Client Update: A Solution for Service Evolution

Meriem Ouederni, Gwen Salaün, Ernesto Pimentel

Client Update: A Solution for Service Evolution

Meriem Ouederni
University of Málaga, Spain
Email: meriem@lcc.uma.es

Gwen Salaün
Grenoble INP, INRIA, France
Email: gwen.salaun@inria.fr

Ernesto Pimentel
University of Málaga, Spain
Email: ernesto@lcc.uma.es

***Abstract*—In service-based systems, service evolution might raise critical communication issues since the client cannot be aware of the changes that have occurred on the black-box services side. In this paper, we propose an automated process to adapt the client to the changes that have occurred. Our approach relies on a compatibility measuring method, and changes the client interface to ensure the system compatibility. This solution is fully automated inside a prototype tool we have implemented.**

***Index Terms*—Web Services; Services Composition**

I. Introduction

Since the venue of Service-Oriented-Computing (SOC), companies tend to build their software systems using heterogeneous and loosely-coupled (Web) services. These services are black-box applications meaning that they can only be accessed — *i.e.*, discovered and composed — through their public interfaces, and their implementation details are hidden from an external point of view. Here, service interfaces describe the exchanged messages and their application order referred to as interaction protocol.

In a challenging business environment, companies' growth is driven by innovations and competitions which cause changes across companies' requirements. Thus, service providers need to constantly change their services to fulfill these new requirements. In particular, there might be a need to add or remove some functionalities. In the context of our work, we assume service-based systems which consist of two parties involved in the interaction — *i.e.*, the service and its client (user). In this setting, service changes are usually performed in a transparent manner such that the client is not aware aware of those changes. This can lead to system disruptions and in particular raise incompatibility issues between the upgraded service and its client. For instance, let us assume a system composed of a client C and a service S, in such a way, that both C and S are initially compatible. The client C can first log on to the service and wait for an acknowledgment. The service S is able to acknowledge a client every time the latter makes a logging request. We now assume that service S is changed, in such a way, that it requires the client to enter a secure code after a successful login request. This change has been made to stop malicious hackers accessing the service S. As a consequence, the future interaction between client C and service S deadlocks because the new request required by S cannot be satisfied by C.

In order to cope with the above interaction issues occurring in the context of service evolution, we should provide an answer to the following challenging question: how to solve system incompatibility? An obvious answer requires the detection and resolution of interface mismatches to guarantee the correct communication. Existing approaches in the field are split into two classes. The first class is devoted to service adaptation [3], [14], [5] which aims at implementing an intermediate service (called adaptor) to work out existing service/client mismatches. However, adaptor generation is an intricate process, and in some cases it is impossible to resolve interface incompatibility using such techniques, *e.g.*, parameter type conversion is not supported. The second class of existing approaches [18], [2], [17], [1] allows the designer to change service interfaces in order to ensure the system compatibility. These works do not consider service as black-boxes, and their techniques used to modify service interfaces consist of two steps: (i) They first detect changes by calculating all differences between the upgraded and old service version; (ii) Then, they propagate all these changes into the upgraded version — *i.e.*, adding and deleting some parts of the service interface. These works describe services using models which do not take internal behaviors and message parameters into account. They also use a compatibility check which does not allow one to detect the changes to be handled. Therefore, they cannot avoid the computation of differences between the updated and new service version.

This paper proposes a new approach for automatic resolution of interface mismatches in the context of service evolution. We consider black-box services that can be discovered and composed through their public interfaces. Internal behaviors and message parameters are considered in service interfaces. We only enable the designer to change the client in order to match the upgraded service interface. In our approach, the changes are detected using a compatibility measure which compares the service with its client *wrt.* a compatibility notion, and returns the mismatches presented between both interfaces. In order to update the client interface, we use this measure to compute an interface mapping tree which relates parts of both interfaces that better match with each other. Using the mapping tree and the mismatch list, we first resolve protocol-related interaction issues called behavioral mismatches. Then, we resolve the static interface mismatches which consider exchanged messages and their parameters. Our process completes when both the service and the client interfaces become compatible. Note that this evolution management process can be also parameterized with some user requirements in order to prevent undesirable behaviors (functionalities) that one does not want to appear in the new client interface.

The rest of this paper is organized as follows. Section II is an overview of related work. Section III presents the service description model. We give in Section IV our method for compatibility measuring and the interface mismatches considered. Our client update techniques are detailed in Section V. Section VI illustrates the use of our approach through a case study. Finally, Section VII concludes the paper and presents some perspectives.

II. Related Work

The resolution of interaction issues is a major concern of SOC to ensure the correct reuse and composition of services which change constantly. To the best of our knowledge, the evolution problem is related to fields such as database schema evolution, software component evolution, software refactoring, workflow evolution, and protocol evolution. Here, we focus on the service evolution area, but more details on evolution problems in the context of the above fields are presented in [18].

Existing work on service evolution, see for instance [18], [2], [17], [1], does not consider Web services as black-boxes, yet they allow access and changes their implementation details. In [17], [1], the authors differentiate between two kinds of changes: shallow changes where the effect of the changes is localized to a service or is strictly restricted to the clients of that service; and deep changes which extend beyond the clients of a service possibly to the clients of these service clients such as outsourcers or suppliers. The goal of [17], [1] is to guarantee the independent evolution of loosely-coupled services. However, this violates the black-box service assumption. Our solution also applies shallow changes, nevertheless we aim at updating the client (user) rather than the service. The work presented in [18] focuses on the management of dynamic evolution — *i.e.*, it deals with the effect of the service changes which occur when a system is running an old service version. More recently, [2] studied static evolution. This work proposes changing the client which can be a service. In our approach, services are considered as black-boxes which cannot be changed.

Techniques used in previous work first detect changes by computing the differences between the upgraded and the old service, and then propagate these changes into either the client or service depending on their context — *i.e.*, client-based or service-based changes. Our approach is different since we detect changes using our compatibility measure which gives a detailed comparison of an upgraded service with its client. This measure enables us to map the interface constituents in spite of existing mismatches, but it also allows us to resolve these mismatches. Lastly, the consideration of user requirements guarantees that the updated client interface does not describe any undesirable behavior.

III. Interface Model

This section presents the model of the service interfaces we use to check their compatibility. We assume that the service interface Int is described by means of a signature (Σ) and an interaction protocol represented by a *Symbolic Transition System* (STS). Formally, an interface Int is defined using the couple (Σ, STS). The interface signature represents the set of operation profiles which may be required and provided, including their argument types.

Definition 1 (Signature): A signature Σ is a set of provided and required operation profiles. An operation profile is defined as an operation name together with its input and output sorts, *i.e.*, type names, (possibly empty):

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

Our STS model is a variant of the STG (Symbolic Transition Graph) model presented in [12], where guards in branching transitions are abstracted into transitions labelled with τ actions. In an STS, communication between services is represented using *events* relative to the emission and reception of messages corresponding to operation calls. An event comes with a list of parameters (possibly empty) whose types respect the operation signature. A *label* describes either the (internal) τ action or an event using the tuple (m, d, pl) where m is the message name, d stands for the communication direction (either an emission ! or a reception ?), and pl is either a list of data terms if the label corresponds to an emission, or a list of variables if the label is a reception.

Definition 2 (STS): A Symbolic Transition System, or *STS*, is a tuple (A, S, I, F, T) where: A is an alphabet which corresponds to the set of labels associated to transitions, S is a set of states, $I \in S$ is the initial state, $F \subseteq S$ is a nonempty set of final states, and $T \subseteq S \backslash F \times A \times S$ is the transition relation.

It is worth noting that communication between services described with STSs relies on a synchronous and binary communication model[1]. The operational semantics of this model is given in [9]. STSs can also be easily derived from higher-level description languages such as Abstract BPEL, see for instance [10], [19], [6] where such abstractions were used for verification, composition or adaptation of Web services.

In the context of our work, as we have mentioned in the Introduction Section, we consider systems composed of two parties, namely, a service and a client (user). In the remainder of this paper, both parties are described using the same interface model.

IV. Service Compatibility

This section first defines the notion considered to measure interface compatibility. Then, we present the interface mismatches which can be detected using our measure.

A. Unidirectional Complementarity (UC)

Compatibility checking verifies the successful interaction between services *wrt.* a criterion set on their observable actions. This criterion is referred to as a compatibility notion. We distinguish two classes of notions depending on

[1]Although checking protocol compatibility is undecidable with asynchronous communication [4], Fu *et al.* proved in [11] that a large class of interfaces can be analyzed under an asynchronous communication model using techniques and tools existing for the synchronous communication model.

the direction of the compatibility checking, that are, bidirectional and unidirectional analysis. Here we consider an unidirectional compatibility notion for illustration purposes, namely unidirectional complementarity (*UC* for short). Two services are compatible *wrt.* the *UC* notion if there is one service (*complementer*) which must eventually receive (send, respectively) all messages that its partner (*complemented*) expects to send (receive, respectively) at all global reachable states. In addition, both services must be deadlock-free in all reachable global states. Hence, the *complementer* service may send and receive more messages than the *complemented* service.[2] This asymmetric notion is useful for checking the successful communication in the client/server model where a server can interact with clients having different behaviors. In this setting, each client behavior must be satisfied by the server.

B. Measuring Interface Compatibility

In order to resolve the interface mismatches, we reuse our measure presented in [15] which computes the interface compatibility and detects the mismatches. In what follows, we present the intuition behind our measure, yet more details can be found in [15]. The computation process accepts as input two service protocols $STS_1 = (A_1, S_1, I_1, F_1, T_1)$ and $STS_2 = (A_2, S_2, I_2, F_2, T_2)$ 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 to what degree the two interfaces are far from being compatible *wrt.* this compatibility notion, *e.g.,* the *UC* compatibility considered in this paper.

To measure the compatibility of two service protocols, we compute the protocol compatibility degrees for all possible global states using a set of static compatibility measures. In our work, we use three static compatibility measures, namely state natures, labels, and exchanged parameters. These measures are used next to analyze the behavioral part (ordering of labels) of both protocols. Intuitively, two states are compatible if their backward and forward neighboring states are compatible, where the backward and forward neighbors of state s' in transitions (s, l, s') and (s', l', s'') are respectively the states s and s''. Hence, in order to measure the compatibility degree of two service protocols, we consider an iterative approach which propagates the compatibility degree from one state to all its neighbors. This process is called compatibility flooding.

C_P^k stands for the resulting matrix where each entry $C_P^k[s_i, s_j]$ represents the compatibility measure of a global state $(s_i, s_j) \in S_1 \times S_2$ at the k^{th} iteration, and P corresponds to the compatibility notion used. C_P^0 represents the initial compatibility matrix where all states are supposed to be perfectly compatible, *i.e.* $\forall (s_i, s_j) \in S_1 \times S_2,\ C_P^0[s_i, s_j] = 1$. The compatibility degree of (s_i, s_j) at the k^{th} iteration is computed as the sum of its previous compatibility degree (at the $k-1^{th}$ iteration) together with the current compatibility degree propagated from its neighbors. The propagated compatibility is computed using function $state\text{-}comp^k_{CN,D}((s_1, s_2))$ defined as the weighted average of three measures: the forward and backward propagated compatibilities, and the value returned by the function $nat(s_1, s_2)$ which compares state natures. The forward and backward propagated compatibilities are computed depending on the parameters CN and D. For instance, in the case of *UC* compatibility which is an unidirectional notion, the interfaces are analyzed from the client point of view since the correct interaction is governed by this client requirements.

Finally, the compatibility degree is normalized, *i.e.,* divided by the maximal value that can be achieved. $C_P^k[s_1, s_2]$ is formally defined as follows:

$$C_P^k[s_1, s_2] = \frac{C_P^{k-1}[s_i, s_j] + state\text{-}comp^k_{CN,D}((s_1, s_2))}{2}$$

Our iterative process terminates when the Euclidean difference $|C^k - C^{k-1}|$ of matrices C^k and C^{k-1} converges, *i.e.,* reaches an epsilon ε such that $\varepsilon > 0$. Lastly, the compatibility matrix comes with a mismatch list which enables understanding of the incompatibility issues.

C. Interface Mismatches

There has been a lot of effort devoted to studying the different classes of interface mismatches, see for instance [8], [13]. In our work, we focus on the most common mismatches in the literature, and organize them into two classes, namely, behavioral and static mismatches.

Behavioral Mismatches. Behavioral mismatches can be detected at the protocol level where the execution order of exchanged messages is taken into account. The first kind of mismatch is that a message can be expected to be received in an order which is different to that which is being sent by the partner. Another kind of behavioral mismatch is known as a n-to-m ($n \neq m$) matching issue where (i) a transition, a state, or a message, on one interface corresponds to n (> 1) matches on the other interface, this is called a split/merge matching issue; (ii) some of these interface constituents do not have any match in the partner's interface, this is called a missing/extra matching issue. The resolution of this mismatch ensures 1-to-1 matching. In the remainder of this paper, as far as the *UC* compatibility notion is considered, 1-to-1 matching must be guaranteed from the client point of view meaning that each client state or transition must exactly match one service state or transition, respectively.

Static Mismatches. There are two subtypes of static mismatches: (i) signature, in which sent and received messages have different names and/or incompatible sort lists (parameter types) — *i.e.,* both lists do not share all types in the same order; and (ii) state nature, in which two compared states do not have the same nature meaning that both states are initial, final or neither.

[2]Our definition is different from simulation or preorder relations [7] since we compare protocols with opposite directions.

V. Client Interface Update

In this section, we apply our compatibility measure to resolve the interface mismatches and make both the client (complemented) and the service (complementer) *UC* compatible. To achieve this objective, we propose a process to change the client interface (Figure 1).

In step 1, we compute the compatibility measure which compares both interfaces. Then, step 2 relies on the resulting compatibility matrix to generate an interface mapping tree which describes the best state matching on both interfaces. Based on the analysis of the mapping tree, we change the client interface as follows. In step 3, behavioral mismatches are resolved first because protocol level is the crucial issue in service-based communication. Protocols guide designers to write clients that correctly interact with a given service. Steps 4 and 5 again compute the interface compatibility and the mapping tree in order to take the changes made in the previous step into account. The analysis of the updated mapping tree enables us to resolve the static mismatches in step 6. Lastly, after partial resolution of interface mismatches, step 7 computes the compatibility measure to check if the updated client is compatible with its service interface.[3] The update process terminates if these interfaces are compatible and the user requirements are satisfied. Otherwise, a new iteration should be started from step 2. Note that this process can be parameterized by a set of user requirements to prevent undesirable behaviors that the designer does not want to appear in the new client interface.

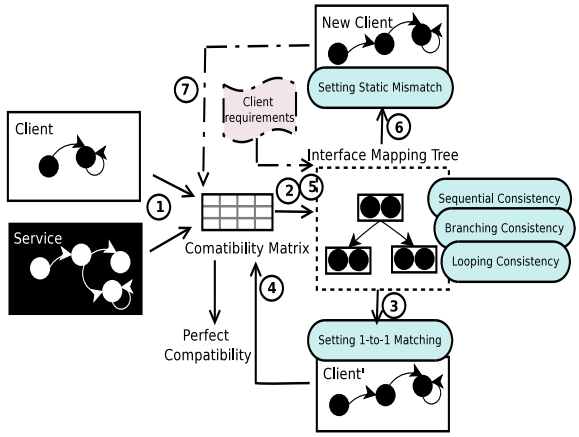

Fig. 1. Overview of our Client Update Process

A. User Requirements

Our update process enables the designer to set some requirements which must be satisfied all along this process. These requirements consist of the set of messages which must not appear in the updated interface. This prevents undesirable behaviors that the designer does not want to appear in the new client interface. The update process may fail and an error message is returned if no compatible client can be generated due to the specified requirements.

B. Interface Mapping Tree

The interface mapping tree describes a set of linked nodes where each node represents the best matching of a client state with one state among those on the service interface. Furthermore, each node is linked to its parent and children nodes.

Definition 3 (Node): Let us consider two STSs, $STS_{i\in\{clt,sce\}} = \{(A_i, S_i, I_i, F_i, T_i)\}$, which describe a client and a service protocol, respectively. A node n is a triple (pr, gs, ch) where: pr is the link to the parent node, gs is the global state $(s_{clt}, s_{sce}) \in S_{clt} \times S_{sce}$ meaning that s_{sce} is the best match for s_{clt}, and ch stands for the set of links to this node's children.

We assume the following functions necessary for analyzing the mapping tree. Given a node $n = (pr, gs, ch)$, $state(n) = gs$, $parent(n) = pr$, and $children(n) = ch$.

Definition 4 (Deadlock Node): Let us consider two STSs, $STS_{i\in\{clt,sce\}} = \{(A_i, S_i, I_i, F_i, T_i)\}$, which describe a client and a service protocol, respectively. A deadlock node is a leaf node n where $children(n) = \emptyset$, and $state(n)$ is not a deadlock state.[4]

Definition 5 (Interface Mapping Tree): Let us consider two STSs, $STS_{i\in\{clt,sce\}} = \{(A_i, S_i, I_i, F_i, T_i)\}$, which describe a client and a service protocol, respectively. An interface mapping tree Υ is defined using its root node (ϵ, I, ch) where I is the initial global state (I_{clt}, I_{sce}), and $children(n) = ch$.

The mapping tree is computed as follows. First, the root node n has no parent — *i.e.*, $parent(n) = \epsilon$ — and stands for the matching of the initial states on Int_{clt} and Int_{sce}. Then, the children nodes are incrementally added using the measures in the compatibility matrix. The evaluation of the compatibility measure returns candidate global states where every global state matches a client state and a service state with which the client has the highest compatibility score. These global states are referred to as best state matchings. A node corresponding to a best state matching can be added if both interfaces are able to evolve into a target global state using messages different than those belonging to user requirements.

This reasoning for generating the mapping tree allows one to add some incorrect nodes into the tree. A node n is considered correct if every child node's state can be reached from $state(n)$ using a correct transition. Thus, in order to avoid incorrect nodes, we rely on the protocol information to check correct transitions relating every node's state with their children nodes' states. In the following, we give the four transition levels we consider to check the nodes' correctness: **Sequential Transition.** This analysis ensures that for every node n all the states of its children nodes must be reached following correct sequential transitions, that is, if $state(n) =$

[3]Our results presented in [15] prove that two interfaces are considered compatible *wrt.* a compatibility notion *CN* if and only if they are compatible at their initial global state — *i.e.*, the compatibility measure is equal to 1.

[4]A global state (s_{clt}, s_{sce}) is considered a deadlock state if $(s_{clt}, s_{sce}) \notin F_{clt} \times F_{sce}$ and no transition goes out from s_{clt} and s_{sce}.

(s_{clt}, s_{sce}) and it exists (s'_{clt}, s'_{sce}) in $children(n)$, then s'_{clt} and s'_{sce} must be successors of s_{clt} and s_{sce}, respectively. Formally, we consider the sequential transition between $state(n)$ and its children nodes correct if the following condition holds: $\forall (pr, gs, ch)$, $ch \subseteq succ(gs)$, where for $i = clt$ and $j = sce$, $gs = (s_i, s_j)$:

$$succ(s_i, s_j) = \begin{cases} \{(s'_i, s'_j)\} \cup succ(s'_i, s'_j) & \text{if } \exists (s_i, l_i, s'_i) \in T_i, \\ & \exists (s_j, l_j, s'_j) \in T_j \\ \{(s'_i, s_j)\} \cup succ(s'_i, s_j) & \text{else if } \exists\ (s_i, \tau, s'_i) \in T_i \\ \{(s_i, s'_j)\} \cup succ(s_i, s'_j) & \text{else if } \exists\ (s_j, \tau, s'_j) \in T_j \\ \emptyset & \text{otherwise} \end{cases}$$

Branching Transition. States holding outgoing choices might lead to mismatched target states but which appear in the mapping tree as best state matching. For instance, one state s_{clt} from the client interface can be matched with two different states s_{sce} and s'_{sce} reached using a choice in the service interface if s_{clt} has the same highest compatibility value with s_{sce} and s'_{sce}. In this case, we keep only one matching from these two solutions in order to ensure 1-to-1 state matching (see Section IV-C). Note that this choice can be enforced by making it explicit in user requirements.

Looping Transition. This analysis compares looping behaviors in the client and the service interfaces, and detects whether there are loops to be added or removed in the client protocol. At the level of mapping tree computation, this consists in adding or removing some nodes. It can also require an update of parent and children links of other existing nodes.

Deadlock-Freedom. A mapping tree can be used to update a client interface if and only if it does not include any deadlock node. If there is a deadlock node, an error message is returned indicating that no compatible client can be generated.

Example: Let us illustrate the computation of an interface mapping tree from a compatibility matrix and some user requirements. We give in Figure 2 a simplified example of a database management system where a user can access an online database to search data or make an update, and waits for its acknowledgement. The database service can first receive a request for an update or a registration to be acknowledged.

First scenario with empty set of user requirements. The resulting mapping tree is obtained using the compatibility matrix computed *wrt. UC* notion. User and Database are the complemented (client) and complementer (service) protocols, respectively. Each node represents a User state with its best state match among those in Database. Our update techniques presented in Section V-D rely on this mapping tree to change the User's interface so that its protocol becomes as User'.

Second scenario with non-empty set of user requirements. Given the requirement set which is equal to {ackR}, this means that the user does not want to receive the registration acknowledgement. The computation of the mapping tree returns a deadlock node (represented by the dashed rectangle in Figure 2). This node does not have any child node because the message ackR! going out from state s3 cannot be matched with any message at state c2 due to the restriction made by the user requirements. Thus, no compatible user interface can be generated using this interface mapping tree.

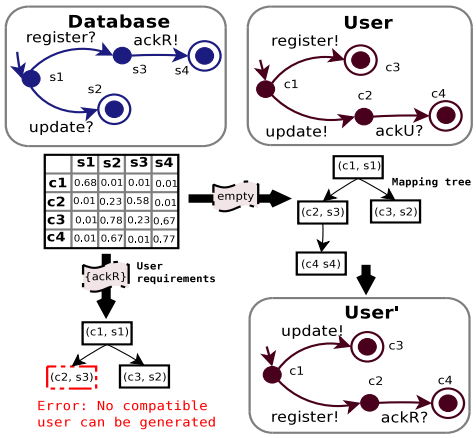

Fig. 2. Database Management System.

C. Resolution of Behavioral Mismatches

In this section, we present our techniques to resolve the behavioral mismatches. These techniques rely on the interface mapping tree, and aim at ensuring the client-based 1-to-1 matching. Thus, the first change pattern to be considered is *add/remove states* defined as follows. Given an interface mapping tree Υ computed for two STSs, $STS_{i \in \{clt, sce\}} = \{(A_i, S_i, I_i, F_i, T_i)\}$:

- Each client's state which does not appear in any Υ node must be removed if this state can be reachable and lead to other states through observable labels (different than τ actions) — *i.e.*, $\forall s_{clt} \in S_{clt}$, if $\forall\ n$, $s_{clt} \notin state(n)$, $S_{clt} = S_{clt} \setminus \{s_{clt}\}$. An example of this change is illustrated in Figure 3, where the client's state c2 needs to be removed.

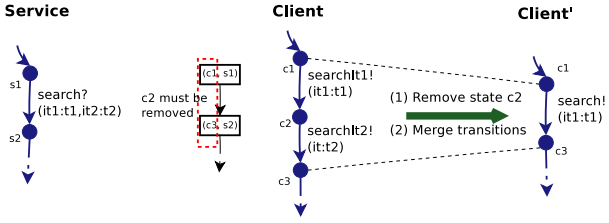

Fig. 3. Remove State and Merge Transitions Patterns.

- A new client's state must be added every time there is a state $s_{sce} \in S_{sce}$ reachable and leading, through observable labels, to states having corresponding matches in Υ, but s_{sce} does not appear in any node. Figure 4 illustrates an example where the service's state s2 does not have any match in Int_{clt}. Since this state forwards state s1 and precedes state s3 which match states c1 and c2, respectively, the client change consists in adding a new state to be matched with s2.

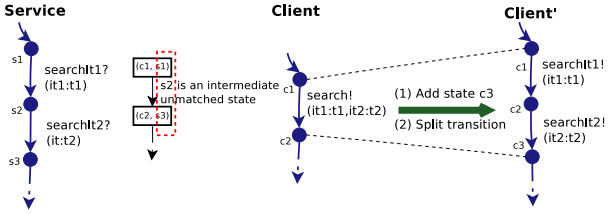

Fig. 4. Add State and Split Transitions Patterns.

The previous pattern may imply another change, namely, *merge/split transitions* where transitions can be removed or added. This change considers the location of handled transitions, but also the links to their predecessor and successor transitions in the interaction protocol. Below, we give the steps to be followed for adding or removing a transition (s, l, s'):

- We compute all predecessors and successors transitions of (s, l, s'). Then, for each of those transitions, we update either its target (for a predecessor transition) or source state (for a successor transition) *wrt.* to s and s'.
- With regards to adding a transition, if there is no operation profile $op(ti_1, \ldots, ti_n, to_1, \ldots, to_m) \in \Sigma_{clt}$ which corresponds to the profile of the action represented with l, this label and its missing profile should be included inside A_{clt} and Σ_{clt}, respectively. However, if l's profile exists in Σ_{clt}, two alternatives are possible: (i) reuse l if $l \in A_{clt}$; (ii) otherwise, $A_{clt} = A_{clt} \cup \{l\}$.
- Transition deletion requires the remove of l and its operation profile — *i.e.*, $A_{clt} = A_{clt} \backslash \{l\}$ and $\Sigma_{clt} = \Sigma \backslash \{m(ti_1, \ldots, ti_n, to_1, \ldots, to_m)\}$ where $l = (m, d, pl)$ and $ti_1, \ldots, ti_n, to_1, \ldots, to_m$ stand for types of parameters in pl.

Checking Internal behaviors. We now focus on the application of both previous patterns when considering internal behaviors. Checking the correct match of states reachable or leading to other states using only internal behaviors can be handled differently as follows:

- τ transitions can be minimized modulo τ-confluence reduction [16] if they appear in sequence. Here, this internal behavior does not have any effect on the external interaction between the client and the service.
- States holding internal choices described with branches of τ transitions are more complicated to handle. Considering the *UC* compatibility notion, two cases must be checked: (i) The existence of such states in Int_{clt} may result in additional transitions with unmatched states *wrt.* Υ. As a consequence, these transitions must be removed from Int_{clt}. An example is illustrated at the bottom of Figure 5. (ii) If the internal choice is detected in Int_{sce}, this could lead to a deadlock due to the consideration of the *UC* notion. An illustration of this issue is given in the example at the top of Figure 5. The state s5 and its successors do not have any match in Int_{clt} although the service can internally reach s5. In this case, adding a transition at state c2 to match cancel! message at s5 will never make both interfaces *UC* compatible at the global states $(c2, s5)$ and $(c2, s3)$ since there will be an unmatched transition in the complemented side, and this does not respect the *UC* requirements. Here, no compatible client can be computed and our approach returns an error message for that issue.

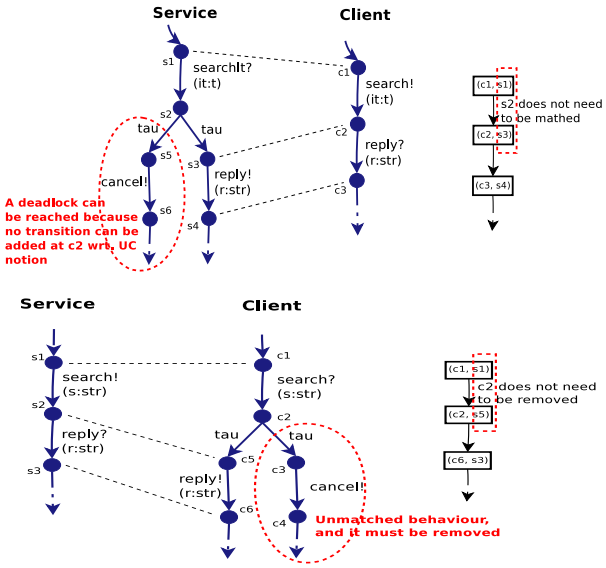

Fig. 5. Branching Internal Behaviors in Service (top) and Client Protocols (bottom).

D. Resolution of Static Mismatches

We have shown in the previous section our method to resolve behavioral mismatches. Before starting the resolution of the static behaviors, we need to compute again the mapping tree and the compatibility measure in order to take the previous updates into account. Then, the resolution of static mismatches can be accomplished considering the exploration of Υ and checking the mismatches detected by our compatibility measure at every node state. Here, there are also different static patterns which can be applied to Int_{clt}. Given an interface mapping tree Υ computed for two STSs, $STS_{i \in \{clt, sce\}} = \{(A_i, S_i, I_i, F_i, T_i)\}$:

Checking State Nature. This step aims at unifying the state nature every time there exists a node n such that $state(n)$ presents a nature mismatch. In this case, the client's state nature must be set similarly to the one of the service's state.

Checking Transition Labels. Here, different alternatives must be studied. First of all, starting from the initial node, we incrementally explore every node n, and systematically apply the following checks. For each n's child node n', if both interfaces can transit from $state(n)$ to $state(n')$ using labels l_{clt} and l_{sce} which present a message name mismatch computed with the compatibility measure, then: (i) If there exists a client's label l'_{clt} which perfectly matches the service's transition label l_{sce}, the update here consists in replacing the client's transition label l_{clt} with l'_{clt}; (ii) Otherwise, a new label and its operation

profile must be added to A_{clt} and Σ_{clt}, respectively, and also the client's transition must be labelled with this new label.

Checking Signatures and Labels. This change consists in adding or removing labels and operation profiles to or from A_{clt} and Σ_{clt}. Another possible change here corresponds to an update of sent and received lists of parameters and their types in the operation profiles.

VI. Case Study: Medical Management System

In this section we illustrate the application of our techniques to resolve the service evolution issues on a case study. We will use an online medical management system which handles patient appointments. This system is inspired from an example originally presented in [6]. As can be seen in Figure 6, we reuse a medical server MedServer and an example of a client interface User. The User can first log on to a server by sending his/her user name and password, and receive an acknowledgement. Then, he/she asks for an appointment with a specialist doctor, and then receives the appointment-related information. Service MedServer first receives a user name followed by its password, and acknowledges the connection request. Then, this service can receive and reply to a request for either an appointment with a general practitioner or a specialist doctor. In the latter case, the service can reply differently depending on the availability of the specialist. In this example, we assume that both interfaces were initially *UC* compatible, but the service interface has been changed to become as shown in Figure 6. As a consequence, the direct reuse of the service interface is not possible due to several mismatches presented between both interfaces — *e.g.*, messages user? and pwd? in the MedServer interface do not match the message login! in the User interface.

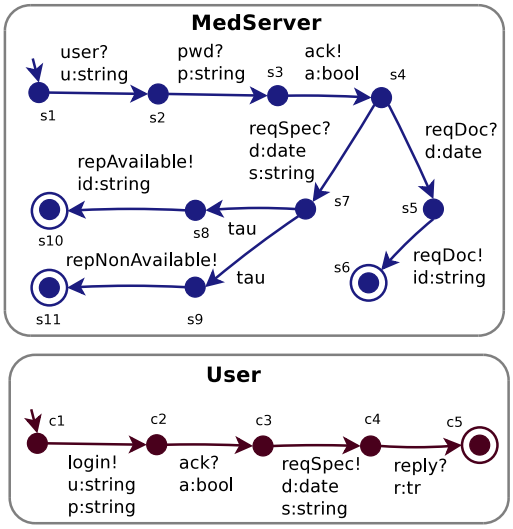

Fig. 6. The STSs of the MedServer Service (top) and its User (bottom)

Let us show how the client update process can be applied in order to resolve these mismatches. We initially assume an empty set of user requirements.

	s1	s2	s3	s4	s5	s6	s7	s8	s9	s10	s11
c1	**0.74**	0.11	0.01	0.04	0.01	0.01	0.01	0.01	0.01	0.01	0.01
c2	0.01	0.45	**0.6**	0.34	0.36	0.01	0.22	0.27	0.26	0.01	0.01
c3	0.04	0.38	0.34	**0.63**	0.34	0.06	0.21	0.26	0.26	0.05	0.01
c4	0.01	0.35	0.39	0.34	**0.48**	0.01	0.4	**0.5**	0.43	0.01	0.01
c5	0.01	0.01	0.01	0.06	0.01	**0.73**	0.01	0.01	0.01	**0.73**	0.67

TABLE I
THE FIRST COMPATIBILITY MATRIX USED TO COMPUTE User'.

	s1	s2	s3	s4	s5	s6	s7	s8	s9	s10	s11
c1	**0.75**	0.06	0.01	0.03	0.01	0.01	0.01	0.01	0.01	0.01	0.01
c2	0.15	**0.56**	0.35	0.37	0.34	0.01	0.21	0.26	0.26	0.01	0.01
nc0	0.01	0.4	**0.65**	0.34	0.41	0.01	0.23	0.28	0.28	0.01	0.01
c3	0.04	0.37	0.34	**0.63**	0.34	0.05	0.21	0.26	0.26	0.04	0.01
c4	0.01	0.34	0.38	0.34	**0.48**	0.01	0.38	**0.48**	0.42	0.01	0.01
c5	0.01	0.01	0.01	0.05	0.01	**0.73**	0.01	0.01	0.01	**0.73**	0.67

TABLE II
THE SECOND COMPATIBILITY MATRIX USED TO COMPUTE User'.

Compatibility Measuring. The very first step as highlighted in Section V is measuring the interface compatibility. We show in Table I the resulting matrix which will be used as input to the client update process (step 1 in Figure 1). Each line in the matrix represents the compatibility measure of a client state with all those on the service interface.

Interface Mapping Tree. The first mapping tree (Mapping Tree1) given in Figure 7 (step 2 in Figure 1) is computed using the matrix in Table I. Every node in this tree describes the best matching of a client state with one state among all those on the service interface, and is computed following the techniques sketched in Section V-B.

Resolution of Behavioral Mismatches. Behavioral mismatches (step 3 in Figure 1) in our example are resolved as follows. The check of the initial node n where $pr(n) = \epsilon$, $state(n) = (\mathsf{c1}, \mathsf{s1})$, and $ch(n) = \{(\mathsf{c2}, \mathsf{s3})\}$, requires adding a new state to be matched with s2, and also splitting the first transition into two transitions in order to resolve the mismatch existing between messages user? and pwd? on MedServer interface and message login! on User interface.

Iterative Process. Once the behavioral mismatches are resolved, we need to again compute the compatibility matrix as well as the mapping tree (steps 4 and 5 in Figure 1) in order to consider the new changes. Regarding our example, the compatibility matrix is given in Table II and the interface mapping tree computed using this matrix is represented with Mapping Tree2 in Figure 7. As we can see, this tree includes new nodes where their states are represented with the dashed rectangles, and which consider the matching of the added state nc0.

Resolution of Static Behaviors. The previous step ensures 1-to-1 state and transition matching, but we still need to apply the static check (step 6 in Figure 1). In our example, checking the nodes in Mapping Tree2 implies the following changes. First, transitions going out from states c4 and c5 must be labelled with messages compatible with the ones going out from states s5 and s6. This change is followed with an update of both A_{User} and Σ_{User} in order to add the labels standing for both messages and their profiles. Lastly, the old labels and profiles which are not used are removed.

	s1	s2	s3	s4	s5	s6	s7	s8	s9	s10	s11
c1	**1.0**	0.06	0.01	0.03	0.01	0.01	0.01	0.01	0.01	0.01	0.01
c2	0.06	**1.0**	0.35	0.37	0.35	0.01	0.21	0.26	0.26	0.01	0.01
nc0	0.01	0.35	**1.0**	0.34	0.41	0.01	0.23	0.28	0.28	0.01	0.01
c3	0.06	0.41	0.34	**1.0**	0.34	0.05	0.21	0.26	0.26	0.04	0.01
c4	0.01	0.35	0.41	0.34	**1.0**	0.01	0.33	0.42	0.32	0.01	0.01
c5	0.01	0.01	0.01	0.05	0.01	**1.0**	0.01	0.01	0.01	0.74	0.67

TABLE III
THE MATRIX OF COMPATIBLE User' AND MedServer.

Updated User Interface. The User update process results in its new interface User' presented in Figure 7. The compatibility matrix computed for User' and MedServer is given in Table III (step 7 in Figure 1), and shows that those interfaces are *UC* compatible — *i.e.*, both interfaces can correctly communicate. This is detected because these interfaces are compatible, that is, the compatibility measure for their initial states $(\mathsf{c1}, \mathsf{s1})$ is equal to 1.

As a very last step of the update process, the end-user can validate the behavior described by the new client interface. For instance, in our example, we assume that the end-user would not accept a consultation with a general practitioner — *i.e.*, the user requirement set is now updated with the message name reqDoc. In this case, our approach will reset the user interface to its initial version and restart the update process taking the requirement set into account. Unfortunately, the process cannot return a compatible user interface because the satisfaction of his/her requirements leads to a state with internal choices which may cause system deadlock (as shown in Section V-C), and an error message is returned to inform the end-user of this issue.

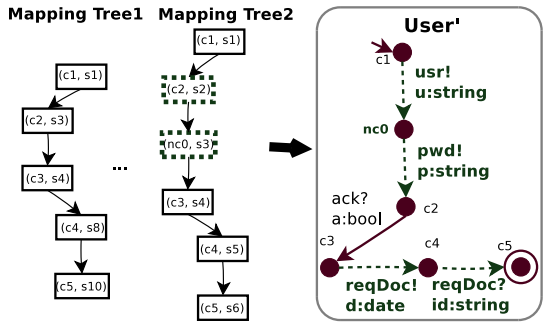

Fig. 7. Mapping Trees and Updated User.

VII. CONCLUSION AND PERSPECTIVES

In this paper, we have introduced a framework to resolve the compatibility issues related to the evolution of black-box services. We proposed a systematic method to update the client and ensure system compatibility. The use of our compatibility measure has enabled us to compare the interfaces and also detect the mismatches to be worked out. The update process can be parameterized with some user requirements to prevent the behavior that a designer does not want to appear in the client interface. Our solution is automated by a prototype tool, called Updator, we have implemented and experimented with many examples.

In the future, we first plan to generate the client implementation corresponding to the updated interface. We will also extend our work in order to consider other compatibility notions existing in the literature. So far, user requirements are described using messages that must not appear in the updated client interface. We would like to consider temporal logic properties to enforce an order in which some messages must appear. Finally, we will study service evolution in dynamic systems.

Acknowledgements. This work has been partially supported by the project TIN2008-05932 funded by the Spanish Ministry of Innovation and Science and FEDER, and by the project P07-TIC03131, funded by the Andalusian government.

REFERENCES

[1] V. Andrikopoulos, S. Benbernou, and M. P. Papazoglou. Evolving Services from a Contractual Perspective. In *Proc. of CAiSE'09*, volume 5565 of *LNCS*, pages 290–304. Springer, 2009.
[2] A. Azough, E. Coquery, and M. S. Hacid. Supporting Web Service Protocol Changes by Propagation. In *Proc. of WI'09*, pages 438–441. IEEE, 2009.
[3] S. Becker, A. Brogi, I. Gorton, S. Overhage, A. Romanovsky, and M. Tivoli. Towards an Engineering Approach to Component Adaptation. In *Architecting Systems with Trustworthy Components*, volume 3938 of *LNCS*, pages 193–215. Springer, 2006.
[4] D. Brand and P. Zafiropulo. On Communicating Finite-State Machines. *J. ACM*, 30(2):323–342, 1983.
[5] J. Cámara, J. Antonio Martín, G. Salaün, J. Cubo, M. Ouederni, C. Canal, and E. Pimentel. ITACA: An Integrated Toolbox for the Automatic Composition and Adaptation of Web Services. In *Proc. of ICSE'09*, pages 627–630. IEEE, 2009.
[6] J. Cámara, G. Salaün, C. Canal, and M. Ouederni. Interactive Specification and Verification of Behavioural Adaptation Contracts. In *Proc. of QSIC'09*, pages 65–75. IEEE Computer Society, 2009.
[7] R. Cleaveland and O. Sokolsky. Equivalence and Preorder Checking for Finite-State Systems. *Handbook of Process Algebra*, pages 391–424, 2001.
[8] M. Dumas, M. Spork, and K. Wang. Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation. In *Proc. of BPM'06*, volume 4102 of *LNCS*, pages 65–80. Springer, 2006.
[9] F. Durán, M. Ouederni, and G. Salaün. Checking Protocol Compatibility using Maude. In *Proc. of FOCLASA'09*, volume 255, pages 65–81. ENTCS, 2009.
[10] X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *Proc. of WWW'04*, pages 621–630. ACM Press, 2004.
[11] X. Fu, T. Bultan, and J. Su. Synchronizability of conversations among web services. *IEEE Trans. Software Eng.*, 31(12):1042–1055, 2005.
[12] M. Hennessy and H. Lin. Symbolic Bisimulations. *TCS*, 138(2):353–389, 1995.
[13] W. Kongdenfha, R. Saint-Paul, B. Benatallah, and F. Casati. An Aspect-Oriented Framework for Service Adaptation. In *Proc. of ICSOC'06*, volume 4294 of *LNCS*, pages 15–26. Springer, 2006.
[14] R. Mateescu, P. Poizat, and G. Salaün. Adaptation of Service Protocols Using Process Algebra and On-the-Fly Reduction Techniques. In *Proc. of ICSOC'08*, volume 5364 of *LNCS*, pages 84–99. Springer, 2008.
[15] M. Ouederni, G. Salaün, and E. Pimentel. Measuring the Compatibility of Service Interaction Protocols. In *Proc. of SAC'11*, volume 2, pages 1560–1567. ACM, 2011.
[16] G. J. Pace, F. Lang, and R. Mateescu. Calculating-Confluence Compositionally. In *Proc. of CAV'03*, volume 2725 of *LNCS*, pages 446–459. Springer, 2003.
[17] M. P. Papazoglou. The Challenges of Service Evolution. In *Proc. of CAiSE'08*, volume 5074 of *LNCS*, pages 1–15. Springer, 2008.
[18] S. H. Ryu, F. Casati, H. Skogsrud, B. Benatallah, and R. Saint-Paul. Supporting the Dynamic Evolution of Web Service Protocols in Service-Oriented Architectures. *TWEB*, 2(2):1–46, 2008.
[19] G. Salaün, L. Bordeaux, and M. Schaerf. Describing and Reasoning on Web Services using Process Algebra. *IJBPIM*, 1(2):116–128, 2006.