THE UNIVERSITY OF MÁLAGA
DEPARTMENT OF COMPUTER SCIENCE

MASTER THESIS

Measuring the Similarity of Service Protocols

Master of Software Engineering and Artificial Intelligence

Author:
Meriem OUEDERNI

Supervisors:
Dr. Gwen SALAÜN
Dr. Carlos CANAL
Dr. Ernesto PIMENTEL

September 2008

Contents

1 Introduction **2**

2 Model of Services **3**

3 Compatibility and Mismatch **5**
3.1 Compatibility Definition 5
3.2 Behavioural Mismatches 6

4 Measuring Protocol Similarity **8**
4.1 State Similarity 8
4.2 Label Similarity 9
4.3 Depth and Graph Similarity 10
4.4 Protocol Similarity 10
4.5 Prototype Tool: ITACA::Sim 11
4.6 Application to our Case Study 11

5 Applications: Adaptation and Re-engineering **13**
5.1 Software Adaptation 13
5.2 Software Re-engineering 14

6 Related Work **17**

7 Concluding Remarks **18**

Abstract

Composition of components or services is a crucial issue in Software Engineering. However, direct reuse and composition of existing services is in most of cases impossible because their interfaces present some incompatibilities. Here, we assume service interfaces described using signatures and protocols. Protocols are essential because, even if services match from a signature point of view, their combination can behave erroneously and lead the system into deadlock situations. In this thesis, we propose a global similarity measure between two service interfaces to detect how compatible they are. This measure is automatically computed by a prototype tool we have implemented. We also show how such a result can be used in areas such as software adaptation or re-engineering in order to solve composition issues.

This work was conducted at the department of Languages and Computer Sciences (Depto. de Lenguajes y Ciencias de la Computación), School of Computer Engineering (E.T.S. de Ingeniería Informática), University of Málaga, Spain.

DEPTO. DE LENGUAJES Y CIENCIAS DE LA COMPUTACIÓN
ETSI INFORMÁTICA
UNIVERSIDAD DE MÁLAGA
CAMPUS DE TEATINOS
29071 - MÁLAGA, ESPAÑA

Acknowledgements

This master internship was first proposed to me by Gwen Salaün, Carlos Canal and Ernesto Piementel. They offered me the opportunity to prepare my master thesis at the department of languages and computer sciences at the university of Málaga, this is why I address them my first thanks. I am particularly grateful for their support and assistance all along this period. Their constructive comments, ideas and propositions of new perspectives were of invaluable support. It has been a great pleasure to work with them, both from a professional and a personal point of view.

I would like to thank the people with whom I shared the office room. We have had some great time together, and I am sure that there is still a lot more to come. My deep gratitude go to every member of the team for their support, formation and most of all patience. All were to me of an invaluable help and allowed me to acquire or increase my knowledge in the domain.

My thanks go to the external members of the jury for their careful reading of my work.

My final and warmest thanks naturally go to my family, and particularly my parents for their support and help. I would probably not have gone so far in my studies without their encouragements and the education that they gave me.

" If you find that you're spending almost all your time on theory, start turning some attention to practical things; it will improve your theories. If you find that you're spending almost all your time on practice, start turning some attention to theoretical things; it will improve your practice. "

Donald Knuth

Chapter 1

Introduction

Composition of components or services is a central issue in Software Engineering. Reuse of existing entities is mandatory not to implement once and again the same blocks of software, and it helps developers to reduce development time, respect delays, and have their companies save money by diminishing software design costs. However, direct reuse and composition of existing services is in most of cases impossible because their interfaces present some incompatibilities.

Services are accessed through their *interfaces* that usually distinguish four interoperability levels [3]: signature level (service names and types), behavioural level (interaction protocols), service level (non-functional properties such as quality of service), and conceptual or semantic level (functional specification of what the service actually does). Here, we consider services described using signatures and protocols. Protocols give the application order of method calls and exchanged messages. This interoperability level is essential [31] because, even if services match from a signature point of view, their combination can lead to erroneous behaviours or deadlock situations if the designer is not aware of their execution flows. More than only considering messages exchanged in protocols, it is important to include value passing coming with messages (data exchanged between services) since this feature may raise composition issues too (mismatching number of parameters, different ordering, non corresponding types, etc).

In this paper, we first present a model of service interfaces which takes their signatures and protocols into account. Next, we present a notion of compatibility between two interfaces described in our model. The main contribution of this paper is a global measure of similarity between two service interfaces *wrt.* the compatibility definition. The goal of this measure is twofold: pointing out mismatches between two protocols, but also detecting parts of them which turn out to be similar. To do so, we compare successively states (*i.e.,* their nature — initial, intermediate, or final —, and the number of incoming/outgoing transitions), labels (message names, directions, parameters), and protocols (depth and graph structure). Our measure returns a similarity value for each pair of states in the two service protocols, and this result is computed automatically thanks to a prototype tool we have implemented.

Such a similarity measure can have several applications while composing entities. Particularly, in order to make services work correctly together in spite of possible mismatches, we will present how this can be used to generate automatically coordinator or adaptor specifications, or how it can help to automate code re-engineering of services.

The remainder of this paper is structured as follows. Chapter 2 formalises our model of services. Chapter 3 introduces our definition of compatibility. In Chapter 4, we present our approach to measure similarity of two service protocols. Chapter 5 shows some applications, namely software adaptation and re-engineering. Finally, Chapter 6 compares our approach to related work, and Chapter 7 ends the paper with some concluding remarks.

Chapter 2

Model of Services

In this chapter, we present our service interface model. We assume that service interfaces are given using both a signature and a behavioural interface (or protocol).

Definition 1 (Signature)**.** *A signature is a set of provided and required operation profiles. An operation profile is the name of an operation, together with its argument types, and its return types.*

Signatures usually correspond in component-based frameworks (*e.g.*, .NET or J2EE) to operation profiles described using an Interface Definition Language (IDL). For instance, WSDL is the accepted standard in the Web services area.

Additionally, we propose that behavioural interfaces are represented by means of it *Symbolic Transition Systems* (STSs). In this paper, STSs are Labelled Transition Systems (LTSs) extended with value passing (data parameters coming with messages). Communication between services is represented using events relative to the emission (!) and reception (?) of messages corresponding to operation calls. Events may come with a set of data terms whose types respect the operation signatures.

Definition 2 (Label)**.** *In our model, a label is either the internal action* τ[1] *or a tuple* (M,D,PL) *where M is the message name, D stands for the direction (!, ?), and PL is either a list of data terms, if the message corresponds to an emission, or a list of variables, if the message is a reception.*

Definition 3 (STS)**.** *A Symbolic Transition System (STS) is a tuple* (A,S,I,F,T) *where: A 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,* $F \in S$ *are final states, and* $T \in S \times A \times S$ *is the transition function (see [22] for semantic aspects).*

This formal model has been chosen because it is simple, graphical, and can be easily derived from existing implementation platforms languages, see for instance [19, 34, 18, 13] where such abstractions for Web services were used for verification, composition or adaptation purposes. In some cases, for conciseness reasons, a textual notation is better than a graphical one. Thus, a process algebra with value passing (such as value-passing CCS or LOTOS) could be used as a higher level language to specify behavioural interfaces. STSs can be automatically obtained from these processes using the operational rules of the process algebra semantics. In the following we will describe service interfaces using STSs only. Signatures will be left implicit, yet they can be inferred from the typing of arguments (made explicit here) in STS labels.

Example. We will use throughout this paper a car rental service as running example. First of all, let us present the service behavioural interface, and an example of end-user requirements also

[1] An internal action stands for an abstraction of the service behaviour. This is used to encode in the service protocol a (non-deterministic) local choice, or an internal computation.

specified using an STS (Fig. 2.1). Service CarRental can receive a car rental request (request?), and returns availability of the specified car for the given dates (request!). Such a request can be received and replied several times until termination (timeout in the service) or reservation (book?). In the latter case, the service confirms the reservation (book!) by sending back an identifier. In this example, the user requirements start with the search of a car (searchCar!). Then, some dates are submitted to check availability of a car for this period (searchDate!). After reception of an answer (reply?), the user can give up (quit!), can submit another search (searchCar!, searchDate!), or can reserve the car (resp. reserve! and confirmation with reserve?).

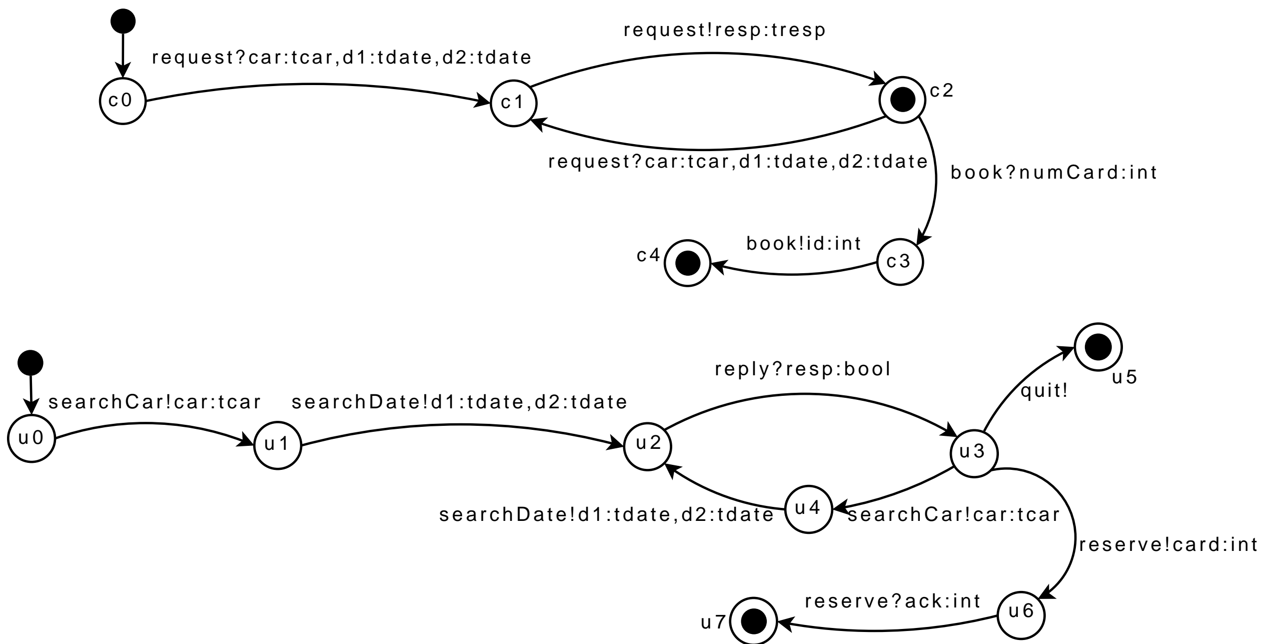

Figure 2.1: STS interfaces of CarRental (top) and User (bottom) services

Chapter 3

Compatibility and Mismatch

3.1 Compatibility Definition

In this subsection, we define the concept of compatibility we rely on in this paper. Compatibility depends on the model of entities, and numerous compatibility notions have already been proposed for automata-based descriptions of services, see for instance [36, 14, 20, 10, 31, 5]. Most of them match all visible labels (labels which are different of τ), and therefore rely on bisimulation-based notions [35]. However, these definitions are too restrictive in some situations. As an example, when two services meet on the Web, they may be able to interact correctly even when one has slots for receptions which the other one does not intend to use. Thus, in this paper, we took inspiration from the definition originally stated by Brand and Zafiropulo [7] who consider two protocols compatible if any emitted message has a matching reception in the mate service, and this works in both directions. Consequently, this definition accepts *unspecified receptions* that are additional receptions which do not have any corresponding emission in the other component. This notion means that services are able to cooperate in a satisfactory way even if they have additional slots for receptions. Thus, services can provide more than required by their partners, but they must be ready to accept all the messages sent by them.

We define below compatibility of two services by setting successively compatibility of labels, states, and finally protocols.

Definition 4 (Label compatibility)**.** *Two labels l_1 and l_2 are compatible if they have the same message names, opposite directions, and matching parameters, that is both lists of parameters have the same number of elements, and each pair of parameters have identical types.*

Definition 5 (State compatibility)**.** *Given two Symbolic Transition Systems $STS_1 = (A_1, S_1, I_1, F_1, T_1)$ and $STS_2 = (A_2, S_2, I_2, F_2, T_2)$, two states $s_1 \in S_1$ and $s_2 \in S_2$ are compatible if:*

- *either both states are initial (resp. final) or none of them is, and*
- *for each transition t outgoing from state s_1 (resp. s_2) whose label l corresponds to an emission, there is in the other state an outgoing transition t' with label l', and both labels l and l' are compatible wrt. Def. 4.*

Two transitions with matching labels are called *complementary*. A pair of states belonging to two STSs is said to be *reachable* if from their respective initial states, there exists a path with complementary transitions that lead to this couple of states.

Definition 6 (Protocol compatibility)**.** *Two Symbolic Transition Systems are compatible if their initial states are compatible, all pairs of states reachable from their initial states are compatible, and both*

STSs end up in final states. In addition, compatibility imposes that all transitions holding an emission have a complementary reception.

Example. Let us illustrate our notion of compatibility using a simplified version of the CarRental service (Fig. 3.1) introduced in Chapter 2. The left hand side protocol can receive and reply requests for both cars and motorbikes whereas the user requirements on the right hand side can only submit requests for cars. These two protocols are compatible *wrt.* Def. 6 stated before, because all emissions have matching labels in the partner (for both services) , and only the reception reqMoto?info:tinfo has no matching emission.

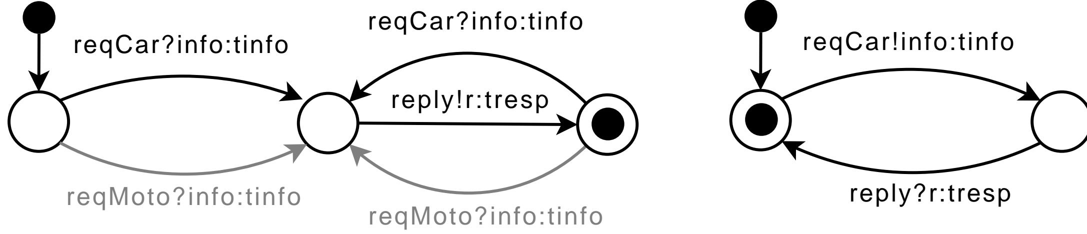

Figure 3.1: Example of compatible STS interfaces

3.2 Behavioural Mismatches

In this subsection, we give an overview of all mismatches that can occur between two service protocols specified using our model *wrt.* the compatibility definition presented in Def. 6. We organize mismatch cases following the three different levels in which they can arise, namely labels, states, and protocols.

A pair of labels may present the following mismatches:

- label type (lm_1): an input/output label (message, direction, parameters) *vs.* a τ action;
- label direction (lm_2): labels with same directions, since matching labels must have opposite directions;
- label name (lm_3): labels with different message names;
- parameter number (lm_4): unmatching number of parameters;
- parameter order (lm_5): wrongly ordered parameters;
- parameter type (lm_6): parameters with different types.

The three kinds of mismatch for parameters focus on different aspects, and therefore neither of them overlaps with the others.

A couple of states may present the following mismatches:

- state nature (sm_1): states with different natures, the nature of a state being either initial, final, or intermediate;
- outgoing transitions (sm_2): all emissions must be matched;

- incoming transitions (sm_3): all emissions must be matched.

As regards the comparison of outgoing (resp. incoming) transitions, a mismatch occurs *wrt.* the compatibility notion defined in Section 3.1 when there are more emissions on one side than receptions on the other. Indeed, the compatibility definition states that every emission must have a reception as counterpart.

A couple of protocols may present the following mismatches:

- additional states (pm_1);
- additional transitions (pm_2);
- order of transitions (pm_3): not respected in both protocols.

At the level of the protocol, especially in the last point mentioned above, a reordering mismatch[1] is detected by taking into account the protocol but also the labels associated to the involved transitions.

Last, let us point out that label and state mismatches are detected while computing the similarity measure (Chapter 4) whereas protocol mismatches are more difficult to find out at this stage. They will be only discovered when analysing similarity results (Chapter 5).

Example. Our running example presents various cases of mismatch at the three levels mentioned before. For instance, message names do not match, *e.g.,* book in CarRental *vs.* reserve in User ($lm3$), or requests sent by the user are split into two messages searchCar and searchDate whereas CarRental expects a single one, request ($pm1$, $pm2$). Consequently, parameters of these labels present also some differences ($lm4$, $lm5$, $lm6$). Note also the explicit termination quit in User which has an implicit counterpart in CarRental, state c2 ($pm1$, $pm2$).

We will see in the next chapter how similarity is computed on protocols and these mismatches are automatically detected.

[1]Reordering of transitions is needed in some communication scenarios to ensure a correct interaction when two communicating entities have transition labels which are not ordered as required.

Chapter 4

Measuring Protocol Similarity

In this chapter, we will present our techniques to measure similarities of two service interfaces described as $STS_1 = (A_1, S_1, I_1, F_1, T_1)$ and $STS_2 = (A_2, S_2, I_2, F_2, T_2)$. We choose a *divide-and-conquer* approach, therefore our similarity measure is split into comparisons of states, labels, depth and graphs. We will show how these four elements are compared, and how an overall measure is computed from them. In the following, we distinguish *graphs* and protocols. Graphs focus on the structure of the protocol, *i.e.,* states without nature associated (all are considered as intermediate) and transitions without labels, whereas the protocol corresponds to the whole STS. We will introduce our prototype tool that automates completely the processing, and illustrate our similarity measure on our running example. In the following, each similarity measure (on states, labels, etc) belongs to [0,1]. We start computing scores by assuming that all elements perfectly match (initial similarity score equals to 1). Then, values are decremented every time a mismatch is encountered[1]. Our similarity computation relies on generic weights written using identifiers starting by W (see Section 4.6 for instantiations of these weights).

4.1 State Similarity

Let us start with the similarity measure (*simS*) between two states $s_i \in S_1$ and $s_j \in S_2$. Function *simS* contains two processings. First, it compares state nature and penalises the measure by weight Wsm_1 if their nature is different (function *sType*). Second, function *emDif* computes the number of emissions outgoing from state s_i (resp. from s_j) which have no corresponding reception in state s_j (resp. in s_i). Next, this value is multiplied by Wsm_2, and the state measure is finally decremented by this result. Note that incoming transitions are compared in a similar way to outgoing transitions. Nevertheless, although they have been taken into account in our prototype tool presented in Section 4.5, we do not introduce them in this paper for simplification purposes.

$$\text{where } sType(s_i, s_j) = \begin{cases} 0 & \text{if}(s_i \in I_1 \wedge s_j \in I_2) \vee (s_i \in F_1 \wedge s_j \in F_2) \\ & \vee (s_i \notin I_1 \cup F_1 \wedge s_j \notin I_2 \cup F_2) \\ 1 & \text{otherwise} \end{cases}$$

$$emDif(s, s') = \begin{cases} 0 & \text{if} card(em(s)) \leq card(rec(s')) \\ abs(card(em(s)) - card(rec(s'))) & \text{otherwise} \end{cases}$$

[1] Our computations may make similarity scores become negative. In this section, for readability purposes, we will not make explicit in the formulas the function that sets back to 0 negative measures.

$em(s) = \{l|(s,l,s') \in T \wedge dir(l) = !\}$, $rec(s) = \{l|(s,l,s') \in T \wedge dir(l) = ?\}$, *abs* computes the absolute value, *card* returns the set cardinality (we use this function to compute list cardinality as well in the following), and *dir* indicates the direction of a label.

4.2 Label Similarity

The similarity measure between two labels $l_i \in A_1$ and $l_j \in A_2$ takes four cases into account. First, two labels with τ perfectly match. Second, if only one label is a τ action, the result is penalised using weight Wlm_1. Third, if directions are the same, we decrement the initial label measure by weight Wlm_2, and we do not compare message names and parameter lists. Last, if labels have opposite directions we compare the message names (function *sem* with weight Wlm_3) and parameter lists (functions *pNb*, *pOrd*, *pType* with respective weights Wlm_4, Wlm_5, Wlm_6).

$$simL(l_i,l_j) = \begin{cases} 1 & \text{if}(l_i = \tau) \wedge (l_j = \tau) \\ 1 - Wlm_1 & \text{elif } (l_i = \tau) \oplus (l_j = \tau) \\ 1 - Wlm_2 & \text{elif } dir(l_i) = dir(l_j) \\ 1 - (Wlm_3 * sem(m_i,m_j)) - (Wlm_4 * pNb(tl_i,tl_j)) & \\ \quad -(Wlm_5 * pOrd(tl_i,tl_j)) - (Wlm_6 * pType(tl_i,tl_j)) & \text{else} \end{cases}$$

where $\oplus$ stands for the XOR operator, m_i and m_j are message names of l_i and l_j, tl_i and tl_j are type lists extracted from parameter lists of l_i and l_j. Function *sem* uses the WordNet::Similarity package [30] to measure the semantic similarity of message names. Particularly, we chose the *Hirst & St-Onge* measure which works by finding lexical chains linking the two word meanings.

As regards parameter lists, function *pNb* returns the difference between the element number in both parameter lists:

$pNb(tl_i,tl_j) = abs(card(tl_i) - card(tl_j))$

Function *pOrd* returns the number of unordered parameter types in both lists:

$$pOrd(tl_i,tl_j) = \begin{cases} pOrd'(tl_i,tl_j) & \text{if } card(tl_i) \leq card(tl_j) \\ pOrd'(tl_j,tl_i) & \text{otherwise} \end{cases}$$

$$pOrd'(tl_i,tl_j) = cmp(0,tl_i,tl_j) + \ldots + cmp(n,tl_i,tl_j)$$

where n is the length of the smallest list tl_i. Function *cmp* returns 1 if the kth parameter type in tl_i belongs to the tl_j list[2], and if types do not appear in the same positions in the lists:

$$cmp(k,tl_i,tl_j) = \begin{cases} 1 & \text{if}(tl_i[k] \neq tl_j[k]) \wedge (tl_i[k] \in tl_j) \\ 0 & \text{otherwise} \end{cases}$$

Finally, function *pType* counts the number of types that appear in one list but not in the other.

[2] The presence of the type in the other list is checked here because such mismatch is considered as a type number issue (see function *pType*), and not as an order problem.

Function $typeDif$ computes the set of types that belong to the first list but not to the second one. We call $typeDif$ twice to detect additional types in both lists.

$pType(tl_i, tl_j) = card(typeDif(tl_i, tl_j)) + card(typeDif(tl_j, tl_i))$
$typeDif(tl_1, tl_2) = \{type | type \in tl_1 \wedge type \notin tl_2\}$

4.3 Depth and Graph Similarity

In order to compare protocol structures, we take into account two measures, namely depth and graph similarity.

Depth measures the number of transitions to be traversed to reach a state from the initial one. To avoid infinite values for depth due to looping protocols, we compute this measure by searching the shortest path (function $depth$ takes inspiration from Dijkstra's algorithm [16]). A pair of states (s_i, s_j) is compared it *wrt.* depth similarity using the following formula:

$$simD(s_i, s_j) = 1 - (Wdepth * abs(depth(s_i) - depth(s_j)))$$

As regards graph similarity, the idea is to compare how similar are two states *wrt.* their positions in both graphs. This measure is computed by reusing the approach proposed in [4]. In this paper, the authors compute a similarity matrix (function *simG*) between vertices of two directed graphs (denoted by G_1 and G_2 in the following, which are derived from STS_1 and STS_2 respectively). To do so, they start with a matrix whose entries are all equal to one. Next, they apply several iterations and refine similarities scores by using adjacency matrices. The processing stops when matrix values converge towards a limit. More formally, *simG* is computed using the following formula:

$$simG_{k+1}(G_1, G_2) = \frac{(AM_1 \otimes AM_2 + AM_1^T \otimes AM_2^T) \times simG_k(G_1, G_2)}{\| (AM_1 \otimes AM_2 + AM_1^T \otimes AM_2^T) \times simG_k(G_1, G_2) \|}$$

where k denotes successive iterations, AM_i is the adjacency matrix for graph G_i, M^T is the transposed matrix, $\otimes$ computes the tensorial product, $+$ (resp. $\times$) returns the sum (resp. product) of two matrices, and $\|\|$ computes the Euclidian norm.

The limit used to stop iterations is computed from the initial similarity matrix (all entries equal to one), denoted [1] below, as follows:

$$simG_lim(G_1, G_2) = lim_{k \longrightarrow \infty} \frac{(AM_1 \otimes AM_2 + AM_1^T \otimes AM_2^T)^{2k} \times [1]}{\| (AM_1 \otimes AM_2 + AM_1^T \otimes AM_2^T)^{2k} \times [1] \|}$$

The reader interested in more details may refer to [4].

4.4 Protocol Similarity

In the former subsections, we have presented how similarities are computed on the different constituents of our STSs. Here, we present our method for computing the overall similarity measure (function *sim*) which results in a matrix $S_i \times S_j$. Each entry of this matrix corresponds to a similarity

score for a couple of states (s_i, s_j) obtained by computing the average of the different measures introduced before. More precisely, each score respectively takes into account state similarity (with weight Ws), similarity of all labels associated to transitions outgoing from states s_i and s_j (with weight Wl), and depth and graph similarity (with weight Wdg). Similarly to Section 4.1, incoming transitions are not made explicit in the following formulas.

$$sim(s_i, s_j) = \frac{(simS(s_i, s_j) * Ws) + (simLs(s_i, s_j) * Wl) + (simDG(s_i, s_j) * Wdg)}{Ws + Wl + Wdg}$$

where *simLs* and *simDG* are defined below.

Function *simLs* considers all associations of transition labels outgoing from states s_i and s_j, and computes the average of their corresponding similarity measures obtained using function *simL*.

$$simLs(s_i, s_j) = \frac{(\forall (l, l') \in L_1 \times L_2) \sum simL(l, l')}{card(L_1) * card(L_2)}$$

where $L_1 = outlabels(s_i, T_1)$, $L_2 = outlabels(s_j, T_2)$, and $outlabels(s, T) = \{l | (s, l, s') \in T\}$.

Function *simDG* computes the average for depth and graph similarity as follows:

$$simDG(s_i, s_j) = \frac{simD(s_i, s_j) + mG[s_i, s_j]}{2}$$

where $mG = simG(G_1, G_2)$.

4.5 Prototype Tool: ITACA::Sim

We have implemented our approach in a prototype tool called ITACA::Sim. This tool accepts as input two XML files corresponding to service interfaces, and returns the detailed measure (states, labels, depth, and graph) for each couple of states as well as the matrix giving the overall similarity measure (as shown in Table 4.1). Note that for each couple of states, our tool also returns a list of mismatches (identified as presented in Section 3.2).

ITACA::Sim is implemented in Python, and uses the Numerical Python library (NumPy) to handle matrices. The tool also employs the external package WordNet::Similarity implemented in Perl for measuring the semantic similarity between message names. In the first experiments we carried out, we applied *ITACA::Sim* on several examples such as a memory manager, an SQL server, an e-commerce system, etc.

4.6 Application to our Case Study

In the former subsections, we have presented our measure using generic weights. We will show in this subsection how these weights can be instantiated. Their instantiation has been obtained by many experimentations on examples, and by respecting the following rules. As far as labels are concerned, mismatch of label direction should be strongly penalised ($Wlm2$) whereas message name ($Wlm3$) is not considered as an important one. Label parameters are essential ($Wlm4$, $Wlm5$, $Wlm6$). However, parameter weights should not be too high because these mismatches are cumulative and takes the number of parameters into account, therefore penalisation will increase with the number of parameters. Last, while computing the overall similarity result, we should give more importance to

	u0	u1	u2	u3	u4	u5	u6	u7
c0	0.68	0.53	0.28	0.37	0.46	0.26	0.23	0.23
c1	0.20	0.31	0.77	0.18	0.26	0.26	0.71	0.21
c2	0.58	0.54	0.36	0.58	0.53	0.45	0.29	0.41
c3	0.15	0.26	0.76	0.21	0.29	0.32	0.87	0.27
c4	0.20	0.23	0.34	0.17	0.34	0.51	0.41	0.47

Table 4.1: Similarity measure between CarRental and User interfaces

labels (Wl) than states and graphs (Ws and Wdg) because services interact through labels and then the semantics of an STS derives mainly from them.

The similarity results obtained on our running example have been computed with the following concrete weights: $Wsm1 = 0.3$, $Wsm2 = 0.2$, $Wlm2 = 0.95$, $Wlm3 = 0.1$, $Wlm4 = 0.1$, $Wlm5 = 0.1$, $Wlm6 = 0.1$, $Wdepth = 0.2$, $Ws = 1$, $Wl = 2$, and $Wdg = 1$. Table 4.1 shows the matrix giving the global similarity measure obtained for our running example. This measure allows to point out mismatches between both CarRental and User protocols as well detecting parts of them which turn out to be similar. Let us comment some similarity scores returned by our tool. For instance, state u0 has a high similarity value with c0, and ITACA::Sim returns the following measures:

```
states c0 (carRental-v7) u0 (user-v7):
            States:  (1.0 [])
            Labels:  (request?car:tcar,d1:tdate,d2:tdate searchCar!car:tcar 0.6
                      ['LM3', 'LM4', 'LM6'])  ->  Labels average:  0.6
            Protocol:  Depth [0] [0] 1.0   Graph 0.0621   -> Average:  0.53
            Total:  0.68
```

We can see that states have the same nature and matching outgoing transitions (one emission on one side and one reception on the other). At the label level, there are three mismatches, respectively different message names ($lm3$), different number of parameters ($lm4$ — two more parameters are expected by the car rental service), and some types (tdate) that do not exist in both parameter lists ($lm6$). Depth is the same, and the measure returned by graph similarity is pretty low. There are two explanations: first, the algorithm used for this measure always returns low values; second, the algorithm does not use initial and final states, and experiments show that it always encounters difficulties to match them correctly.

State u0 has its lower value with state c3, and this is the smallest value of the whole table:

```
states c3 (carRental-v7) u0 (user-v7):
            States:  (0.3 ['SM1', 'SM2'])
            Labels:  (book!id:int searchCar!car:tcar 0.05 ['LM2'])
                      -> Labels average:  0.05
            Protocol:  Depth [3] [0] 0.4   Graph 0.0207   -> Average:  0.21
            Total:  0.15
```

For this couple of states, they first present nature mismatch ($sm1$), but also mismatches at the level of outgoing transitions ($sm2$) because on both sides transitions involve emissions which are not matched by the partner. Labels are not compared since they present same directions ($lm2$). Last, there is a low depth value (difference of three between both states), and graph similarity measure is low as well.

Chapter 5

Applications: Adaptation and Re-engineering

5.1 Software Adaptation

Software Adaptation [3] is a promising solution to compose in a non-intrusive way black-box components or (Web) services whose functionality is as required for the new system, although they present interface mismatches. Adaptation techniques aim at automatically generating new components called *adaptors* from an *adaptation contract* which is an abstract description of how mismatches can be worked out. All the messages pass through the adaptor which acts as an orchestrator, and this enables it to compensate mismatches by matching exchanged information (messages, arguments, etc) as defined in the contract. As a result, in spite of the mismatches, the adaptor makes the involved services work correctly together.

Adaptor generation techniques rely on the adaptation contract. However, most of the approaches (see for instance [6, 11, 17, 27]) require the contract to be written, which induces that a designer has to understand the subtleties of the services at hand, and specify manually how services interact and how existing mismatches can be corrected. In this section, we will sketch some ideas on how by analysing the results computed by our similarity function we can automate the generation of the adaptation contract.

Since our model deals with value passing, we have to consider an adaptation contract notation that maps not only messages but also data parameters. Thus, we rely on the vector-based notation advocated in [6] and reused for instance in [25]. A vector involves at most one label from each service at hand, and results in an interaction between them. Sometimes, a vector contains a single label when this label does not have any counterpart in the partner. This corresponds to an independent evolution of this service. Last but not least, being given an adaptation contract specified using this notation, approaches as those proposed in [6, 25] can be used to generate automatically adaptor protocols.

Let us illustrate on our running example how the similarity results can be used for adaptation purposes. The analysis step aims at relating states in both protocols: for each service, each state is associated to a state in the partner protocol whose similarity is the highest. Then, from Table 4.1 we obtained correspondences in Figure 5.1 where we can notice that some states match in both directions (solid lines), that is from the user point of view but also from the car rental one, and the others states match only in one direction (dashed line).

In a second step, we consider couples of states connected with solid lines, and for each couple we build vectors that match labels associated to transitions outgoing from these states. In case if several associations of labels are possible (*it e.g.,* two transitions outgoing from c2 and three transitions outgoing from u3, see Figure 2.1), the label similarity is used (see Section 4.2), and only best matches

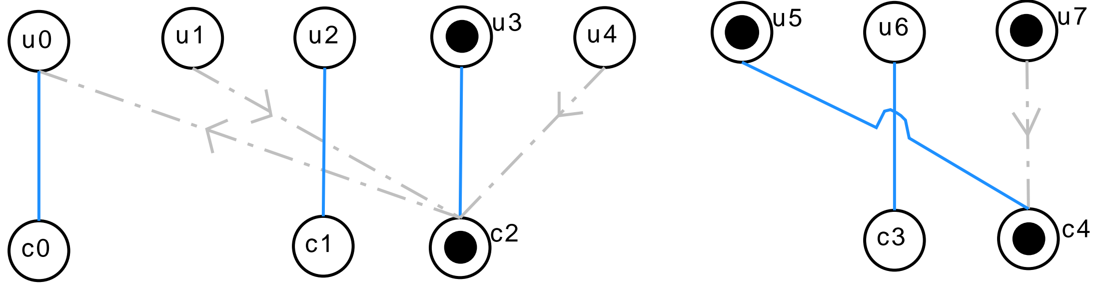

Figure 5.1: Analysis and matching the states of our running example

are preserved. Vectors introduced below can be generated automatically from our similarity measure and its analysis. Multiple occurrences of the same vector are detected and removed (v_{sc} below).

(u0,c0) $\leadsto$ $v_{sc} = \langle$User:searchCar! ; CarRental:request?$\rangle$
(u2,c1) $\leadsto$ $v_{repl} = \langle$User:reply? ; CarRental:request!$\rangle$
(u3,c2) $\leadsto$ ~~$v_{sc} = \langle$User:searchCar! ; CarRental:request?$\rangle$~~
$v_{res1} = \langle$User:reserve! ; CarRental:book?$\rangle$
$v_{quit} = \langle$User:quit!$\rangle$
(u5,c4) $\leadsto$ *no outgoing transitions*
(u6,c3) $\leadsto$ $v_{res2} = \langle$User:reserve? ; CarRental:book!$\rangle$

Next, the message searchDate! in the user does not have a clear correspondence (see u1 and u4 in Figure 5.1), then a vector with this label is generated: $v_{sd} = \langle$User:searchDate!$\rangle$. Last, the contract relates messages but also parameters by using placeholders. The idea is that for each vector, parameters with same type are related using the same placeholder name, and respecting their order in the list if there are several parameters with the same type (see v_{sc} below where request? in the CarRental involved two parameters of type tdate). Here are the previous vectors extended with parameter matching:

$v_{sc} = \langle$User:searchCar!C ; CarRental:request?C,D1,D2$\rangle$
$v_{repl} = \langle$User:reply?T ; CarRental:request!B$\rangle$ $\leadsto$ *type conversion needed*
$v_{res1} = \langle$User:reserve!I1 ; CarRental:book?I1$\rangle$
$v_{quit} = \langle$User:quit!$\rangle$
$v_{res2} = \langle$User:reserve?I2 ; CarRental:book!I2$\rangle$
$v_{sd} = \langle$User:searchDate!D3,D4$\rangle$ $\leadsto$ $v_{sd} = \langle$User:searchDate!D1,D2$\rangle$

Let us give some comments on the contract obtained above. First, in the very final step, a manual adjustment is needed to make dates in v_{sc} and in v_{sd} match in spite of the initial erroneous matching. Last, in some cases (v_{repl}) some type mismatch prevent to match some message parameters, and this is still an open issue in software adaptation.

5.2 Software Re-engineering

As we have seen in the former subsection, adaptation is a promising solution to compose black-box entities. However, the process is still quite complicated and adaptor generation is costly (algorithms are exponential [11, 32]). Thus, we will see in this subsection a second application of our similarity measure, namely software re-engineering [2], which is an alternative to service adaptation.

Re-engineering assumes entities' code is accessible, and such assumption, yet rather strong, makes sense in the area of software components [9, 21, 33]. Indeed, no Interface Description Language tackling more than the signature level has never become widely accepted, and the designer does

not have at his disposal a sufficiently expressive description of the component during its integration. Accordingly, the black-box assumption is not always meaningful, and white-box components might be assumed while building systems to allow a better understanding of new entities being integrated.

Software re-engineering aims at modifying the code of one or several components to be reused for a new system under construction in order to solve existing mismatches and make them compatible. This leads one to rely on a formal definition of compatibility, and automate as most as possible the code's modifications. As far as services are concerned, re-engineering consists in modifying user requirements since services are already deployed and cannot be modified. The main limitation of re-engineering is the validation need that becomes mandatory due to the code's component modification.

Let us illustrate on our running example how re-enginering can be guided by using our similarity measure. The final goal is to make both protocols compatible *wrt.* Def. 6. Since we are dealing with services in this paper, we will apply modifications only on the user requirements. Note that the method should indicate modifications of both the interface and the code corresponding to it.

The re-engineering process relies on Figure 5.1 and works in two steps. First, states in the user without clear matching with the car rental, namely u1 and u4, are removed (u7 is kept because no transition goes out from this state). Second, for each couple of states strongly related in Figure 5.1, we focus on labels appearing on outgoing transitions, and we use mismatch lists generated by our tool and presented in Section 3.2 to modify them.

As an example, state u0 matches with state c0. These two states only have one outgoing transition. However, the mismatch list indicates different message names (*lm*3), and a different number of parameters (*lm*4) as well as missing types (*lm*6). Therefore, searchCar is renamed in request, and two parameters with type tdate are added. Let us consider now state u3 where three transitions go out from this state. By using values returned for label comparison, we can deduce that searchCar matches with request, consequently the same re-engineering that presented for state u0 has to be applied. Next, reserve matches with book, and the only re-engineering needed here is that message names have to be unified. Last, quit does not have any counterpart in the CarRental service, therefore this transition (and its target state u5) is cut away. As a result, the interface obtained after re-engineering is described in Figure 5.2.

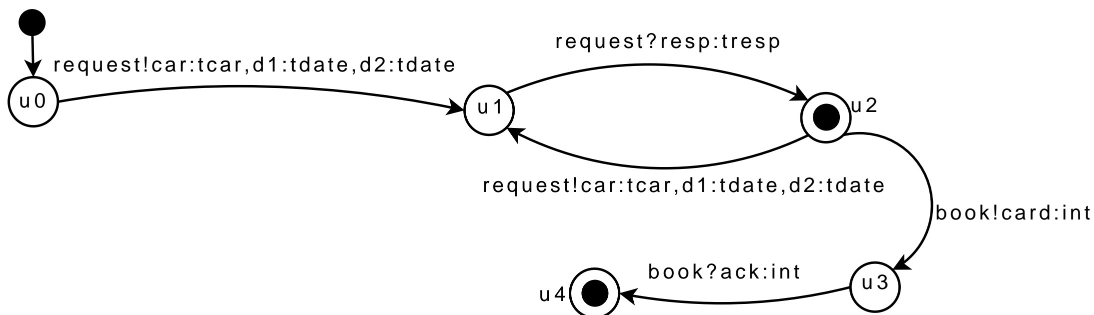

Figure 5.2: User requirements protocol obtained after re-engineering

Note that the re-engineering process ensures that user requirements and the service are compatible. However, this process yet automated still needs a human validation to be sure that the final objective of the user is preserved from a semantic point of view. Indeed, modifications performed during the re-engineering stage may not respect this objective (*e.g.,* by cutting too many transitions). A way to automate this would be to add a composition goal (using a temporal formula for example) as input to the process, and this goal would guide the re-engineering steps.

Some re-engineering steps may be quite complicated, especially when type re-engineering is required as it is the case when the user receives the reply to his request (request?) since in this situation

parameter type has to be adjusted to make labels match correctly (tresp instead of bool before). Last, re-engineering does not always generate a mirror protocol as it is the case with our running example: some additional receptions may appear and be maintained to respect the compatibility definition.

Chapter 6

Related Work

The contribution of this work fall into two categories: protocol compatibility and measuring the similarity of service protocols *wrt.* to our compatibility definition.

Compatibility is an issue intensively studied in Software Engineering, and areas such as software components [36, 10, 14], coordination [8, 37], or (Web) services [26, 24, 5], in which the authors have used Petri nets, process algebra or automata-based formalisms to describe interfaces. However, in most of cases, these approaches return a Boolean that indicates whether two interfaces are compatible. Our goal here was more ambitious since we presented an approach giving a measure of similarity between two service interfaces described using protocols with value passing. More than computing protocols' similarity, our approach identifies the set of differences encountered between service protocols. Being aware about these differences eases the adaptation process as well as the system reconfiguration, *i,e* re-engineering.

In the field of Case Based Reasoning (CBR), Champin and Solnon [12] address the similarity measure of labeled graphs by searching a best mapping (correspondence) between graphs vertices. Their approach relies on Tversky's formula [1] which aims, in the field of psychology, at measuring the similarity between two objects by comparing the amount of features which are common to both objects, to the total amount of their features. In [12], the authors consider a much simpler model, for instance value passing is not taken into account. Furthermore, they target as application area CBR that consists in solving new problems based on the solutions of similar problems, whereas our purpose is tackling compositional issues while constructing software by reuse of existing services or components.

To the best of our knowledge, the work the most related to ours is the recent paper by Nejati *et al.* [28]. In this paper, the authors present two operators, namely *match* and *merge*, to be applied to specifications described using Statecharts. Therefore, the goal of their work is to compare Statecharts to merge them in a second step, whereas we focus on the compatibility issue between similar models. As far as similarity computation is concerned (referred as *match* in their work), although we share some similarities with them, such as the use of the WordNet::Similarity package for message comparison, our proposal differs in several aspects. First, they employ a depth measure as we do, but they do not further compare protocol structures whereas we reuse a recent graph similarity algorithm [4] to this purpose. As regards label comparison, beyond semantic comparison of messages, we additionally compare parameter types (see Section 4.2). Last, our similarity measure is supported by a tool we have implemented.

Chapter 7

Concluding Remarks

In this paper, we have presented a similarity measure between a couple of protocols. More than stating whether two interfaces are compatible or not, our approach computes the similarity of all states involved in both protocols. In a second step, such a result can be used to ease their composition by relating similar states and pointing out protocol differences. Thus, at last, mismatch cases can be corrected by using an intermediate adaptor which compensates mismatches, or by re-engineering interfaces to make them compatible. The computation of our similarity measure is completely automated by the ITACA::Sim tool we implemented. Our tool is currently used as an external module in the context of other research to help the construction of adaptation contracts (by indicating best correspondences between states and labels) through a graphical interface.

Perspectives of this work are the following. First of all, we plan to use machine learning in our experiments to compute concrete weights on which rely our similarity measure. Another perspective aims at refining our similarity measure by improving label comparison (*e.g.,* keeping best matches, or considering cyclic comparison of parameters), and experimenting other graph similarity algorithms.

As an alternative, to our *divide-and-conquer* approach, we would like to study the computation of similarity by taking states, labels, and protocols all together into account at the same time (*i.e.,* traversing the protocol and comparing states and labels). To make that more concrete, we are thinking about several alternatives. A first alternative could be an implementation of an *ad-hoc* algorithm. Then, it seems promising to reuse existing approaches and adapt them to our problem. Actually, we are studying two existing approaches namely the probabilistic processes, *i,e* Markov processes [15] and the partitioning algorithms [23, 29].

Finally, we plan to work thoroughly on the application and use of our similarity results for software adaptation and re-engineering.

Bibliography

[1] T. Amos. Features of Similarity. *Psychological Review*, 84:327–352, 1977.

[2] R. S. Arnold. *Software Reengineering*. IEEE Computer Society Press Los Alamitos, 1993.

[3] S. Becker, A. Brogi, I. Gorton, S. Overhage, A. Romanovsky, and M. Tivoli. Towards an Engineering Approach to Component Adaptation. In *Architecting Systems with Trustworthy Components*, volume 3938 of *Lecture Notes in Computer Science*, pages 193–215. Springer-Verlag, 2006.

[4] V. D. Blondel and P. Van Dooren. Similarity Matrices for Pairs of Graphs. In *Proc. of the 30th International Colloquium on Automata, Languages and Programming (ICALP'03)*, volume 2719 of *Lecture Notes in Computer Science*, pages 739–750. Springer, 2003.

[5] L. Bordeaux, G. Salaün, D. Berardi, and M. Mecella. When are Two Web Services Compatible? In *Proc. of 5th Int. Workshop on Technologies for E-Services (TES'04)*, volume 3324 of *Lecture Notes in Computer Science*, pages 15–28. Springer-Verlag, 2004.

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

[7] D. Brand and P. Zafiropulo. On Communicating Finite-State Machines. *J. ACM*, 30(2):323–342, 1983.

[8] A. Brogi, E. Pimentel, and A. M. Roldán. Compatibility of Linda-based Component Interfaces. In *Proc. of workshop on Formal Methods and Component Interaction (FMCI'02)*, volume 66(4) of *Elec. Notes on Theor. Comput. Science*, 2002.

[9] M. Buchi and W. Weck. The Greybox Approach: When Blackbox Specifications Hide Too Much. Technical Report 297, Turku Center for Computer Science, 1999.

[10] C. Canal, E. Pimentel, and J. M. Troya. Compatibility and Inheritance in Software Architectures. *Sci. Comput. Program.*, 41(2):105–138, 2001.

[11] C. Canal, P. Poizat, and G. Salaün. Synchronizing Behavioural Mismatch in Software Composition. In *Proc. of the 8th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'06)*, volume 4037 of *Lecture Notes in Computer Science*, pages 63–77. Springer-Verlag, 2006.

[12] P.-A. Champin and C. Solnon. Measuring the Similarity of Labeled Graphs. In *Proc. of the 5th International Conference on Case-Based Reasoning (ICCBR'03)*, volume 2689 of *Lecture Notes in Computer Science*, pages 80–95. Springer-Verlag, 2003.

[13] J. Cubo, G. Salaün, C. Canal, E. Pimentel, and P. Poizat. A Model-Based Approach to the Verification and Adaptation of WF/.NET Components. In *Proc. of the 4th International Workshop on Formal Aspects of Component Software (FACS'07)*, Electronic Notes in Theoretical Computer Science (ENTCS) series. Elsevier, 2007. To appear.

[14] L. de Alfaro and T. Henzinger. Interface Automata. In *Proc. of the 8th European Software Engineering Conference held jointly with the 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'01)*, pages 109–120. ACM Press, 2001.

[15] J. Desharnais, V. Gupta, R. Jagadeesan, and P. Panangaden. Metrics for Labelled Markov Processes. *Theoretical Computer Science*, 318(3):323–354, 2004.

[16] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[17] M. Dumas, K. W. S. Wang, and M. L. Spork. Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation. In *Proc. of 4th International Conference on Business Process Management (BPM'06)*, volume 4102 of *Lecture Notes in Computer Science*, pages 65–80. Springer-Verlag, 2006.

[18] H. Foster, S. Uchitel, and J. Kramer. LTSA-WS: A Tool for Model-based Verification of Web Service Compositions and Choreography. In *Proc. of 28th International Conference on Software Engineering (ICSE'06)*, pages 771–774. ACM Press, 2006.

[19] X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *Proc. of the 13th International Conference on World Wide Web (WWW'04)*, pages 621–630. ACM Press, 2004.

[20] N. Hameurlain. Flexible Behavioural Compatibility and Substitutability for Component Protocols: A Formal Specification. In *Proc. of the 5th IEEE International Conference on Software Engineering and Formal Methods (SEFM'07)*, pages 391–400. IEEE Computer Society, 2007.

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

[22] A. Ingolfsdottir and H. Lin. *A Symbolic Approach to Value-passing Processes*, pages 427–478. Handbook of Process Algebra. Elsevier, 2001.

[23] P. C. Kanellakis and S. A. Smolka. CCS expressions finite state processes, and three problems of equivalence. *Inf. Comput.*, 86(1):43–68, 1990.

[24] A. Martens. On Compatibility of Web Services. *Petri Net Newsletter*, 65, 2003.

[25] R. Mateescu, P. Poizat, and G. Salaün. On-the-Fly Adaptation of Services with Value-Passing Protocols. Submitted to ASE'08.

[26] M. Mecella, B. Pernici, and P. Craca. Compatibility of E-services in a Cooperative Multi-platform Environment. In *Proc. of VLDB satellite workshop on Technologies for E-Services (TES'01)*, volume 2193 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2001.

[27] H. R. Motahari-Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-Automated Adaptation of Service Interactions. In *Proc. of the 16th International Conference on World Wide Web (WWW'07)*, pages 993–1002. ACM Press, 2007.

[28] S. Nejati, M. Sabetzadeh, M. Chechik, S. M. Easterbrook, and P. Zave. Matching and Merging of Statecharts Specifications. In *Proc. of 29th International Conference on Software Engineering (ICSE'07)*, pages 54–64. ACM Press, 2007.

[29] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987.

[30] T. Pedersen, S. Patwardhan, and J. Michelizzi. WordNet::Similarity - Measuring the Relatedness of Concepts. In *Proc. of the 19th National Conference on Artificial Intelligence, 16th Conference on Innovative Applications of Artificial Intelligence (AAAI'04)*, pages 1024–1025. American Association for Artificial Intelligence, 2004.

[31] F. Plasil and S. Visnovsky. Behavior Protocols for Software Components. *IEEE Transactions on Software Engineering*, 28(11):1056–1076, 2002.

[32] P. Poizat and G. Salaün. Adaptation of Open Component-based Systems. In *Proc. 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*, pages 141–156. Springer-Verlag, 2007.

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

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

[35] R. J. van Glabbeek. *The Linear Time - Branching Time Spectrum I*, chapter 1, pages 3–99. Handbook of Process Algebra. Elsevier, 2001.

[36] D. M. Yellin and R. E. Strom. Protocol Specifications and Component Adaptors. *ACM Trans. Program. Lang. Syst.*, 19(2):292–333, 1997.

[37] J. M. Zaha and A. Albani. Compatibility Test for Coordination Aspects of Software Components. In *Proc. of the 17th Australian Software Engineering Conference (ASWEC'06)*, pages 41–48. IEEE Computer Society, 2006.