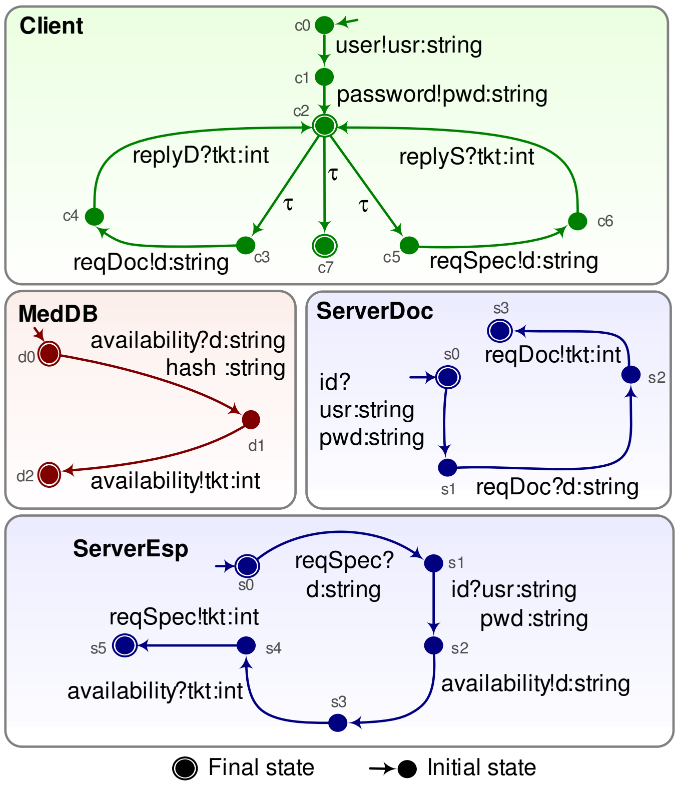


Figure 8: Behavioural interfaces for the online medical management system.

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

- **Name mismatch** occurs if a service expects a particular message, while its counterpart is willing to send one with a different name (*e.g.*, service ServerDoc sends reqDoc!, whereas the client is expecting replyD?). Thus there is a single, one-to-one, conceptual correspondence between a pair of messages in both services, but their names are different.

- **N to M correspondence** appears for instance if a message on a particular interface corresponds to several in its counterpart's interface (or similarly, a message has no correspondence at all). In Figure 8 it can be observed that while the client intends to perform authentication on a service sending user! and password! subsequently, service ServerDoc expects only message id? for authentication. This would be a case of 1-to-N correspondence between message names, and we could consider M-to-N correspondences as the general case, where a group of messages in one of the service corresponds with several others in the counterpart services.

- **Incompatible order of messages.** The relative order of operation invocations among the different protocols involved is different. We may observe this in our example when the client first sends its authentication information and then requests an appointment with a specialist doctor, whereas the ServerEsp service expects these messages in the inverse order.

- **Argument mismatch** may occur when the number and/or type of arguments either being sent or received do not match between the events on the different interfaces. This can be observed in ServerDoc, when id? expects both a username (usr) and a password (pwd). The first data term corresponds to user! on the client interface, whereas the second belongs to password!. Moreover, parameter correspondences on the different interfaces may not always be one-to-one. Thus, in some cases a specific parameter value has to be synthesized from several other values that correspond to parameters sent from one or more services. In our example, we may mention that the value of the hash parameter required by MedDB has to be generated using the values of the user's name and password sent from the client.

The interested reader can refer to [7, 5, 21] for further classifications of mismatch situations made by other authors. In the upcoming section, we describe how to specify the resolution of the mismatch situations presented above by making use of adaptation contracts.

3. Contract Specification and Adaptor Generation

In this section, we present the specification language for adaptation contracts. Although the generation of adaptor protocols from such specifications is not a part of this work, we briefly discuss this process in order to illustrate the basic principles upon which our adaptation approach works.

3.1. Contract Specification Language

Adaptors can be automatically built from an abstract description, called *adaptation contract*, of how mismatch situations can be solved. 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 each other). Some other cases (reordering of messages or data) will be worked out by our adaptor generation algorithms (presented in Section 3.2), which use as input an adaptation contract but also the service interfaces. In this work, we use *compositional vectors* (inspired from *synchronous vectors* [22]) and a *vector LTS* (VLTS) as adaptation contract specification language [23, 1, 9]. A compositional vector (or vector, for short) contains a set of event specifications (message, direction, set of parameters). Each event specified in the vector is executed by one service, and the overall result corresponds to one or several interactions between the involved services and the adaptor. Vectors express correspondences between messages, like bindings between ports, or connectors in architectural descriptions. In particular, we consider a binary communication model, therefore our vectors are always reduced to one event specification (when a service evolves independently) or two (when services communicate indirectly through the adaptor). Furthermore, variables local to the adaptor are used as placeholders in message parameters when these appear in vectors. The same placeholder variable names appearing in different event specifications (possibly in different vectors in the contract) relate sent and received arguments in the messages. It is worth noticing that placeholders are only used in event specifications included in the contract (specifically, in vectors), and not in actual communication events on the different STSs.

Definition 4 (Parameter Substitution). *We define a parameter substitution function* $psub$ *which subtitutes parameters in communication events by place-*

```
1   <id> ::= <alphabetic_char> | <id> <alphabetic_char> | <id> <digit>
2   <ph_id> ::= <id> | <id>"#"
3   <id_set> ::= <id> | <id> "," <id_set>
4   <ph_id_set> ::= <ph_id> | <ph_id> "," <ph_id_set>
5   <event_spec> ::=  <id> ":" <id> <dir> |  <id> ":" <id> <dir> <ph_id_set>
6   <dir> ::= "?" | "!"
7   <vector_prefix> ::= "o" | "c"
8   <event_spec_set> ::= <event_spec> | <event_spec> ";" <event_spec>
9   <vector> ::= <id> "=" <vector_prefix> ":" "<" <event_spec_set> ">"
10  <vector_definitions> ::= <id> "=" "{" <vector_set> "}"
11  <vector_set> ::= <vector> | <vector> "," <vector_set>
12  <param_definition> ::= "(" <id> "," "{" <id_set> "}" "," <id> ")"
13  <param_definitions> ::= <id> "=" "{" <param_definition_set> "}"
14  <param_definition_set> ::= <param_definition> | <param_definition> "," <param_definition_set>
15  <transition> ::= "(" <id> "," <id> "," <id> ")"
16  <transition_set> ::= <transition> | <transition> "," <transition_set>
17  <vlts> ::= "(" <id_set> ";"  <id_set> ";"  <id> ";"  <id_set> ";" <transition_set> ")"
18  <contract> ::= "(" <vector_definitions> "," <param_definitions> "," <vlts> ")"
```

Figure 9: BNF Grammar for adaptation contracts.

holders defined in a set of names F as:
$psub((M, D, PL), F) = (M, D, PL')$,
where $\mid PL \mid = \mid PL' \mid$ and $\forall\, p' \in PL',\ p' \in F$. Let us remind that a communication event is a tuple (M, D, PL) where M is a message name, D stands for communication direction, and PL is a list of expressions/variables (please refer to Definition 2 for more details). Moreover, we assume that valid placeholder names in F and valid names in PL belong to disjoint sets.

Definition 5 ((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 $id \times \{\{\{o, c\} \times A'_j\} \cup \{\{c\} \times A'_k \times A'_m\}\}$, with $j, k, m \in \{1, \ldots, n\}$ and $k \neq m$. $\forall\, A_i = \{l_0, \ldots, l_q\}, A'_i = \{psub(l_0, F), \ldots, psub(l_q, F)\}$, F being a set of fresh names. The BNF grammar presented in Figure 9 details the syntax of vectors (lines 5-9).*

Let us remark that function $psub$ substitutes parameter names by placeholder names for a single event specification. When applied to several event specifications in the same (or different) vectors, placeholder names are reused for subsequent event specifications when the specific argument value that the placeholder refers to is involved in different points of service interaction [1]. In any case, it is worth observing that the association between placeholders and operation arguments is determined by the designer of the adaptation contract, as it is described in Section 5.

According to Definition 5, vectors may be prefixed by o or c (referred to as *open* and *closed* vectors, respectively). In this section, we will consider only the use of closed vectors. The use of open vectors is related to the hierarchical compositional aspects of our approach which will be discussed in Section 4.

Parameter correspondences on the different interfaces may not always be one-to-one. Thus, in some cases, specific parameter values have to be synthesized from several ones that are received from one or more services. In order to specify how these values can be obtained, we include in our contract notation an additional construct that enables the definition of new placeholders, based on the values of the placeholders already described in vectors.

Definition 6 (Synthetic Parameter). *A synthetic parameter is a tuple $(nph, \{iph_1, \ldots, iph_n\}, f)$, where nph is the placeholder name for the synthesized*

[1]For further details about placeholder reuse, please refer to Appendix A.

parameter, and $\{iph_1, \ldots, iph_n\}$ is the set of input parameters required to synthesize the value of nph. Function f specifies how the new value is computed from parameters $\{iph_1, \ldots, iph_n\}$. Please refer to line 12 in the BNF grammar presented in Figure 9 for the syntax of the definition of synthetic parameters.

Furthermore, specifying that a value to be received by a service is to be consumed once read is also possible in our contract notation. This is indicated by the "#" tag in the correspoding placeholder variable identifier (*e.g.,* in vector $v_{reqdoc} = c : \langle c :$ reqDoc!D; $sd :$ reqDoc?D#$\rangle$, we indicate that the value sent by the client, represented by placeholder variable D, is to be eliminated from the adaptor store or consumed when received by the counterpart service, and therefore, it will not be available to other services after its consumption).

In addition, the contract notation includes an LTS with vectors on transitions (that we call vector LTS or VLTS).

Definition 7 (VLTS). *A vector LTS is a tuple (A, S, I, F, T), where given a set of vectors V built over a set of n STS, A is an alphabet which corresponds to the set of vector identifiers in V. 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. Please refer to lines 15-17 in the BNF grammar presented in Figure 9 for the specific syntax of a VLTS.*

This vector LTS 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 sequential execution 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 8 (Adaptation Contract). *An adaptation contract for a set of services STS_i, $i \in \{1, .., n\}$, is a tuple $(D, V, VLTS)$ where:*

- *D is a set of synthetic parameters defined over the set of placeholders in event specifications in V.*
- *V is a set of vectors built over the set of services STS_i.*
- *$VLTS$ is a vector LTS, whose alphabet is defined in V.*

Example. Let us recall our on-line medical management system described in Section 2 which handles patient appointments within a health care institution, either with general practitioners, or specialist doctors (Figure 8). We intend to compose these services into a working system where the client can request an appointment with a general practitioner, or also request an appointment with a specialist doctor, provided that there is a previous appointment with a general practitioner (*i.e.*, the client cannot directly schedule an appointment with the specialist).

Figure 10 displays the adaptation contract used to solve mismatch among the services. On the left-hand side of the figure, we have the set of vectors (Definition 5). It is worth observing that all vectors in this contract are closed (prefix c) [2]. 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 three vectors, respectively v_{user}, v_{pwd} and v_{viddoc}, in which we solve existing mismatches by relating different message names (sd:id is related with c:user and c:password). Concretely, we specify the independent evolution of c:user! and c:password! (in vectors v_{user} and v_{pwd}, respectively), and specify how parameter values are to be exchanged by substituting parameters usr and pwd with placeholders U and P in the events specified in the three vectors, making use of the $psub$ function (Definition 4). In particular, it is worth noting that the order in which placeholders appear in vectors respect the order in which parameters are expected by the services (*e.g.*, U appears before P in v_{viddoc}, since sd:id is expecting usr before pwd). Figure 11 further explains how placeholders connect parameters.

The rest of the vectors in the contract work in a similar fashion, relating the remaining parts of the

[2]The use of open vectors for hierarchical composition will be introduced in Section 4.

interfaces. Furthermore, if we focus on the bottom-left part of Figure 10, it can be observed that the contract also includes a synthetic parameter (Definition 6), where the placeholder name is H, its input parameters are U and P, and its value is obtained by applying a function that is the message digest ($md5$ algorithm) of the concatenation of the input parameters.

Regarding the specification of additional constraints on the composition, we can observe in the right-hand side of Figure 10 that the Vector LTS (Definition 7) for the contract constrains the interaction of the Client, ServerDoc, and ServerEsp interfaces by imposing the request for an appointment with a general practitioner ($v_{reqdoc1}$) always before the request of an appointment with a specialist doctor ($v_{reqesp1}$). This is achieved by excluding $v_{reqesp1}$ from the possible transitions in state 0, and including the transition $(0, v_{reqdoc1}, 1)$. 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.

3.2. Generation of Adaptor Protocols

Given a set of service interfaces, and an adaptation contract, an adaptor protocol can be generated using automatic techniques as those presented in [1, 9]. An adaptor is a third-party component that is in charge of coordinating the services involved in the system with respect to the set of constraints defined in the contract. Consequently, all the services communicate through the adaptor, which is able to compensate mismatches by making required connections as specified in the contract. All protocols (adaptor and services) interact with respect to the COM and INE_τ rules presented in Figure 4.

Here, we have extended the techniques presented in [9] to take into account the two enhancements we made on the contract language, namely data synthesis and data consumption. To do so, we have modified the Compositor tool which is in charge of generating the LOTOS code used in a second step as input to Scrutator, a tool which generates the adaptor protocol corresponding to the LOTOS specification (see [9] for more details). The first extension (data synthesis), at the LOTOS specification level, consists of checking the availability of variables (placeholders in contract specifications) involved in the synthesis. Suppose for example that we want to compute the addition of two variables x and y previously received, and send the result to another service. In order to be able to compute this result ($x + y$), we need to check the availability of these variables in the LOTOS process Store which contains all the variables received by the adaptor at any point in its behaviour. As far as the data consumption is concerned, there are now two possible behaviours in the LOTOS specification when some variables are sent along with messages: either the variable does not need to be consumed (no “#” tag for this variable in the contract) and every time this variable is sent by the adaptor, no modification is done on the store, or this variable is tagged in the contract and can be used only once; in such a case the variable is removed from the store. In this sense, the piece of data received by the adaptor from a sender which corresponds to a placeholder tagged with “#”, is destroyed from the store by the adaptor itself, and this does not affect the sending nor the receiving services in any way.

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 [24], or techniques presented in [9] and [15] 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 [25].

Example. Figure 11 shows a small portion of the adaptor protocol generated from the three vectors $v_{user} = c : \langle c : \text{user!U} \rangle$, $v_{pwd} = c : \langle c : \text{password!P} \rangle$ and $v_{viddoc} = c : \langle sd : \text{id?U, P} \rangle$ given in Figure 10. This makes service ServerDoc and the Client interact correctly. We emphasize that the adaptor synchronizes with the services using the same name of messages but the reversed directions, *e.g.,* communication between id? in ServerDoc and id! in the adaptor. Furthermore, when a vector includes more

$$
\begin{aligned}
V = \{v_{user} &= c{:}\langle c{:}\mathsf{user!U}\rangle,\\
v_{pwd} &= c{:}\langle c{:}\mathsf{password!P}\rangle,\\
v_{vidづesp} &= c{:}\langle se{:}\mathsf{id?U,P}\rangle,\\
v_{reqesp1} &= c{:}\langle c{:}\mathsf{reqSpec!DATE};\ se{:}\mathsf{reqSpec?DATE\#}\rangle,\\
v_{reqesp2} &= c{:}\langle c{:}\mathsf{replyS?RES1};\ se{:}\mathsf{reqSpec!RES1}\rangle,\\
v_{viddoc} &= c{:}\langle sd{:}\mathsf{id?U,P}\rangle,\\
v_{reqdoc1} &= c{:}\langle c{:}\mathsf{reqDoc!DATE};\ sd{:}\mathsf{reqDoc?DATE\#}\rangle,\\
v_{reqdoc2} &= c{:}\langle c{:}\mathsf{replyD?RES2};\ sd{:}\mathsf{reqDoc!RES2}\rangle,\\
v_{avail1} &= c{:}\langle se{:}\mathsf{availability!DATE};\ d{:}\mathsf{availability?DATE\#,H}\rangle,\\
v_{avail2} &= c{:}\langle se{:}\mathsf{availability?RES1};\ d{:}\mathsf{availability!RES1}\rangle\ \}\\
D = \{(\mathsf{H},\{\mathsf{U},\mathsf{P}\}, &\ md5(concat(\mathsf{U},\mathsf{P})))\}
\end{aligned}
$$

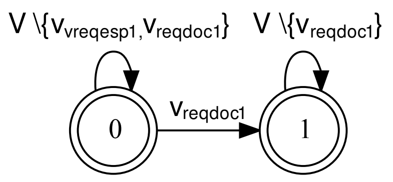


Figure 10: Adaptation contract for our example: vectors and synthetic parameters (left) and vector LTS (right). In this article we use a graphical notation for VLTSs, instead of the actual notation in contracts presented in Figure 9 for the sake of clarity.

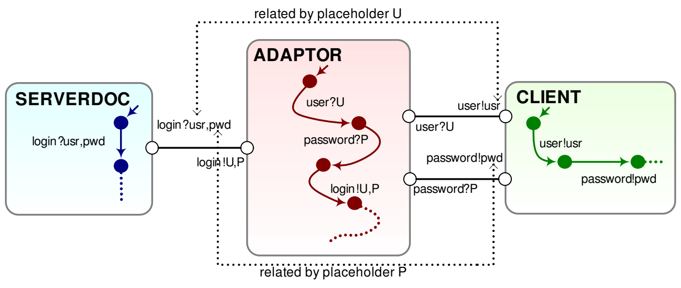


Figure 11: Example of adaptation for authentication mismatches.

than one communication action, the adaptor always starts the set of interactions formalised in the vector with the receptions (which correspond to emissions on service interfaces), and next handles the emissions. In line with these considerations, the development of events in Figure 11 is the following: (i) the adaptor receives the value of the username (parameter usr) sent by the client in in emission c:user!, which is stored in the data environment of the adaptor under the name U; (ii) the adaptor receives the value of the password (parameter pwd) sent by the client in emission c:password!, and stores it in its data environment as P; and (iii) the adaptor can now perform the emission that corresponds to login!U,P, since ServerDoc is ready to receive, and the values of U and P are already available in the adaptor's data environment.

Figure 12 displays the adaptor protocol generated using the adaptation contract shown in Figure 10 where only vectors are considered (the VLTS consists of a single state with all vector transitions looping on it). Interaction starts by receiving the user and password messages sent by the Client. Next, the adaptor can alternatively (i) receive reqDoc from the Client; (ii) login to ServerDoc (id); or (iii) receive reqSpec from the Client. It is worth observing that the adaptor consists of two main parts: on the left-hand side, the client successively interacts with the doctor and with the specialist; on the right-hand side, the client first interacts with the specialist and afterwards with the doctor. The left part is quite sequential, whereas the right one contains more interleavings corresponding to all possible (correct) interaction scenarios.

The full adaptor protocol for our example contains 49 states and 60 transitions. This adaptor has a moderate size and complexity. If we consider the adaptation contract with the vector LTS given in Figure 10 and the same set of bindings, the adaptor protocol consists of 20 states and 20 transitions. This reduction in the number of states and transitions occurs in this case because the addition of the VLTS constrains the composition by imposing sequentiality on the different actions (interactions first with the doctor and in a second step with the specialist), thus reducing interleaving.

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

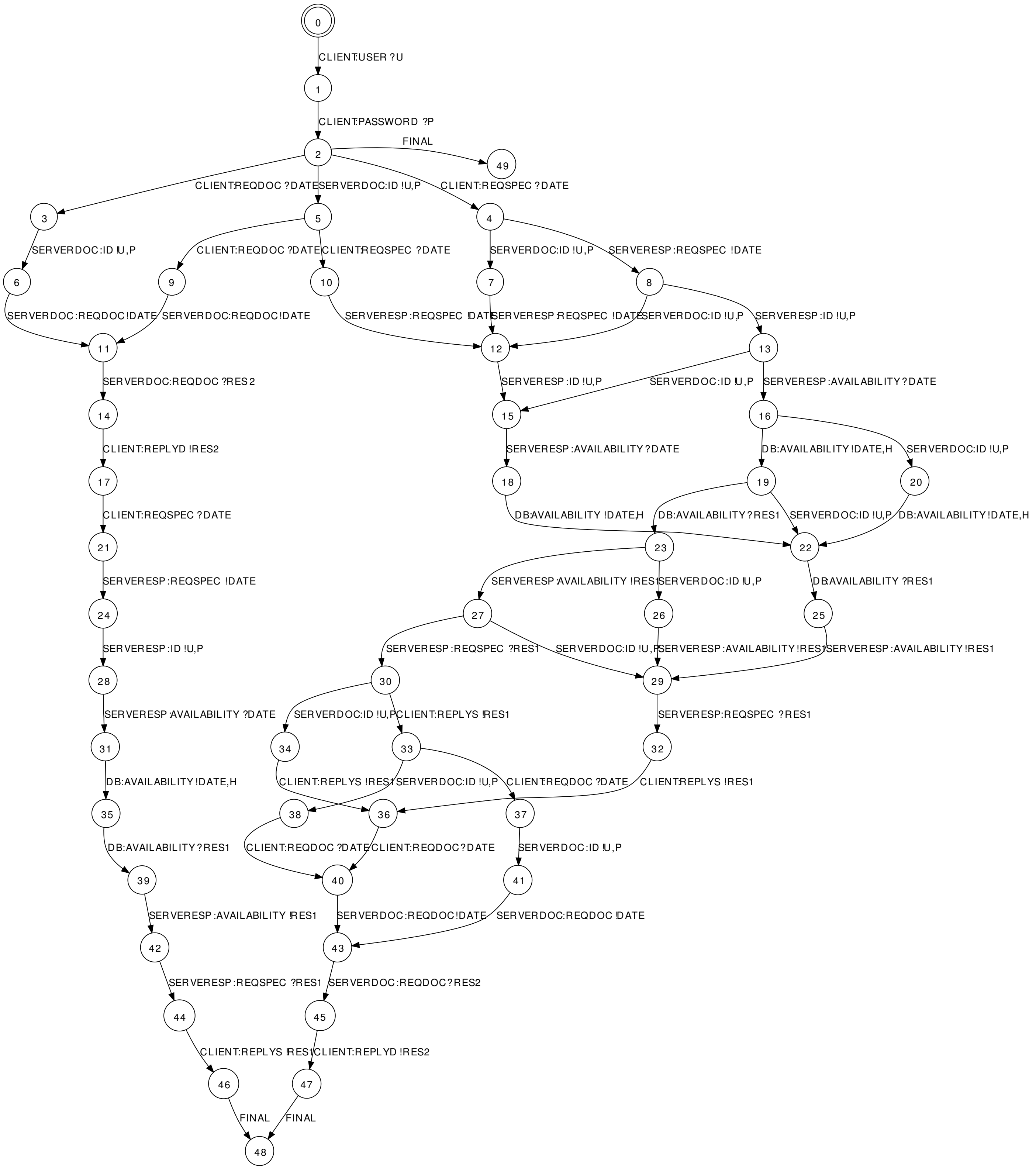


Figure 12: Adaptor protocol generated for the Online Medical System example

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 from the designer since in our approach contracts are automatically obtained from their graphical description.

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, composite services may be independently developed, tested, and modularly replaced by new elements as requirements change.

Definition 9 (Composite Service). *A composite service is a pair* (SI, C) *where:*

- *SI is a set of (composite or basic) service interfaces (i.e., an Id-indexed set of STSs $S_i, i \in 1..n$).*
- *$C = (D, V = V_{int} \cup V_{ext}, LTS_v)$ is an adaptation contract for the set of services in SI:*
  - *D is a set of synthetic parameters.*
  - *V_{int} is a set of vectors of the form $c : \langle l_i, l_j \rangle$, $i, j \in 1..n$, $i \neq j$, where $l_i \in A_i$ and $l_j \in A_j$ are messages which belong to the alphabets of two different STSs in SI. It represents internal bindings between the composite sub-services. We refer to this kind of vector as closed (or c-vector) in the remainder of this article.*
  - *V_{ext} is a set of vectors of the form $o : \langle l \rangle$, where l is a message which belongs to the alphabets of a STSs in SI. It represents ports on the composite subservices which remain open to the environment and therefore are exposed through the composite public interface. We refer to this kind of vectors as open (or o-vector) in the remainder of this article.*
  - *LTS_v is a vector LTS with its alphabet defined on V.*

Example. In our online medical system, services ServerDoc, ServerEsp, and MedDB are bundled within a composite MedService, which interacts with the Client (Figure 13, top). In the remainder of this paper, we will informally refer in our example to the scope of the MedService composite as the *bottom level* of the hierarchy, and to the global scope of the system which contains the Client and the MedService as *top level*:

$$\text{MedService}=(\{\text{ServerDoc},\text{ServerEsp},\text{MedDB}\}, (\emptyset, V_{bot}, LTS_{vbot}))$$

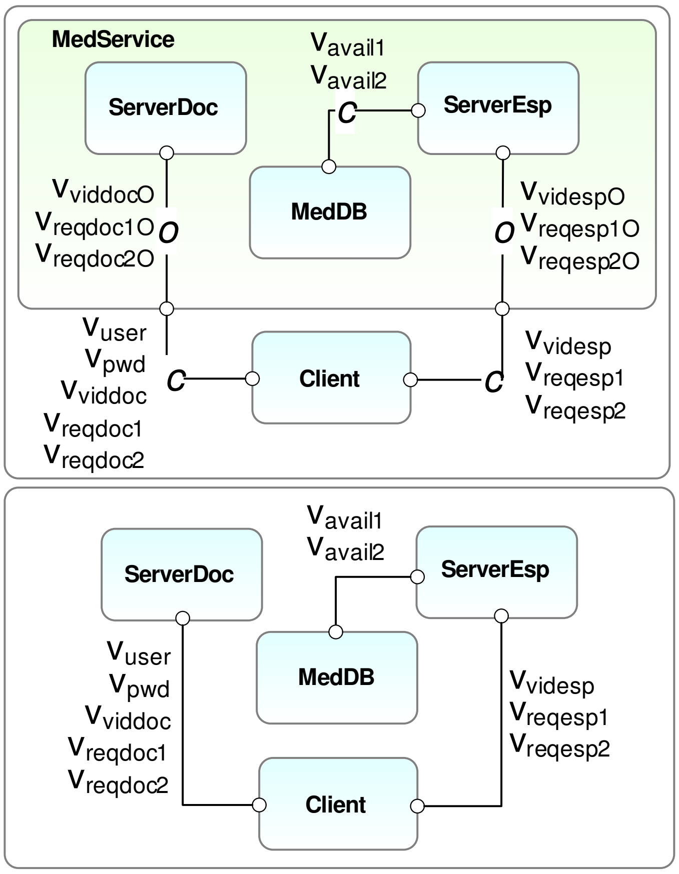


Figure 13: Service hierarchy and bindings (top) and flattened structure and bindings (bottom).

The set of vectors in the bottom level (V_{bot}, internal to the MedService composite interface) represents both bindings internal to the scope of the composite, as well as open ports (which correspond to closed and open vectors, respectively). The former allow the interaction of ServerEsp and MedDB services (closed vectors v_{avail1}, v_{avail2}), whereas the latter (open vectors) enable us to export the rest of the ports in ServerDoc and ServerEsp for external interaction with the client:

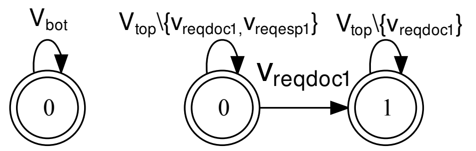
$$
\begin{array}{lll}
V_{bot}=\{ & v_{viddocO} & = o:\langle sd:\mathsf{id?U,P}\rangle, \\
 & v_{reqdoc1O} & = o:\langle sd:\mathsf{reqDoc?DATE\#}\rangle, \\
 & v_{reqdoc2O} & = o:\langle sd:\mathsf{reqDoc!RES2}\rangle, \\
 & v_{videspO} & = o:\langle se:\mathsf{id?U,P}\rangle, \\
 & v_{reqesp1O} & = o:\langle se:\mathsf{reqSpec?DATE\#}\rangle, \\
 & v_{reqesp2O} & = o:\langle se:\mathsf{reqSpec!RES1}\rangle, \\
 & v_{avail1} & = c:\langle se:\mathsf{availability!DATE,H}; \\
 & & \quad d:\mathsf{availability?DATE\#,H}\rangle, \\
 & v_{avail2} & = c:\langle se:\mathsf{availability?RES1}; \\
 & & \quad d:\mathsf{availability!RES1}\rangle\ \}
\end{array}
$$

At the top level, we define the interaction of the Client with the MedService composite interface. It is worth observing that the highest level of any 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.

$$
\mathsf{System}=(\{\mathsf{Client},\mathsf{MedService}\},\\
(\{(\mathsf{H},\{\mathsf{U},\mathsf{P}\}, md5(concat(\mathsf{U},\mathsf{P})))\}, V_{top}, LTS_{vtop}))
$$

$$
\begin{array}{lll}
V_{top}=\{ & v_{user} & = c:\langle c:\mathsf{user!U}\rangle, \\
 & v_{pwd} & = c:\langle c:\mathsf{password!P}\rangle, \\
 & v_{viddoc} & = c:\langle s:\mathsf{id?U,P}\rangle, \\
 & v_{reqdoc1} & = c:\langle c:\mathsf{reqDoc!DATE}; \\
 & & \quad s:\mathsf{reqDoc?DATE\#}\rangle, \\
 & v_{reqdoc2} & = c:\langle c:\mathsf{replyD?RES2}; \\
 & & \quad s:\mathsf{reqDoc!RES2}\rangle, \\
 & v_{videsp} & = c:\langle s:\mathsf{id?U,P}\rangle, \\
 & v_{reqesp1} & = c:\langle c:\mathsf{reqSpec!DATE}; \\
 & & \quad s:\mathsf{reqSpec?DATE\#}\rangle, \\
 & v_{reqesp2} & = c:\langle c:\mathsf{replyS?RES1}; \\
 & & \quad s:\mathsf{reqSpec!RES1}\rangle\ \}
\end{array}
$$

Regarding the specification of additional constraints on the composition we can observe in Figure 14 that the vector LTS in the bottom level contract only contains one state with a transition for all vectors specified in the contract looping on it (it does not impose any constraints on the composition). In contrast, the top-level VLTS constrains the interaction of the Client and the MedService just like the VLTS described in Section 3 did.



V_{bot}
$V_{top}\backslash\{v_{reqdoc1}, v_{reqesp1}\}$
$V_{top}\backslash\{v_{reqdoc1}\}$
$v_{reqdoc1}$
0
0
1

Figure 14: Vector LTSs for the bottom (MedService composite –left) and top level contracts (right).

Both vector and VLTS specifications are automatically generated in our approach from the graphical description of the system that will be presented in Section 5. □

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. 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 open vectors).

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 (and therefore messages exchanged) in the system (Figure 15, 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 15 shows two alternative system architectures: (left) an adaptor which leaves o-vectors open to the environment (A1) is first generated for service composite MedService, and another adaptor (A2) is generated in a second step to enable interoperability between MedService and Client and; (right) a centralized adaptor enables the interaction of all the services after applying contract flattening. □

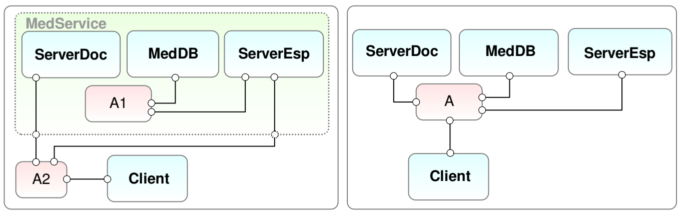


Figure 15: Alternative architectures.

The obtaining 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.

We define now the two functions used in Algorithm 1:

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

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 $FC$
```

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

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 [22] of the bottom and top level VLTSs (LTS_{vi} and LTS_{ve}, respectively), where transitions containing merged vectors (Algorithm 2, lines 7, 11, and 15) have been previously relabeled.

Now, we define more formally the different functions we use in Algorithm 2:

- Function $id(e!(v_1 \ldots v_n)) = e!$, $id(r?(x_1 \ldots x_n)) = r?$, returns a unique identifier for each label (by using its name and direction).
- Function $ids(\{l_1, \ldots, l_n\}) = \{id(l_1)\} \cup \cdots \cup \{id(l_n)\}$ extends function id to obtain a set of unique label identifiers from a label set.
- Function $obs(e : \langle l_l, l_r \rangle) = e$ is used to determine if a vector is observable from outside the scope of its composite or not (*i.e.*, if it is open or

Algorithm 2 *merge_vectors*

Merges two sets of vectors of adjacent hierarchical levels.

inputs Bottom level vector set V_{int}, Top level vector set V_{ext}

output Vector set V

```
1:  $Observable := \{v \in V_{int} \mid obs(v) \neq c\}$
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:     else if $\exists\, v_{ext} = c : \langle s_{ext} : l_{ext} \rangle \in V_{ext}, id(l_o) = id(l_{ext})$ then
15:         $v_n := c : \langle s_o : l_o \rangle$
16:         $Vaux_{ext} := Vaux_{ext} \backslash \{v_{ext}\}$
17:         $V := V \cup \{v_n\}$
18:     end if
19: end for
20: $V := V \cup Vaux_{ext}$
21: return $V$
```

closed). This is achieved by returning the prefix of the vector (o or c for open and closed vectors, respectively).

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 10). All bindings in a flat contract are always represented by closed vectors. Figure 13 (bottom) shows a simplified graphical representation of the bindings in the flat contract. Figure 10 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.

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. Hence, in our approach the designer can perform the specification of a contract through a graphical user interface, and receive at the same time both suggestions and feedback about the inputs that (s)he is providing to the process. For this purpose, we propose: (i) a notation to graphically make explicit bindings between ports; and (ii) a compatibility measure which aims at pointing out mismatches between two protocols, but also at detecting parts of them which turn out to be compatible.

5.1. 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 of the interface. 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 connector lines, respectively). Starting from the graphical representation of the interfaces, 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 T-shaped *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 16 summarizes ports and bindings used in our notation.

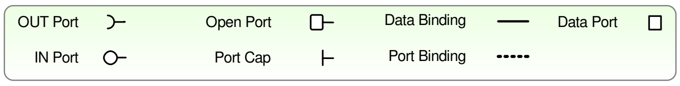


Figure 16: Graphical notation: ports and bindings

Example. Let us focus on the graphical representation of the MedDB service in our example (Figure 17 gives a graphical description of the service hierarchy). It can be observed that it contains a port for the reception of the availability request with a data port attached representing the date (d), and another port for the emission of the availability response with a data port attached representing the ticket identifier issued for the given date. There is also an additional independent data port that represents the synthetic parameter H created for the hash that has to be included in the availability request. Moreover, the figure depicts the hierarchy of services in our example, where the ServerDoc, ServerEsp and MedDB interfaces are placed inside a composite interface (MedService) 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 MedService. □

In our approach, the vector LTS imposing an order on the application of the bindings is built implicitly as new bindings are created. Initially, the VLTS has a single state and no transitions. Each time a new connection is made, the VLTS can be extended in three different ways:

- **Abstract mode.** No order on the application of the bindings is imposed. Let s be the current state of the VLTS. Creating a binding labeled as b in this mode results in the creation of a transition (s,b,s) looping on the current state.
- **Sequential mode.** Bindings created in this mode must be executed one after the other. This results in the extension of the VLTS with a fresh state s' and a transition (s, b, s'). Once this transition is added, the current VLTS state is updated to s'. A subsequent binding creation in this mode will extend the VLTS with a state s'' and a transition (s', b, s'').
- **Branching mode.** Bindings created in this mode are mutually exclusive. The VLTS is extended in this case with a fresh state s' and a transition (s, b, s'). Unlike in sequential mode, the current state is not updated. Thus, a subsequent binding creation in this mode will correspond to the creation of a new state s'', and a transition (s, b, s'').

By proceeding this way, it is possible to build a VLTS for most contracts. However, in some cases the designer may have to adjust the specification to describe situations which cannot be represented using the VLTS extension modes mentioned above. Consider for instance a binding which has to be executed more than once in different parts of the specification. For this purpose, the user should be able to explicitly create a transition between two states in the VLTS, and assign to it a binding (vector) which has been previously created using one of the afore-described extension methods. In order to do this, the environment includes an explicit representation of the VLTS which can be manipulated by the user performing operations such as changing the current state, deleting transitions, or creating transitions assigned to previously created bindings.

5.2. Protocol Compatibility Measure

Comparing two protocols helps to build adaptation contracts by suggesting the best possible interface matches to the user. To do so, we compute the behavioural compatibility degree which aims at detecting parts of both protocols which turn out to be compatible. Our measure accepts as input two service protocols $STS_1 = (A_1, S_1, I_1, F_1, T_1)$ and $STS_2 =$

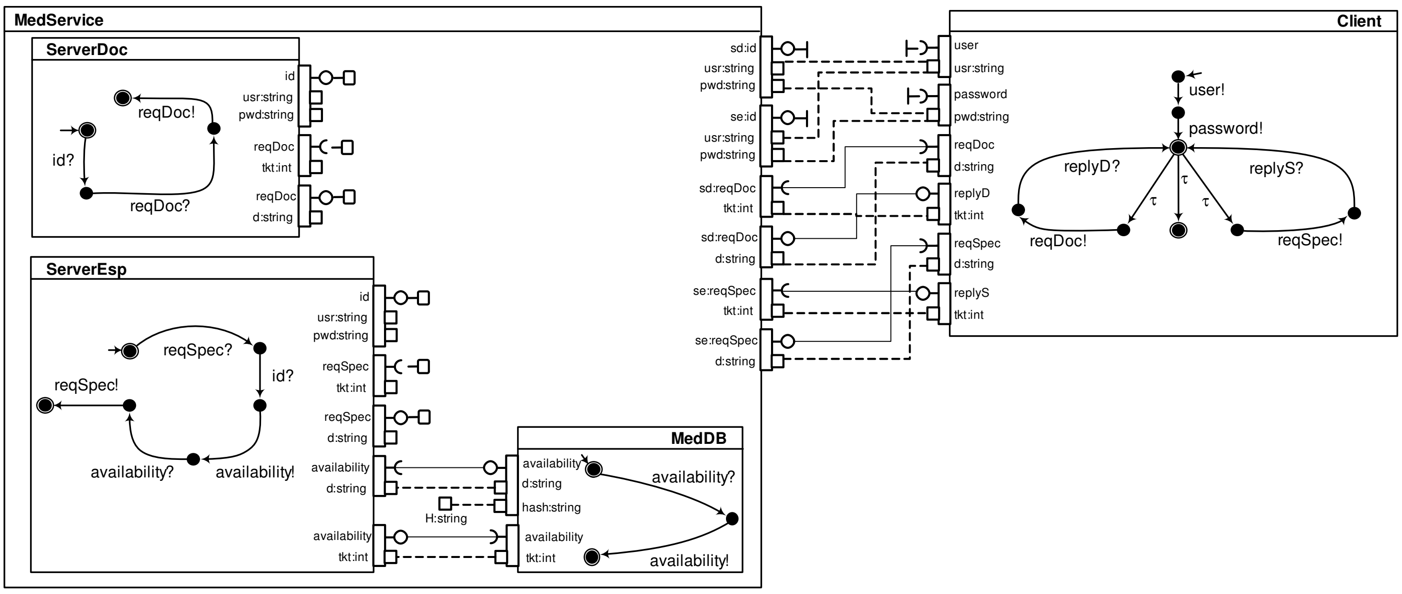


Figure 17: Graphical contract specification for the online medical management system.

$(A_2, S_2, I_2, F_2, T_2)$ and computes a compatibility degree for each global state, *i.e.,* each couple of states (s_i, s_j) with $s_i \in S_1$ and $s_j \in S_2$. All compatibility scores range between 0 and 1, where 1 means a perfect compatibility. Our approach is parameterized by a compatibility notion, that is, we measure how much the two interfaces are far from being compatible *wrt.* this compatibility notion. So far, we have carried out experiments for several notions, such as *unspecified receptions* (UR) [2] or *unidirectional complementarity* (UC) [26]. The UR compatibility requires that if one service can send a message at a reachable state, then the other service must receive that emission. Furthermore, one service can be able to receive messages that cannot be sent by the other service, *i.e.*, there might be additional unmatched receptions. Two services are compatible with respect to UC notion if and only if there is one service which is able to receive (send, respectively) all messages that its partner expects to send (receive, respectively) at all reachable states. Hence, the "bigger" service may send and receive more messages than the "smaller" one. Additionally, for both compatibility notions, services must be free of deadlocks. We will use the UR compatibility notion for illustration purposes in the rest of this section.

Our approach consists in computing first a set of static compatibility measures, and use them in a second step for computing the behavioural compatibility degree for all global states in $S_1 \times S_2$.

Static Compatibility. We use three auxiliary static compatibility measures, namely state nature, labels, and exchanged parameters.

State Nature. The comparison of state nature assigns 1 to each pair of states which have the same nature, *i.e.*, both states are initial, final or none of them. Otherwise, the measure is equal to 0.

Parameters. The compatibility degree of two parameter lists pl_1 and pl_2 depends on three auxiliary measures, namely: (i) the compatibility of parameter number comparing the list sizes; (ii) the compatibility of parameter order measuring the number of types which does not appear in the same order, and (iii) the compatibility of parameter type using the set of unshared types in both lists. These measures must be set to 1 if these lists are empty.

Labels. Protocol synchronisation requires that compatible labels must have opposite directions. Therefore, given a pair $(l_1, l_2) \in A_1 \times A_2$, the label compatibility – $lab\text{-}comp(l_1, l_2)$ – is measured as 0 if these labels have same directions. Otherwise, the computation of this measure uses the semantic distance between message names and the parameter compatibility degree presented above. Here, message names are compared using the Wordnet similarity package [27].

Behavioural Compatibility. We consider a flooding algorithm which performs an iterative measuring of behavioural compatibility for every global state in $S_1 \times S_2$. This algorithm incrementally propagates the compatibility between neighbouring states using backward and forward processing. The compatibility propagation is based on the intuition that two states are compatible if their backward and forward neighbouring states are compatible, where the backward and forward neighbours of global state (s'_1, s'_2) in transition relations $T_1 = \{(s_1, l_1, s'_1), (s'_1, l'_1, s''_1)\}$ and $T_2 = \{(s_2, l_2, s'_2), (s'_2, l'_2, s''_2)\}$ are the states (s_1, s_2) and (s''_1, s''_2), respectively. The flooding algorithm returns a matrix denoted $COMP^k_{CN,D}$ where each entry $COMP^k_{CN,D}[s_1, s_2]$ stands for the compatibility measure of global state (s_1, s_2) at the k^{th} iteration. The parameter *CN* refers to the considered compatibility notion which must be checked according to D, that is, a bidirectional ($\leftrightarrow$) protocol analysis in this article. $COMP^0_{CN,D}$ represents the initial compatibility matrix where all states are supposed to be perfectly compatible, *i.e.*, $\forall(s_1, s_2) \in S_1 \times S_2$, $COMP^0_{CN,D}[s_1, s_2] = 1$. Then, in order to compute $COMP^k_{CN,D}[s_1, s_2]$, we consider the observational compatibility function, $obs\text{-}comp^k_{CN,D}$, and the state compatibility function, $state\text{-}comp^k_{CN,D}$, which combines the forward and backward propagations. In this article, we only present the forward compatibility for lack of space, the backward compatibility can be handled in a similar way based upon incoming rather than outgoing transitions.

Unspecified Receptions. For every global state (s_1, s_2): (i) $obs\text{-}comp^k_{UR,\leftrightarrow}$ returns 1 if and only if every outgoing emission at state s_1 (and s_2) perfectly matches an outgoing reception at state s_2 (and s_1) and all synchronisations on those emissions lead to compatible states; (ii) $obs\text{-}comp^k_{UR,\leftrightarrow}$ returns 0 if there is a deadlock; (iii) otherwise, $obs\text{-}comp^k_{UR,\leftrightarrow}$ measures the best compatibility of every outgoing emission at s_1 with the outgoing receptions at s_2, leading to the neighbouring states which have the highest compatibility degree, and vice-versa.

Forward Propagation. The compatibility is computed from both services point of view. The function $fw\text{-}propag^k_{CN,\leftrightarrow}((s_1, s_2))$ propagates to (s_1, s_2) the compatibility degrees obtained for the forward neighbours of state s_1 with those of state s_2, and vice-versa. For each τ transition, $fw\text{-}propag^k_{CN,\leftrightarrow}$ must be checked on the target state. Observable transitions going out from (s_1, s_2) are compared using $obs\text{-}comp^k_{CN,\leftrightarrow}((s_1, s_2))$.

State Compatibility. The function $state\text{-}comp^k_{CN,D}((s_1, s_2))$ computes the weighted average of three measures: the forward and backward compatibilities, and the value returned by the function comparing state natures.

Compatibility Flooding. As a final measuring step, $COMP^k_{CN,D}[s_1, s_2]$ is computed as the average of its previous value $COMP^{k-1}_{CN,D}[s_1, s_2]$ and the current state compatibility degree. Our iterative process terminates when the Euclidean difference $\varepsilon_k = \|COMP^k_{CN,D} - COMP^{k-1}_{CN,D}\|$ converges.

The interested reader can find more details in [28].

Application to Contract Design. As far as the interactive contract design is concerned, the aforementioned compatibility measure can be used first to automatically generate port bindings for labels that perfectly match. Furthermore, the user can also select a transition label l in one protocol (we call s its source state in the rest of this paragraph), and we return the best label matching in the other protocol. So far, we have implemented two functions which: (i) labels all states in the other protocol with compatibility measures between s and every state in the partner interface, and (ii) seeks the highest value (s, s_j) in the matrix (where s_j is a partner state) and returns the label going out from s_j the most compatible with l. These functions can be completed with other alternatives such as returning the best label matching for each state in the partner, or for each state whose compatibility measure with respect to s is higher than a threshold t. To highlight these results in the graphical interface, we do not only display the compatibility measures but also color in red the best matchings.

Example. Compatibility measures first help to detect port bindings for labels that perfectly match, saving time to the developer who would otherwise have to relate manually ports which are obviously compatible. For instance, the input port reqDoc on the Serverdoc interface perfectly matches with the output port reqDoc on the Client interface, so they can be automatically bound together based on that information. As far as best matching computations are concerned, while connecting the external interface of the MedService composite with the ports on the Client interface, one can for instance click on the output port reqDoc corresponding to a Serverdoc action. In such a case, the best label matching (second function presented above) returns the Client input port replyD with a value of 0.50 (see Figure 18). Therefore, we can choose to bind these ports and this corresponds to vector $v_{reqdoc2}$ in the contract we presented in Section 3.

6. Validation and Verification of Adaptation Contracts

In this section, we propose a set of validation and verification techniques to check that an adaptation contract makes the involved services work correctly. In this sense, it is worth observing that even if there are several adaptation contracts defined at different levels of the service hierarchy, we can obtain a single contract from them by applying Algorithm 1. These techniques are intended to help the designer in understanding potential problematic behaviours of the system which are not obvious (even to the trained eye) just by observing service interaction protocols and adaptation contracts. Indeed, contracts are only abstract specifications that express message and data correspondences between interfaces (vectors), as well as partial order on the interactions to be carried out in the composition (VLTS). As such, contracts do not provide an explicit representation of the actual behaviour of the composition, which has to be generated using state-space exploration techniques in order to assess whether the application of the contract results in a composition that complies with the designer's intentions.

We considered as an alternative to the techniques presented in this section the use of a model-checker (such as CADP or MuCRL2) for verification. However, we decided to develop our own tools for a number of reasons. Firstly, we developed exactly what we need for analysing contracts, and this relies on a single algorithm (simulation) from which we can animate the system under construction and also check temporal properties on the set of traces generated from simulation. Secondly, using an existing model checker would require a translation from one language to another is far from easy because (i) the translation must preserve the semantics of the original language, and (ii) we also need to translate back the results of the verification in the original language if we want the user to understand them. If we consider this last argument, we think that our solution was not more costly (in time and complexity) than the alternative solution of reusing an existing model checker. Finally, we also wanted a self-contained framework, with the additional advantage of having everything available in Acide. This avoids the installation and configuration of other tools in order to use our solution.

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.

6.1. Static Checks

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.

6.2. Simulation

In order to be able to also detect behavioral issues, our approach includes a set of algorithms to perform simulation that are inspired in the composition engine we presented in [29]. However, this new set of algorithms are extended with value-passing, taking into account data consumption and data synthesis. Furthermore, these new mechanism enables message reordering, simulating the execution of the system step-by-step and determining how the different behavioural interfaces evolve as different vectors in the contract are executed.

This section introduces successively two parts. The first one deals with the search for the existence of a correct termination state for the system using depth-first search. The second one includes an algorithm that simulates the behaviour of the system according to a specification given in an adaptation contract, making use of the mechanisms described in the first part.

6.2.1. Existence of Final States

Algorithm 3 takes as input the current state of the system N, and determines if the set of service STSs involved in the composition can reach a correct termination state for the execution under the current adaptation contract. Please observe that for the sake of clarity, we assume in the rest of this section that the different functions and algorithms defined have access to the set of service interfaces $STS_{i \in \{1,\dots,n\}} = (A_i, S_i, I_i, F_i, T_i)$, as well as to the adaptation contract $C = (D, V, VLTS = (A_c, S_c, I_c, F_c, T_c))$, even if these are not explicitly passed as input parameters.

The global state of the system at each step of the execution includes the current *states* of the different STSs, the current state s_c of the vector LTS in the contract, the set of vectors currently under execution RV (initially empty), and a data environment E (initially empty). This algorithm relies on a depth-first search traversal of the state-space of the system, and stops as soon as a final state for the whole system has been found (*i.e.,* the states in all the STSs and the VLTS are final, and all vectors have finished execution). The main idea is that communication actions that belong to the vectors in the contract are applied going in depth until a final state is reached (end of the algorithm), or a deadlock state is found (no further communication actions can be executed). In the latter case, we backtrack and try another path. The algorithm keeps track of already traversed states to avoid endless execution.

Now, we formally define the different functions used in Algorithm 3:

- Function $goal$ determines if the current state of the system is a correct termination state of the execution:
  $goal(N = (states, s_c, RV, E)) = states[1] \in F_1 \land \ldots \land states[n] \in F_n \land s_c \in F_c \land RV = \emptyset$

Algorithm 3 *exist_final*

tests if a final state of the system may be reached from the current state of the system N.

inputs state of the system $N = (states, s_c, RV, E)$
output a boolean

```
 1: $OPEN := []$   // list of open nodes
 2: $CLOSED := []$   // list of visited nodes
 3: $current := N$
 4: $OPEN := push(current, OPEN)$
 5: while $OPEN \neq \emptyset$ do
 6:     $current := pop(OPEN)$
 7:     if $goal(current)$ then
 8:         $return$ True
 9:     else
10:         $CLOSED := push(current, CLOSED)$
11:         for all $n \in successors(current),\ n \notin CLOSED$ do
12:             $CLOSED := push(n, CLOSED)$
13:         end for
14:     end if
15: end while
16: $return\ goal(current)$
```

- Functions $push$ and pop insert and remove and return the last element of a list, using it as a stack:
  $push(e, L = [e_1, \ldots, e_n]) = [e_1 \ldots, e_n, e]$
  $pop(L = [e_1, \ldots, e_{n-1}, e_n]) = e_n\ ,\ L = [c_1, \ldots, c_{n-1}]$

- Function $successors$ obtains the set of reachable system states through the execution of one communication action from the current system state. The result of this function is computed by Algorithm 4.

The different functions used in Algorithm 4 are defined as follows:

- Functions $enabledEM$ and $enabledREC$ determine whether an emission (or a reception, respectively) in vector v can be fired from the current state of the system:

  $enabledEM(N, v) = \exists\, l \in v, (s, l, s') \in T_i\ ,\ s \in states \land em(l) \neq \emptyset$

  $enabledREC(N, v) = \exists\, l \in v, (s, l, s') \in T_i\ ,\ s \in states \land rec(l) \neq \emptyset$

Algorithm 4 *successors*

generates the set of successor states from the current state of the system N.

inputs state of the system $N = (states, s_c, RV, E)$
output set of successor states N_n

```
 1: $N_n := \emptyset$
 2: for all $v \in V$ do
 3:     if $enabledEM(N, v) \vee enabledREC(N, v)$ then
 4:         $successor := successor(N, v)$
 5:         if $successor \notin N_n$ then
 6:             $N_n := N_n \cup \{successor\}$
 7:         end if
 8:     end if
 9: end for
10: return $N_n$
```

with:

$$em(l) = \begin{cases} e & \text{if } l = e!([p_1, \ldots, p_n]) \\ \emptyset & \text{otherwise} \end{cases}$$

$$rec(l) = \begin{cases} r & \text{if } l = r?([p_1, \ldots, p_n]) \\ \emptyset & \text{otherwise} \end{cases}$$

- Function $successor$ obtains the state reachable through the execution of a communication action in vector v. It is worth observing that for any given state of the system, only one communication action in a vector can be fired (either an emission or a reception). We keep track of this with the inclusion of the set of vectors under execution (RV) in the current state of the system in each node representing an execution state. If a vector is contained in RV, it means that its emission has already been executed, and only the reception remains to be processed.

  $successor(N = (states, s_c, RV, E), v) =$
  $(next\_states(states, v), next(s_c, v),$
  $extendRV(RV, v), extendDATA(N, v))$

  where:

  – Functions $next\_states$ and $next$ compute the next states of the involved STS and the VLTS respectively, from their current states and a vector by executing the emission in v (if $v \in RV$), or its reception otherwise:

    $next\_states([s_1, \ldots, s_n], v) = [s'_1, \ldots, s'_n],$
    $\forall\, i \in \{1, \ldots, n\}\ (s_i, l_i, s'_i) \in T_i$
    $next(s_c, v) = s'$ where $(s, v, s') \in T_c$

  – Function $extendRV$ updates the set of running vectors whose execution has started with vector v:

    $$extendRV(RV, v) = \begin{cases} RV \backslash \{v\} & \text{if } v \in RV \\ RV \cup \{v\} & \text{otherwise} \end{cases}$$

  – Function $extendDATA$ updates the data store E with the set of placeholder variables received from the emission that is being executed from vector v. The result of this function is computed using Algorithm 5. Function $consume$ determines if the data value is to be consumed in the reception (this corresponds to placeholders p# in the contract, as we saw in Section 3).

Algorithm 5 *extendDATA*

updates the placeholders available in the data store after the execution of an action in v at state N.

inputs state of the system $N = (states, s_c, RV, E)$, vector v
output updated data store E_n

```
 1: $E_n := E$
 2: for all $l = (m, d, PL) \in v, p \in PL$ do
 3:     if $em(l)$ then
 4:         $E_n := E_n \cup \{p\}$
 5:     else
 6:         if $rec(l) \wedge consume(p)$ then
 7:             $E_n := E_n \backslash \{p\}$
 8:         end if
 9:     end if
10: end for
11: for all $def = (nph, IPH, f) \in D$ do
12:     if $IPH \cap E_n = IPH$ then
13:         $E_n := E_n \cup \{nph\}$
14:     end if
15: end for
16: return $E_n$
```

6.2.2. Composition Algorithm

This section presents an algorithm that manages the composition between several service STSs with respect to a given adaptation contract. Moreover, the proposed approach does not always need to respect the sequential interactions described within vectors of the adaptation contract, that is, events specified in different vectors can be interleaved. Such an interleaving is interesting in cases where sequential interactions as described in the vectors are not enough to reach a correct termination state of the system, and event reordering is required.

To perform the composition of the involved services, Algorithm 6 applies successively actions in vectors that can be fired with respect to the current state of the system. For each vector, first receptions in the adaptation engine are executed (corresponding to emissions in the services), followed by emissions (corresponding to receptions in the services). The algorithm ends when the system has reached a global final state (*i.e.,* the states in all the STSs and the VLTS are final, and all vectors have finished their execution). Since the selection of an applicable vector also relies on the final state existence algorithm presented in Section 6.2.1, we engage the first time in the *while* loop only if there exists a global final state for the system, otherwise the composition is not launched.

Algorithm 6 *composition*

composes a set of service STSs with respect to an adaptation contract

inputs services $STS_{i\in\{1,\dots,n\}} = (A_i, S_i, I_i, F_i, T_i)$, Adaptation contract $C = (V, VLTS = (A_c, S_c, I_c, F_c, T_c))$

```
1: $states := [I_1, \dots, I_n]$   // current states in $STS_i$
2: $E := \emptyset$   // data store
3: $s_c := I_c$   // current state in the vector LTS
4: $RV := \emptyset$
5: $current := (states, s_c, RV, E)$
6: $first := True$
7: $v := select\_vector(current)$
8: while $(\neg goal(current) \lor first) \land v \neq v_\perp$ do
9:     $first := False$
10:    $current := successor(current, v)$
11:    $v := select\_vector(current)$
12: end while
```

The selection of the vector to start executing at each step is performed by function $select\_vector$:

$$select\_vector(N = (states, s_c, RV, E)) = \begin{cases} v & \text{if } enabledEM(N) \lor enabledREC(N), v \neq v_\perp, \\ & exist\_final(successor(N, v)) \\ v_\perp & \text{otherwise} \end{cases}$$

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.
- Unsafe mode. All applicable vectors can be selected. Although this allows the application of vectors leading to deadlock states, this possibility is interesting in order to observe and understand potential flaws in the contract under specification.

6.3. Trace-checking

We also propose some automated techniques to check execution traces. The basic idea is to generate many execution traces using our engine that we will use in a second step to evaluate the adaptation contract. In order to obtain all possible execution traces, and above all the erroneous ones, the final state existence test is turned off. Moreover, we make sure that all traces are finite, making use of bounded loops (the maximum number of allowed loop iterations can be configured by the user in the engine). From such a set of traces, we extract the following information that can be used by the designer to refine and if necessary correct the contract:

- *Unreachable states* allows the designer to identify which states of the vector LTS in the contract cannot be visited.
- *Unreachable transitions* identify the transitions which cannot ever be fired in the vector LTS, preventing access in some cases to a specific state or branch of the composition. It is worth observing that the different behavioural interfaces may prevent the execution of vectors in some cases.
- *Deadlock traces* are particular sequences of applied vectors that lead to a deadlock situation. This information is not obvious at all and the potential number of vector sequences to apply is usually huge. In non-trivial cases, it is impossible for the engineer to check all these potential deadlock situations manually.
- *Livelock traces* are sequences of vector executions that lead to a livelock situation. We may recall that these situations are similar to a deadlock, but in these cases the different service STSs and the vector LTS constantly change their state with respect to one another, although

none of them progress. Hence, the overall system is not able to reach a global final state.

6.4. Trace-Based Verification of Temporal Logic Formulas

We now describe how the trace generation technique described in the previous section can be used in order to verify user-defined safety and liveness properties on the interaction of different services which must be satisfied by every possible execution trace of the system. In order to express such properties, we make use of linear temporal logic (LTL). In particular, we use the next-free variant of LTL (LTL-X), which denotes the class of LTL formulas without the *next* temporal operator and is guaranteed to be insensitive to stuttering [30]. Moreover, we use a LTL finite-trace semantics similar to the one defined in [31], commonly used in run-time verification. In our approach, LTL atomic propositions correspond to communication actions in vectors. Since our simulation algorithm executes these actions one after another in order to make the system evolve, we can assume that the execution of $a!$ is synonymous to the proposition $a!$ in a temporal logic formula.

Defining Composition Properties Using Temporal Logic. When the designer is defining how services must interact in the context of the system, it is interesting to specify:

- **Safety properties**, declaring what should not happen while services are interacting. Hence, no state of the execution path of the system should violate any of the safety properties. Following with our running example, the designer can specify for instance a safety property stating that the client should not in any case obtain an appointment with a specialist doctor if there is no prior appointment with the general practitioner. This can be expressed by the formula $\Box(\neg c : \mathsf{reqSpec}!\ U\ c : \mathsf{replyDoc}?)$. This formula implies that any trace containing the execution of $c : \mathsf{reqSpec}!$ (request for an appointment with the specialist doctor), would have to be preceded at some point by the reception of an appointment with a general practitioner ($c : \mathsf{reqDoc}!$). Otherwise, the trace would violate the property.

- **Liveness properties**, stating what should eventually happen while the service interacts with the rest of the system. As a consequence, the property must hold at some point of the execution path to be satisfied. In our example, an interesting liveness property would be for instance ensuring that an appointment is going to be made at some point either with a specialist or with a general practitioner ($\Diamond(c : \mathsf{replyDoc}? \lor c : \mathsf{replySpec}?)$). Hence, any trace needs to contain at least one execution of any of the two actions in the formula in order to satisfy this property.

  Some other interesting liveness properties, such as responsiveness, can be enforced in the composition too. Clients or services very often send requests that have to be acknowledged (or responded to) by other services. For such systems we are interested in the responsiveness property (*e.g.*, whether every request is eventually acknowledged). In our example we can for instance make sure that the information about availability requested by the specialist service to the database ($se : \mathsf{availability}!$) is going to have a response ($d : \mathsf{availability}!$). Hence, we can express this responsiveness constraint in the formula:

$$\Box(se : \mathsf{availability}! \rightarrow \Diamond d : \mathsf{availability}!)$$

Properties are verified on the execution of the system in two steps:

- Trace generation. Traces are obtained with the final state existence test turned off, if the designer is not interested in general properties covered by trace checking, such as deadlock-freedom, and wants to focus on the specified properties. In contrast, traces can also be generated in safe mode, hence guaranteeing general properties.

- Trace verification. Each trace is run against observer automata built from the LTL formulas specified (see [31] for further details). In particular, for a safety property of the form $\Box\phi$ ($\neg\Diamond\neg\phi$), we build the automaton A_s for $\Diamond\neg\phi$

and the trace violates the property if at the end of the trace, A_s has reached an acceptance state. For a liveness property of the form $\Diamond\phi$, we build its automaton A_l, and the property is satisfied if at the end of the trace, A_l has reached an acceptance state.

Both trace checks and verification of temporal logic formulas can help to quickly identify undesirable situations in the composition, and be used in conjunction with simulation in order to understand the behaviour of the system.

7. Tool Support and Experimental Results

7.1. Tool Support

The different contributions we have presented in the preceding sections are fully implemented in a prototype tool named ACIDE (Adaptation Contract Interactive Design Environment). ACIDE has been implemented in Python, using the wxWidgets toolkit technology for the development of the user interface and a custom-made graphics library built on top of OpenGL$^{\text{TM}}$ in order to visualise the different inputs of the tool. Interfaces and contract are described using an XML-based format specific to the tool.

ACIDE aims at helping the designer in specifying a contract, reducing the risk of errors introduced by manual specification. In contrast with using textual notations where the designer can write any (correct or incorrect) statement, our tool uses the graphical notation presented in Section 5 which enables interactive and incremental construction, as well as checks on the contract (see Figure 18). Thus, any contract produced with the tool is syntactically correct and consistent (*e.g.*, all port and data port names in the contract exist on the interfaces involved in the composition, all vector labels in the VLTS correspond to actual vector definitions in the contract, etc.).

ACIDE has been validated on many 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.

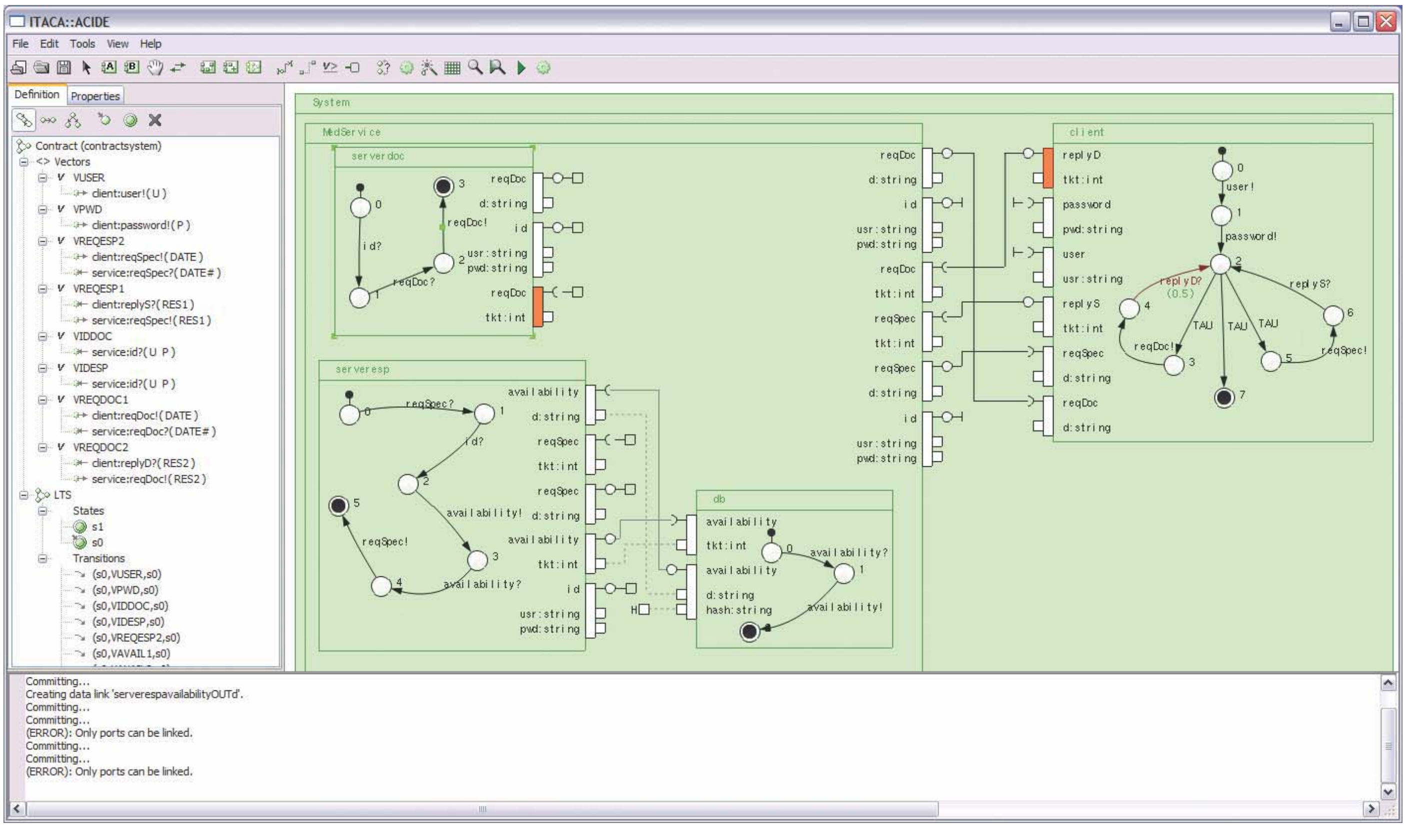
7.2. Experimental Results for Protocol Compatibility

This measure is computed automatically by a prototype tool we implemented and validated on more than 100 examples, *e.g.*, a car rental, a travel booking system, or an online email service. Table 1 summarises the experimental results of some of the examples of our database. Experiments have been carried out on a Mac OS machine running on a 2.53 GHz Intel dual core processor with 4 GB of RAM. In this table, we give successively the number of states and transitions for both service protocols, the compatibility notion used (Unspecified Receptions or Unidirectional Complementarity, see Section 5.2), the global measure of compatibility (between 0 and 1, 1 meaning that services are compatible), the time required to automatically perform this check, and the number of iterations necessary to obtain this result.

Experiments show that small examples with few states and transitions (*e.g.*, Ex9, Ex44, Ex71) require a negligible time for measuring their compatibility, whereas bigger examples (*e.g.*, Ex90, Ex101) need more time (see Table 1). The computation time increases with respect to the number of τ branchings and loops. For instance, Ex85 is quite big but consists of protocols with sequential structure and including very few loops, therefore the computation time does not exceed two minutes. On the other hand, protocols involving many loops (*e.g.*, Ex9) require more time (and iterations) than those having only few loops (*e.g.*, Ex85). To sum up, experiments have shown that our prototype tool computes the compatibility degree of quite large systems (*e.g.*, services with more than 200 states and transitions) in a reasonable time (many iterations are performed in a few minutes). In addition, the returned compatibility measures were very satisfactory. As an example, each time a couple of states in two protocols presents several mismatches, this corresponds to a low value in the matrix and vice-versa. The reader may refer to [32] for some case studies illustrating the preciseness of our compatibility measure.

7.3. Experimental Results for Specification and Verification of Contracts

With the assistance of a group of volunteers, we conducted a small experimental study which



```
Contract (contractsystem)
  <> Vectors
    V VUSER
        client:user!(U)
    V VPWD
        client:password!(P)
    V VREQESP2
        client:reqSpec!(DATE)
        service:reqSpec?(DATE#)
    V VREQESP1
        client:replyS?(RES1)
        service:reqSpec!(RES1)
    V VIDDOC
        service:id?(U P)
    V VIDESP
        service:id?(U P)
    V VREQDOC1
        client:reqDoc!(DATE)
        service:reqDoc?(DATE#)
    V VREQDOC2
        client:replyD?(RES2)
        service:reqDoc!(RES2)
  LTS
    States
      s1
      s0
    Transitions
      (s0,VUSER,s0)
      (s0,VPWD,s0)
      (s0,VIDDOC,s0)
      (s0,VIDESP,s0)
      (s0,VREQESP2,s0)
      (s0,VAVAIL1,s0)
```

```
Committing...
Creating data link 'serverespavailabilityOUTd'.
Committing...
Committing...
(ERROR): Only ports can be linked.
Committing...
Committing...
(ERROR): Only ports can be linked.
```

Figure 18: Hierarchical contract specification for the Online Medical System in ACIDE

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. Our volunteers were categorized in three groups (expert, average, novice) according to their expertise and familiarity with behavioural interfaces and software composition. 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:

- Manual contract specification (M). The user had to directly type on a text file or write down on a piece of paper the contract without further assistance.

- Interactive contract specification (I). Volunteers made use of our interactive environment for contract specification (ACIDE). Users were introduced to the graphical specification of contracts, simulation, and trace checks. Out of the twelve volunteers who participated in the experiment, eight of them (66%) decided to use simulation to check the result, whereas only three of them (25%) used trace checks. The process followed to solve all adaptation problems was incremental construction, combining partial specification of the contract with simulation runs to assess the behaviour of the system. In the case of volunteers who decided to perform trace checks, these were used to directly obtain problematic traces and replicate their behaviour in the simulator in order to understand the problem and fix the contract specification.

In order to avoid obtaining biased results in the experiments: **(i)** the same number of users from each of the three levels of expertise were assigned to each of the two approaches, and **(ii)** case studies were arranged in such a way that no user could be handed

Example	States	Transitions	Comp.	*global*	Time (mn)	Iterations
Ex9	8/5	8/5	UR	0.29	0m0.415s	8
			UC	0.18	0m10.581s	8
Ex44	20/22	19/21	UR	1	0m4.440s	8
			UC	0.81	0m13.860s	3
Ex71	20/4	19/3	UR	1	0m4.112s	8
			UC	1	0m5.848s	9
Ex85	59/59	64/75	UR	0.70	1m1.717s	4
			UC	0.69	0m47.513s	3
Ex90	86/86	90/90	UR	0.72	8m15.806s	7
			UC	0.74	2m48.400s	3
Ex101	124/86	135/90	UR	0.69	19m0.575s	10
			UC	0.70	8m0.460s	6

Table 1: Compatibility degree - some experimental results ($t = 0.7$).

over the same adaptation problem more than once (each user solved different problems using different approaches to prevent previous user knowledge of a particular case study).

For our study we used different adaptation problems that were either borrowed from research papers, or obtained from our own archive of adaptation problems. In particular, we chose a set of problems which ranged from simple protocols and small sets of services, to more complex problems in order to test the scalability of the approach and quantify its benefits with different levels of complexity. Table 2 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.

7.3.1. Efficiency

Figure 19 shows the results of our experiments. As it can be observed on the left part of the figure, there is a substantial 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.

7.3.2. Accuracy

We measure as errors those of a semantic nature in the construction of contracts, *i.e.*, the number of bindings created between ports which were either wrong or useless for the resulting contract, as well as the incorrect connections between data ports on interfaces. In the case of manual specification, it is worth observing that there is a remarkable amount of syntactic errors, although we did not include them for comparison with the interactive approach. In particular, in some cases our volunteers would not even respect the syntax defined for contracts, whereas our approach avoids syntactic errors, and this would have resulted in a distortion of the results obtained by the experiments. It is worth observing that the number of errors appearing in Table 2 is averaged over the volunteers who participated in the experiments. In Figure 19 (right), 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 pointing out 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

In this section, we survey a few key related works which give some solutions to the behavioural adap-

					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 2: Problem size and experimental results for the two tested approaches.

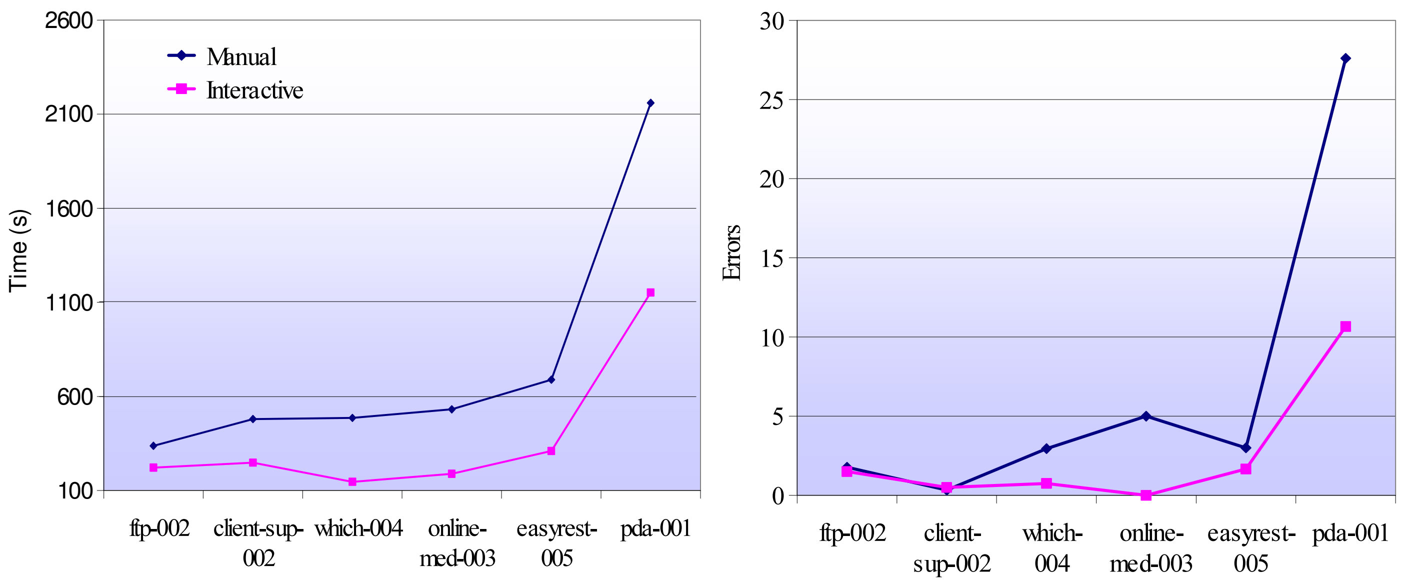


Figure 19: Experimental results: Time elapsed and accuracy

tation of software components and services. In particular, we focus on approaches using a mapping or contract notation in order to guide either the process or the designer (or both) to work out mismatch cases.

Inverardi *et al.* [33, 4] address the enforcement of behavioral properties out of a set of components. Starting from the specification with Message Sequence Charts 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. The set of aforementioned properties has to be given as input to the adaptation process. With respect to adaptor verification, their solution is an alternative to what we propose in this paper. They build an adaptor which ensures by construction the set of properties given as input whereas we advocate model-based verification techniques to check *a priori* that the contract specifies exactly what we expect from the forthcoming adaptor.

Nezhad *et al.* [5] 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 [34]. 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. [5] does not enable the user to write a contract giving an abstract specification of the adaptation (operations and parameter correspondences, etc.), but presents each mismatch case not automatically solvable between two interfaces (mismatch tree), and this can help the user to specify an adaptation contract. In [21], the authors extend the static matching presented in [5] to support one-to-many correspondences. Furthermore, they improve the protocol matching proposed in [5] using depth-based comparison and flooding algorithms similarly to the approach given in [35].

Brogi *et al.* [6] present a methodology for 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, [1, 9] proposed state-of-the-art adaptation approaches that 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.

Concerning interactive contract specification, [7] 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 compatibility measures which help to guide the specification of adaptation using the graphical notation.

In [23], 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, [23] 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 compatibility results (presented in Section 5) 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 article, we have presented an interactive approach which speeds up the contract specification process and reduces the risk of mistakes in the specification. Our approach relies on compositional and hierarchical techniques, a graphical notation, and different verification and validation techniques. Moreover, our proposal is fully supported by a prototype tool we implemented. We also provided some experimental results that we obtained from the application of our approach to different case studies.

These results showed a reduction both in the amount of effort that the designer has to put into building the contract, as well as in the number of errors present in the final result (noticeably higher in the case of manual specification). Since the test cases used for our experimental study were of a small-medium size and complexity, we think that the difficulty of specifying contracts for bigger systems involving dozens of components or services would be not manageable by the designer by using just manual specification. This puts forward the importance of providing support to the development of adaptation contracts, since their incorrect specification induces erroneous executions of the system. To the best of our knowledge, such support has not been provided by any other approaches so far, and hence we consider the techniques described in this paper as an important contribution to the area of behavioural software adaptation.

Concerning future work, we 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 as guidance for the adaptation process. Moreover, we intend to propose 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 (*e.g.*, the environment should be able to inform about the violation of a safety property caused by the binding of two ports as the user is connecting them).

Acknowledgements. The authors would like to thank José Antonio Martín, Pascal Poizat, and the members of the GISUM group at the University of Málaga who participated in the experiments described in this article. This work has been partially supported by the project TIN2008-05932 funded by the Spanish Ministry of Innovation and Science (MICINN), FEDER, and the projects P07-TIC-3184 and PB-TIC-3131 funded by the Andalusian Regional Government.

References

[1] C. Canal, P. Poizat, G. Salaün, Model-based adaptation of behavioural mismatching components, IEEE Transactions on Software Engineering 4 (2008) 546–563.

[2] D. Yellin, R. Strom, Protocol specifications and component adaptors, ACM Transactions on Programming Languages and Systems 2 (1997) 292–333.

[3] A. Brogi, R. Popescu, Automated generation of BPEL adapters, in: Proceedings of the 4th International Conference on Service-Oriented Computing (ICSOC '06), volume 4294 of *Lecture Notes in Computer Science*, Springer, 2006, pp. 27–39.

[4] M. Autili, P. Inverardi, A. Navarra, M. Tivoli, SYNTHESIS: A Tool for Automatically Assembling Correct and Distributed Component-based Systems, in: Proceedings of the 29th International Conference on Software Engineering (ICSE '07), IEEE Computer Society, 2007, pp. 784–787.

[5] H. Nezhad, B. Benatallah, A. Martens, F. Curbera, F. Casati, Semi-automated adaptation of service interactions, in: Proceedings of the 16th International Conference on the World Wide Web (WWW '07), ACM, 2007, pp. 993–1002.

[6] A. Bracciali, A. Brogi, C. Canal, A formal approach to component adaptation, The Journal of Systems and Software 74 (2005) 45–54.

[7] M. Dumas, M. Spork, K. Wang, Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation, in: Proceedings of the Fourth International Conference on Business Process Management (BPM'06), volume 4102 of *Lecture Notes in Computer Science*, Springer, 2006, pp. 65–80.

[8] J. Cámara, G. Salaün, C. Canal, M. Ouederni, Interactive specification and verification of behavioural adaptation contracts, in: B. Choi (Ed.), Proceedings of the 9th International Conference on Quality Software (QSIC'09), IEEE Computer Society, 2009, pp. 65–75.

[9] R. Mateescu, P. Poizat, G. Salaün, Adaptation of service protocols using process algebra and on-the-fly reduction techniques, in: A. Bouguettaya, I. Krüger, T. Margaria (Eds.), Proceedings of the 6th International Conference on Service-Oriented Computing (ICSOC '08), volume 5364 of *Lecture Notes in Computer Science*, pp. 84–99.

[10] M. Hennessy, H. Lin, Symbolic Bisimulations, Theoretical Computer Science 138 (1995) 353–389.

[11] R. Milner, J. Parrow, D. Walker, Modal Logics for Mobile Processes, Theoretical Computer Science 114 (1993) 149–171.

[12] X. Fu, T. Bultan, J. Su, Analysis of interacting BPEL web services, in: S. Feldman, M. Uretsky, M. Najork, C. Wills (Eds.), Proceedings of the 13th International Conference on the World Wide Web (WWW '04), ACM, 2004, pp. 621–630.

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

[14] H. Foster, S. Uchitel, J. Kramer, LTSA-WS: A Tool for Model-based Verification of Web Service Compositions and Choreography, in: Proceedings of the 28th International Conference on Software Engineering (ICSE '06), ACM, 2006, pp. 771–774.

[15] J. Cubo, G. Salaün, C. Canal, E. Pimentel, P. Poizat, A Model-Based Approach to the Verification and Adaptation of WF/.NET Components, in: Proceedings of the 4th International Workshop on Formal Aspects of Component Software (FACS '07), Electronic Notes in Theoretical Computer Science, Elsevier, 2007, pp. 39–55.

[16] L. Alfaro, T. Henzinger, Interface Automata, in: Proceedings of the of the Joint 8th European Software Engineering Conference and the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'01), ACM, 2001, pp. 109–120.

[17] N. A. Lynch, M. R. Tuttle, An introduction to input/output automata, CWI Quarterly 2 (1989) 219–246.

[18] C. Koehler, D. Clarke, Decomposing port automata, in: S. Y. Shin, S. Ossowski (Eds.), SAC, ACM, 2009, pp. 1369–1373.

[19] F. Arbab, C. Baier, J. J. M. M. Rutten, M. Sirjani, Modeling component connectors in reo by constraint automata: (extended abstract), Electr. Notes Theor. Comput. Sci. 97 (2004) 25–46.

[20] S. Meng, F. Arbab, Web services choreography and orchestration in reo and constraint automata, in: Proceedings of the 2007 ACM symposium on Applied computing, SAC '07, ACM, New York, NY, USA, 2007, pp. 346–353.

[21] H. R. M. Nezhad, G. Y. Xu, B. Benatallah, Protocol-aware Matching of Web Service Interfaces for Adapter Development, in: Proc. of the 19th International Conference on World Wide Web (WWW'10), ACM, 2010, pp. 731–740.

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

[23] P. Poizat, G. Salaün, Adaptation of Open Component-based Systems, in: Proceedings of the 9th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'07), volume 4468 of *Lecture Notes in Computer Science*, Springer, 2007, pp. 141–156.

[24] Pi4SOA Project, 2008. `http://www.pi4soa.org`.

[25] G. Salaün, Generation of Service Wrapper Protocols from Choreography Specifications, in: Proceedings of the 6th IEEE International Conference on Software Engineering and Formal Methods (SEFM '08), IEEE Computer Society, 2008, pp. 313–322.

[26] M. Ouederni, G. Salaün, Tau Be or not Tau Be? A Perspective on Service Compatibility and Substitutability, in: Proceedings of the First International Workshop on Component and Service Interoperability (WCSI'10), volume 37, EPTCS, 2010, pp. 57–70.

[27] T. Pedersen, S. Patwardhan, J. Michelizzi, WordNet::Similarity - Measuring the Relatedness of Concepts, in: Proc. of AAAI'04, pp. 1024–1025.

[28] M. Ouederni, G. Salaün, E. Pimentel, Measuring the Compatibility of Service Interaction Protocols (ITI 4-10), Technical Report, Dept. of LCC, University of Malaga, 2010. Available on Meriem Ouederni's Web Page.

[29] J. Cámara, G. Salaün, C. Canal, Composition and Runtime Adaptation of Mismatching Behavioural Interfaces, Journal of Universal Computer Science 14 (2008) 2182–2211.

[30] E. Clarke, O. Grumberg, D. Peled, Model Checking, The MIT Press, Cambridge, Massachusetts, 1999.

[31] D. Giannakopoulou, K. Havelund, Automata-based verification of temporal properties on running programs, in: Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE'01), IEEE Computer Society, 2001, pp. 412–416.

[32] M. Ouederni, Comparator Web Page, 2010. Available at `http://www.lcc.uma.es/~meriem/comparator.html`.

[33] P. Inverardi, L. Mostarda, M. Tivoli, M. Autili, Synthesis of Correct and Distributed Adaptors for Component-based Systems: An Automatic Approach, in: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05), IEEE Computer Society, 2005, pp. 405–409.

[34] E. Rahm, P. Bernstein, A survey of approaches to automatic schema matching, The VLDB Journal 10 (2001) 334–350.

[35] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, P. Zave, Matching and Merging of Statecharts Specifications, in: Proceedings of the 29th International Conference on Software Engineering of (ICSE '07), IEEE Computer Society, 2007, pp. 54–64.

Appendix A: Placeholder Reuse

Placeholders in a contract are reused whenever we add a vector that contains parameters already associated to a placeholder in the current (partial) contract. Specifically, during the contract specification process, we incrementally build a binary relation $R \subseteq PN \times PH$ (initially empty) over the set of possible parameter names PN and placeholder names PH that enables us to tell which parameter names are already associated to a placeholder name in the current (partial) contract.

Furthermore, we also make use of data bindings specified by the designer to keep track of which arguments should be substituted by the same placeholder in different labels:

Definition 10 (Data Binding). *A data binding defined over two labels $l_A = (M_A, D_A, PL_A)$ and $l_B = (M_B, D_B, PL_B)$ is a couple (a, b) such that $a \in PL_A$ and $b \in PL_B$.*

Let us remark that data bindings are specified by the designer according to the particular semantics of the example at hand. Further details about the specification of data bindings can be found in Section 5.

Algorithm 7 substitutes parameters by placeholder names in parameter lists PL_A and PL_B associated to labels in a vector l_A and l_B, respectively, returning the associated lists of placeholders for both labels PL'_A and PL'_B. If the vector contains only one label, we use as input an empty list for PL_B, and the algorithm will output an empty placeholder list PL'_B. Moreover, if two parameter lists are used as input, we assume that l_A corresponds always to the emission, and l_B to the reception.

The algorithm works by processing all parameters in PL_A first, and checking if they are already associated to a placeholder using relation R. If this is the case, the placeholder in use is added to the placeholder list. If the parameter is not associated to a placeholder, we extract a new placeholder name from the pool of placeholders PH, and add it to the placeholder list, extending relation R with the new placeholder at the same time. Once all parameters in PL_A are processed, the algorithm starts processing parameters in PL_B. The process is analogous, but in this case, we first check if there is a data binding relating the parameter with another parameter in PL_A. If this is the case, we obtain the placeholder already associated to the bound parameter in PL_A and use it to extend the list PL'_B.

Algorithm 7 *sub_ph*

substitutes a set of parameter names with placeholder names for a vector with two labels $l_A = (M_A, D_A, PL_A)$, $l_B = (M_B, D_B, PL_B)$

inputs parameter list for labels PL_A and PL_B, current binary relation R, current placeholder set PH, current set of data bindings B

```
1: $PL'_A := []$
2: $PL'_B := []$
3: while $PL_A \neq []$ do
4:     $p := extract\_first(PL_A)$
5:     if $\exists\, ph : (p, ph) \in R$ then
6:         $append(PL'_A, ph)$
7:     else
8:         $ph := extract(PH)$
9:         $R := R \cup \{(p, ph)\}$
10:        $append(PL', ph)$
11:    end if
12: end while
13: while $PL_B \neq []$ do
14:     $p := extract\_first(PL_B)$
15:     if $\exists (q, p) \in B$ then
16:         $append(PL'_B, ph) : (q, ph) \in R$
17:     else if $\exists\, ph \in PH : (p, ph) \in R$ then
18:         $append(PL'_B, ph)$
19:     else
20:         $ph := extract(PH)$
21:         $R := R \cup \{(p, ph)\}$
22:         $append(PL'_B, ph)$
23:     end if
24: end while
25: return $PL'_A, PL'_B$
```

Let us formalize the set of functions we apply in the algorithm:

- Function $append$ adds a new element at the end of a list $append([e_1, \ldots, e_n], e) = [e_1, \ldots, e_n, e]$
- Function $extract\_first$ returns and removes the first element of a list $extract\_first([e_1, e_2, \ldots, e_n]) = [e_2, \ldots, e_n]$
- Function $extract$ returns a random element e_r from a set $E = \{e_1, \ldots, e_r, \ldots, e_n\}$, and removes it from the set ($E := E \backslash \{e_r\}$).