Diagnosing and measuring incompatibilities between pairs of services

Ali Ait-Bachir, Marie-Christine Fauvet

Diagnosing and measuring incompatibilities between pairs of services*

Ali Aït-Bachir[1], Marie-Christine Fauvet[1]

LIG, University of Grenoble, France
{Ali.Ait-Bachir, Marie-Christine.Fauvet}@imag.fr

Abstract. This text presents a tool, from its design to its implementation, which detects all behavioural incompatibilities between two service interfaces. Unlike prior work, the proposed solution does not simply check whether two services are incompatible or not, it rather provides detailed diagnosis, including the incompatibilities and for each one the location in the service interfaces where these incompatibilities occur. A measure of similarity between interfaces which considers outputs from the detection algorithm is proposed too. A visual report of the comparison analysis is also provided which pinpoints a set of incompatibilities that cause a behavioural interface not to simulate another one.

1 Introduction

A service *interface* is defined as the set of messages the service can receive and send, and the inter-dependencies between these messages. Service interfaces can be seen from at least three perspectives: structural, behavioural and non-functional. The structural interface of a service describes the types of messages that the service produces or consumes and the operations underpinning these message exchanges. In the case of web services, the structural interface of a service can be described for example in WSDL [20]. The behavioural interface refers to the order in which the service produces or consumes messages. This can be described for example using BPEL ([20]) business protocols, or more simply using state machines as discussed in this paper. Finally, the non-functional interface refers to reliability, security and other aspects that are not considered to be part of the functional requirements of a service. The work presented here focuses on behavioural interfaces and is complementary to other work which has studied the problem of structural interface incompatibility [17].

The study described in this text aims at providing a tool which is capable of reporting incompatibilities between two service interfaces. Its main contributions are:

– An algorithm which detects *all* differences that cause two service interfaces not to be compatible from a behavioural viewpoint.

* This work is partially funded by the Web Intelligence Project, Rhône-Alpes French Region

- A measure of similarity between behavioural interfaces of services which is based on the outputs of the detection algorithm. This measure evaluates the degree of similarity between two interfaces.
- A tool which implements the algorithm and the similarity measure and provides business process designers a visual diagnosis, resulting from the incompatibility detection process applied on two interfaces.

The paper is structured as follows. Section 2 frames the problem addressed and introduces a motivating example. In Section 3 we show how we model service interfaces according to their behavioural dimension. Section 4 presents the principle of the proposed approach while Section 5 details the detection algorithm and discusses implementation details and experiments. Section 6 compares the proposal with related ones, and Section 7 concludes and sketches further work.

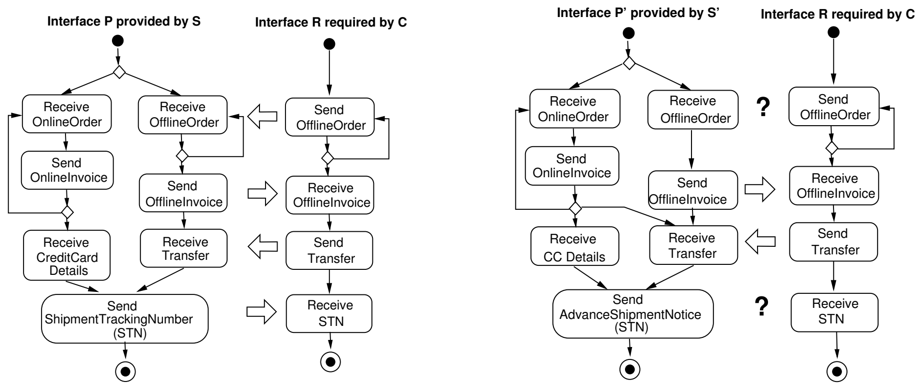
2 Motivation

As a motivating example, we consider services that handle purchase orders processed either online or offline. In Figure 1 the behavioural interfaces are described using UML activity diagram notation that captures control-flow dependencies between message exchanges (i.e. activities for sending or receiving messages). The figure distinguishes between the *provided* interface that a service exposes, and its *required* interface as it is expected by its clients or peers. Specifically, Figure 1-a shows the provided interface P of a service S. S interacts with a client application C that requires an interface R. We consider the scenario where C wishes to interact with another service S' whose interface is P' while meeting the same needs then S (see Figure 1-b).

In this setting, and considering client applications or peers of the service S, the questions that we address are: (i) do the differences between P and P' cause incompatibilities between S' and client(s) of S? and if so, (ii) which differences lead to these incompatibilities? Specifically, we consider three situations: (1) an operation[1] is defined in P while it is not in P', (2) conversely, an operation is defined in P' while it is not in P, (3) an operation is defined in P and changed with another one in P'. We argue that other changes can be described in terms of these ones.

In Figure 1, we observe that the flow which loops from *Receive OfflineOrder* back to itself in P does not appear in P'. In other words, customers of S' are not allowed to alter offline orders. This is a source of incompatibility since clients that rely on interface P may attempt to send messages to alter their offline order while the service S' does not expect a new order after the first one. On the other hand, message *ShipmentTrackingNumber* (STN in short) has been replaced in P' by message *AdvanceShipmentNotice* (ASN in short). This difference will certainly cause an incompatibility *vis-a-vis* of S's clients and peers. Another difference is that paying by bank transfer is offered in service S' while it is

[1] We use the terms *operation* and *message* interchangeably, while noting that strictly speaking, messages are events that initiate or result from operations.



(a) P compatible with R (b) P' not compatible with R

Fig. 1. Differences between two service interfaces.

not in service S. However, this difference does not lead to any incompatibilities since S's clients have not been designed to use this option. In technical terms, a difference between P' and P only leads to an incompatibility if it causes P' not to simulate P.

3 Modelling behavioural dimension of service interfaces

In our approach, the detection of incompatibilities relies on an abstract representation of service interfaces with an emphasis on behavioural aspects. Thus, we consider order dependencies between messages but we do not look into the schema of these messages. Accordingly, we model the behaviour of a web service interface using *Finite State Machines (FSM [5,16])*. Our choice of FSMs is motivated by the following reasons:

– It is arguably the simplest and most widely understood model of system behaviour and it has been used in several previous work in the area of behavioural service interface analysis [6,4,15].
– It is sufficiently powerful to capture most forms of behaviour encountered in service interfaces, including race conditions and interleaved parallelism.
– There exist transformations from other notations for service behaviour modelling to FSMs. In particular several transformations from BPEL to FSMs are implemented in existing tools such as WS-Engineer [9].

Following [5,14], we adopt a simple yet effective approach to model service interface behaviour using *Finite State Machines* (FSMs). In the FSMs we consider, transitions are labelled with messages (to be sent or received). When a message is sent or received, the corresponding transition is fired. Figure 2 depicts FSMs of provided interfaces P and P' of the running example presented in Section 2. The message m has prefix $>$ (respectively $<$) when it is sent (respectively received). Each conversation initiated by a client starts an execution of the corresponding FSM. The figure shows also all differences between P and P'. This latter will be discussed in the next section.

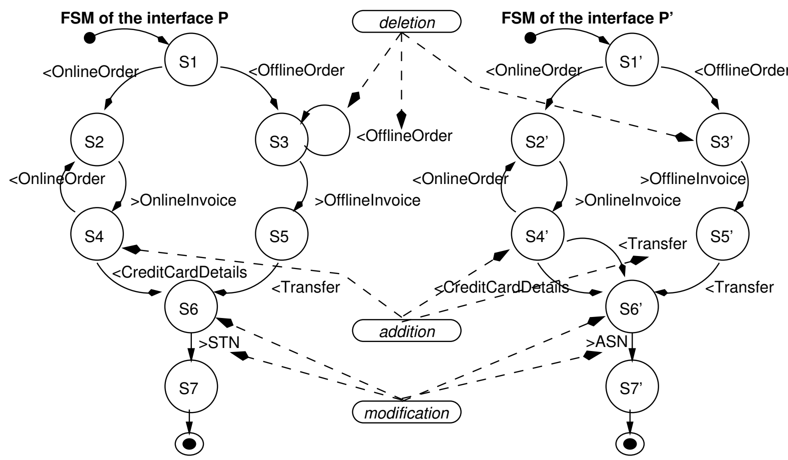


Fig. 2. FSMs modelling P and P'.

Definitions and notations :

An FSM is a tuple (S, L, T, s_0, F) where: S is a finite set of states, L a set of events (actions), T the transition function ($T : S \times L \longrightarrow S$). s_0 is the initial state such as $s_0 \in S$, and F the set of final states such as $F \subset S$. The transition T associates a source state $s_1 \in S$ and an event $l_1 \in L$ to a target state $s_2 \in S$.

To check whether or not differences between an interface P (of service S, seen as a reference) and another one P' (of service S') lead to incompatibilities, it is necessary to identify situations when P' does not simulate P. Actually, if P' simulates P then each interface R required by the clients of S, which are compatible with P remain compatible with P' (see [2] for a proof).

Assumptions :

(1) Even thought web service communication is not always synchronous, we assume synchronous communication as it provides, to a certain extent, a suitable basis for analysing service behaviour. First of all, synchronous communication is more restrictive than asynchronous communication. Therefore, incompatibilities that arise within the asynchronous case arise in the synchronous case as well. Second, for a relatively large class of interfaces, it has been shown that adopting the synchronous communication model leads to the same analysis results than adopting the asynchronous model [10].

(2) We focus on interfaces that expose only externally visible behaviour. In particular, internal actions or timeouts do not appear in the service interface unless they are externalised as messages.

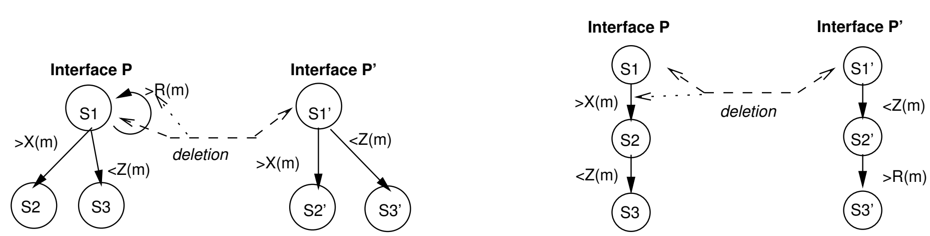
(3) We assume messages with the same structure to be equivalent.

4 Detection of differences

To detect differences between P and P', their respective FSMs are traversed synchronously starting from their respective initial states s_0 and s'_0. The traversal seeks for two states s and s' (belonging respectively to P and P') which are such as the sub-automaton starting from s in P and the one starting from s' in P' are *incompatible* (details are given in Section 5.1). We first discuss and illustrate the conditions that need to be evaluated when P has an operation which does not exist in P' (for the sake of simplicity we call this situation, a deletion, see Section 4.1) and when an operation in P is replaced with another one in P' (this is called a modification, see Section 4.2). We do not detail here the situation when P' has an operation which does not exist in P as it is transposed from the addition mentioned above.

4.1 Deletion of an operation

Figure 3 depicts two situations where an operation appears in P and not in P'. First in Figure 3-a, we observe that all operations enabled in state $S1'$ are also enabled in state $S1$. Moreover, there is an operation (namely >R(m)) enabled in state S that has no match in state $S1'$. Hence we conclude that, considering the pair of states $S1$ and $S1'$, >R(m) is missing in P'. Once this difference has been detected, the pairs of states to be examined next in the process of comparing P and P' are $\langle S2, S2' \rangle$ and $\langle S3, S3' \rangle$: $S2$ in P and $S2'$ in P' are targets of transitions both labelled by the same operation: >X(m). The same remark applies to $S3$ and $S3'$ with the operation <Z(m).



(a) >R(m) deleted in P' (b) >X(m) deleted in P'

Fig. 3. Diagnosis of deletions

In Figure 3-b we note that first, the operation <Z(m) is enabled in $S1'$ and not in $S1$, and second the operation >X(m) is enabled in $S1$ but not in $S1'$. There are two reasons for this mismatch: either operation >X(m) has been modified and has become <Z(m), or >X(m) has been deleted. In this example, we can discard the former possibility because <Z(m) appears downstream in the FSM of P' (it labels an outgoing transition of state $S2$). Hence, <Z(m) can not be considered as a replacement for >X(m). Thus, we conclude that >X(m) has been deleted in P'. Once this difference has been detected, the pair of states to be examined next in the process of comparing P and P' is $\wr angle S2, S1'\rangle$.

Formally, when comparing two interface FSMs P and P', the fact an operation is defined in P and missing in P' is diagnosed in a pair of states $\langle s, s'\rangle$ (respectively belonging to P and P) if the following condition holds (each part of this condition is explained further down).

$$\|Label(s\bullet) - Label(s'\bullet)\| \geqslant 1 \wedge \|Label(s'\bullet) - Label(s\bullet)\| = 0 \quad (1)$$

$$\vee\ \exists t \in s\bullet, \exists t' \in s'\bullet : Label(t) \notin Label(s'\bullet) \wedge ExtIn(t', (t\circ)\bullet) \quad (2)$$

In the previous equations, the notations given below apply (examples refer to Figure 3):

– $s\bullet$ is the set of outgoing transitions of s (e.g. $\mathsf{S1}\bullet = \{\langle\mathsf{S1},\ \mathsf{>X(m)},\ \mathsf{S1}\rangle, \langle\mathsf{S1},\ \mathsf{<Z(m)},\ \mathsf{S3}\rangle, \langle\mathsf{S1},\ \mathsf{>R(m)},\ \mathsf{S2}\rangle\}$
– $t\circ$ is the target state of the transition t. (e.g. $\langle\mathsf{S1},\ \mathsf{<Z(m)},\ \mathsf{S2}\rangle\circ = \mathsf{S2}$).
– Label(t) is the label of t. (e.g. Label($\langle$ S1, <Z(m), S2$\rangle$) = <Z(m))
– $\parallel X \parallel$: cardinality of X.
– The $\circ$ operator (respectively $\bullet$) is generalised to a set of transitions (respectively states). For example, if $T = \bigcup_{i=1}^{n}\{t_i\}$ then $T\circ = \bigcup_{i=1}^{n}\{t_i\circ\}$; where $n = \parallel T \parallel$. Similarly, operator *Label* is generalised to a set of transitions.

A deletion is detected in state pair (s, s') in two cases. The first one (line 1) is when every outgoing transition of s' can be matched to an outgoing transition of s, but on the other hand, there is an outgoing transition of s that can not be matched to a transition of s'. A second case is when there exists a pair of outgoing transitions t and t' (of states s and s' respectively) such that: (i) transition t can not be matched to any outgoing transition of s'; and (ii) the label of t' occurs somewhere in the FSM rooted at the target state of t (line 2).[2] This second condition is tested in order to determine whether the non-occurrence of t's label among the outgoing transitions of s' should indeed be interpreted as a deletion, as opposed to a modification or an addition. To check if a transition label occurs somewhere in the FSM rooted at the target of a given transition, we use the following recursive Boolean function: $ExtIn(t, T) \equiv T \neq \emptyset \wedge (Label(t) \in Label(T) \vee \bigcup_{i=1}^{\|T\|} ExtIn(t, (T_i\circ)\bullet))$. In other words, $ExtIn(t, T)$ (where t is a transition and T is a set of transitions) evaluates to true if either transition t's label appears among the labels of transitions in T ($Label(t) \in Label(T)$) or, there exists a transition taken in T which has a target state whose set of outgoing transitions (namely $T1$) is such that $ExtIn(t, T1)$ evaluates to true. The way it is defined, this recursive function does not converge if the FSM has cycles, but it can be trivially extended to converge by adding an input parameter to store the set of visited states and to ensure that each state is only visited once.

[2] By *FSM P rooted at s* we mean FSM P in which the initial state is set to be s. This means that we ignore any state or transition that is not reachable from s.

4.2 Modification of an operation

Figure 4 shows a situation where we can diagnose that operation $>\mathsf{X(m)}$ has been replaced by operation $>\mathsf{Y(m)}$ (i.e. a modification). The reason is that the operation $>\mathsf{X(m)}$ is enabled in $S1$ but not in $S1'$, and conversely $>\mathsf{Y(m)}$ is enabled in $S1'$ but not in $S1$. Moreover, the transition labelled $>\mathsf{X(m)}$ does not match to any transitions t' in state $S1'$ such that operation $>\mathsf{X(m)}$ occurs downstream along the branch starting with t', and symmetrically, $>\mathsf{Y(m)}$ does not match any transitions t of state $S1$ such that $>\mathsf{Y(m)}$ occurs downstream along the branch starting with t. Thus we can not diagnose that $>\mathsf{X(m)}$ has been deleted, nor can we diagnose that $>\mathsf{Y(m)}$ has been added.

In this case, the pairing of transition $>\mathsf{X(m)}$ with transition $>\mathsf{Y(m)}$ is arbitrary. If state $S1'$ had a second outgoing transition labelled $>\mathsf{Z(m)}$, we would just as well diagnose that $>\mathsf{X(m)}$ has been replaced by $>\mathsf{Z(m)}$. Thus, when we diagnose that $>\mathsf{X(m)}$ has been replaced by $>\mathsf{Y(m)}$, all we capture is that $>\mathsf{X(m)}$ has been replaced by another operation, possibly $>\mathsf{Y(m)}$. The output produced by the proposed technique should be interpreted in light of this.

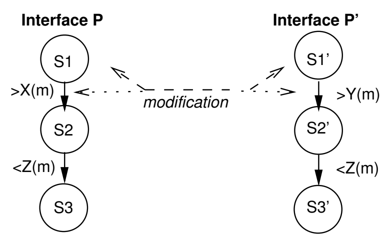


Fig. 4. Diagnosis of a modification/replacement

The state pair to be visited next in the synchronous traversal of P and P' is such that both transitions involved in the modification are traversed simultaneously. In this example, $\langle S2, S2' \rangle$ should be visited next.

Formally, a modification is diagnosed in state pair (s,s') if the following condition holds:

$$\exists t1 \in s\bullet, \exists t1' \in s'\bullet : \mathit{Label}(t1) \notin \mathit{Label}(s'\bullet) \wedge \mathit{Label}(t1') \notin \mathit{Label}(s\bullet)$$
$$\wedge \neg\exists t2 \in s\bullet : \mathit{ExtIn}(t1', (t2\circ)\bullet)) \wedge \neg\exists t2' \in s'\bullet : \mathit{ExtIn}(t1, (t2'\circ)\bullet))$$

5 Implementation details and experiments

The detection algorithm presented below (see Section 5.1) is implemented in a tool whose main feature is to detect differences between two behavioural interfaces that cause that the second interface does not simulate the behaviour of the first one[3] [1].

[3] See `http://mrim.imag.fr/ali.ait-bachir/webServices/webServices.html`

5.1 Detection algorithm

The algorithm implementing the detection illustrated in the previous section is detailed in Figure 5. Given two interface FSMs P and P', the algorithm traverses P and P' synchronously starting from their respective initial states s_0 and s'_0. At each step, the algorithm visits a state pair consisting of one state from each of the two FSMs. Given a state pair, the algorithm determines if an incompatibility exists and if so, it classifies it as an addition, deletion or modification. If an *addition* is detected (e.g. an operation is enabled from s'_0 in P' and not from s_0 in P), the algorithm progresses along the transition of the operation in the interface it has been added. Conversely, if the change is a *deletion* (e.g. an operation is enabled from s_0 in P and not from s'_0 in P'), the algorithm progresses along the transition of the deleted operation in. However, if a *modification* is detected, the algorithm progresses along both FSMs simultaneously. While traversing the two input FSMs, the algorithm accumulates a set of differences represented as tuples of the type Difference defined as below:

type Difference: $<$ State, Transition, State, Transition $>$

{ *Let* $\langle s, t, s', t' \rangle$ *be of type Difference:* s *and* s' *are states respectively belonging to FSMs* P *and* P' *to be compared.* $t = null \iff t' \neq null \wedge t'$ *is enabled in* P' *while it is not in* P *(*t' *added in* P'*),* $t' = null \iff t \neq null \wedge t$ *is enabled in* P *while it is not in* P' *(*t *is deleted),* $t \neq null \wedge t' \neq null \iff t$ *in* P *is modified by* t' *in* P'. }

For instance, the detection algorithm applied on the motivating example (see Figure 2) returns the set of tuples {⟨S2, <OfflineOrder, S2', null⟩, ⟨S4, null, S4', <Transfer⟩ ⟨S6, >STN, S6', >ASN⟩} which summarises the differences found when comparing P' to P. It is worth noting that comparing P to P' returns {⟨S2',<null, S2, OfflineOrder⟩, ⟨S4',<Transfer, S4, null⟩ ⟨S6',>ASN, S6, >STN⟩}.

The algorithm proceeds as a depth-first algorithm over state pairs of the compared FSMs. Two stacks are maintained: one with the visited state pairs and another with state pairs to be visited (see Figure 5, line 5). These state pairs are such that the first state belongs to the FSM of Pi while the second state belongs to the FSM of Pj. The first state pair to be visited is the one containing the initial states of Pi and Pj (line 6). Once a pair of states is visited it will not be visited again. To ensure this, the algorithm uses the variable visited to memorise the already visited pairs of states (line 10).

Labels in common among those of outgoing transitions of si and labels of outgoing transitions of sj are considered as unchanged (no change to detect). Thus, a set of state pairs is built where states are target states of common labels (line 11). Also, the algorithm reports all differences between the outgoing transitions of si and the outgoing transitions of sj (line 12). The two set differences of transitions are put in two variables difPiPj (transitions whose labels belongs to $Label(si\bullet)$ but do not belongs to $Label(sj\bullet)$) an difPjPi (transitions whose labels belong to $Label(sj\bullet)$ but do not belong to $Label(si\bullet)$). Line 13 calculates all combinations of transitions whose labels are not in common among $Label(si\bullet)$ and $Label(sj\bullet)$.

```
Detection (Pi: FSM, Pj: FSM): {Difference}
2  { Detection (Pi,Pj) is the set of differences between Pi and Pj. }
3  setRes: { Difference } { the result }
4  si, sj: State { auxiliary variables }
5  visited, toBeVisited: Stack of type <State, State>
       { pairs of states that have been visited / must be visited }
7  toBeVisited.push(< initState(Pi), initState(Pj) >)
8  while notEmpty(toBeVisited)
9      < si, sj > ← toBeVisited.pop()
10     visited.push( < si, sj > ) { < si, sj > is now considered as visited }
11     combEqual ← {(ti, tj) ∈ si• × sj• | Label(ti) = Label(tj)}
       { pairs of matching transitions }
12     difPiPj ← {ti ∈ si• | Label(ti) ∉ Label(sj•)}
       difPjPi ← {tj ∈ sj• | Label(tj) ∉ Label(si•)}
13     combPiPj ← difPiPj × difPjPi
       { all pairs of si and sj uncorresponding outgoing transitions. }
14     If ‖difPiPj‖ ⩾ 1 and ‖difPjPi‖ = 0 then                          { deletion }
15         For each t in difPiPj do setRes.add(< si, t, sj, null>)
16             If((t∘, sj) ∉ visited) then toBeVisited.push((t∘, sj))
17     If ‖difPjPi‖ ⩾ 1 and ‖difPiPj‖ = 0 then                          { addition }
18         For each t in difPjPi do
19             If (polarity(t) = 'send') then setRes.add(< si, null, sj, t>)
               { otherwise this addition does not lead to incompatibility }
20             If ((si, t∘) ∉ visited) then toBeVisited.push((si, t∘))
21     For each (ti, tj) in combPiPj do
22         If ExtIn(ti, (tj∘)•) then                                     { addition }
23             setRes.add(< si, null, sj, tj>)
24             If ((si, tj∘) ∉ visited) then toBeVisited.push((si, tj∘))
25         If ExtIn(tj, (ti∘)• ) then                                    { deletion }
26             setRes.add(< si, ti, sj, null,'deletion'>)
27             If ((ti∘, sj) ∉ visited) then toBeVisited.push((ti∘, sj))
28         If ( (¬∃tj′ ∈ sj• : ExtIn(ti, (tj′∘)•))
               ∧(¬∃ti′ ∈ si• : ExtIn(tj, (ti′∘)•))) then                  { modif. }
29             setRes.add(< si, ti, sj, tj>)
30             if((ti∘, tj∘) ∉ visited) then toBeVisited.push((ti∘, tj∘))
31     For each (ti, tj) in combEqual do
           If ((ti∘, tj∘) ∉ visited) then toBeVisited.push((ti∘, tj∘))
32 Return setRes
```

Fig. 5. Detection algorithm

Lines 14 to 16 are dedicated to detect a deletion when an outgoing transition of si does not match any transition in $sj\bullet$. The result is returned as set of tuples $< si, t, sj, null >$ where t is one of the outgoing transitions of si whose label does not appear in any of sj's outgoing transitions. As mentioned in Section4.1, when an operation is deleted in Pj FSM the algorithm progresses in Pi FSM, along the branch of the transition which does not exist in Pj, but remains in the same state in Pj FSM.

The detection of an addition is quite similar to the detection of a deletion (lines 17 to 20).

The variable combPiPj contains transition pairs such that the label of the first transition ti belongs to $si\bullet$ but does not belong to $Label(sj\bullet)$ while the label of the second transition tj belongs to $sj\bullet$ but not to $Label(si\bullet)$. For each transition pair satisfying this condition, the algorithm checks the conditions for diagnosing an *addition* (lines 22 to 24), a *deletion* (lines 25 to 27) or a *modification* (lines 28 to 30).

Finally, the algorithm also progresses along pairs of matching transitions, i.e. pairs of transitions with identical labels (line 31). In fact, if no incompatibilities are detected in the current state pair, the algorithm will only progress along pairs of transitions that match one another.

5.2 Complexity of the detection algorithm

Let P and P' be two interface FSMs given as input to the detection algorithm, P (respectively P') has n (resp. n') states and m (resp. m') transitions. Also, let w and w' be the number of distinct transition labels appearing in P and P' respectively. We observe that the algorithm performs a depth-first search over the space of state pairs $\langle s, s' \rangle$ such that s is a state of P and s' is a state of P'. The algorithm visits each state pair at most once, therefore one component of the complexity is $O(n * n')$. We then observe that for each visited state pair, the algorithm examines transitions pairs $\langle t, t' \rangle$ such that t is an outgoing transition of s and t' is an outgoing transition of s'. Also, when a transition t in one FSM can not be matched to a transition in the other FSM, we examine t individually. Overall each transition pair $\langle t, t' \rangle$ such that t is a transition of P and t' is a transition of P' is examined at most once. Additionally, each transition t in P and t' in P' is examined at most once individually. Thus another component of the complexity is $O(m * m' + m + m')$. Since the first term dominates the other two, this can be written as $O(m * m')$. Thus, the complexity of the traversal is $O(n * n' + m * m')$.

For each visited pair $\langle t, t' \rangle$ of transitions a condition is evaluated. This condition is based on the transition labels and, in some cases, it also involves a "look-ahead" operation. The purpose of this look-ahead is to find, for a given label, whether or not this label appears in the FSM rooted at either the target of t or the target of t'. This look-ahead can be avoided as follows. In a pre-processing stage, we traverse each of the two FSMs individually using a breadth-first search algorithm. During this traversal, we construct a look-up table that maps each state s to a list of pairs $\langle l, b \rangle$ where l is a transition label and b is a Boolean value indicating whether or not l is the label of a transition reachable from s. For each state s, we calculate the value of b for each label, based on the corresponding values of b for each direct successor of s. This step is linear on the number of labels appearing in the FSM. Thus, the complexity of this pre-processing is $O((n + m) * w)$ for P and $O((n' + m') * w')$ for P' . Since the number of distinct labels in an FSM is bounded by the number of transitions, the complexity of the pre-processing stage is bounded by $O(n * m + (m)^2 + n' * m' + (m')^2)$.

Adding up the complexity of the pre-processing and the detection algorithm, the overall complexity is $O(n*m+(m)^2+n'*m'+(m')^2+n*n'+m*m')$. Assuming the number of transitions in an FSM is greater than the number of states (which, modulo one transition, holds because the FSMs are connected graphs), the complexity is bounded by $O((m+m')^2)$. Thus the worst-case complexity is quadratic on the total number of transitions in both FSMs.

5.3 Measure of similarity

This section presents a measure meant to give a quantitative evaluation of *how much* an interface is different from another one. This measure relies on a function $QS : VStates \rightarrow [0..1]$ where $VStates$ is the set of state pairs visited by the detection algorithm ($VStates \subseteq S \times S'$, S being the set of states in P and S' the set of those in P'). Given a pair of states $\langle s, s' \rangle \in VState$, $QS(\langle s, s' \rangle)$ measures incompatibilities detected at $\langle s, s' \rangle$ relatively to the number of transitions in common between s and s'. The formulæ is (see explanations below):

$$QS(\langle s, s' \rangle) = \begin{cases} 1 & if\, s\bullet = \emptyset \\ \dfrac{\parallel LC \parallel + \sum_{d \in Diff(\langle s,s' \rangle)} Weight(d)}{\parallel LC \parallel + \parallel Diff(\langle s, s' \rangle) \parallel} & otherwise \end{cases}$$

$LC = Label(s\bullet) \cap Label(s'\bullet)$ is the set of labels in common in transitions whose sources are s and s'. $Diff(\langle s, s' \rangle)$ is the set of differences pinpointed from the state pair $\langle s, s' \rangle$. The function $Weight : \mathsf{Difference} \rightarrow [0..1[$ is such as $Weight(d)$ is the penalty associated with d. Penalties are arbitrary chosen and depend on whether the difference is an addition, a deletion or a modification.

When s does not have any outgoing transitions, $QS(\langle s, s' \rangle) = 1$. Otherwise, QS tends toward zero as the weight of incompatibilities, evaluated relatively to the global number of transitions in common, rooted at s and s'. For a fixed number of these transitions, more differences are found at $\langle s, s' \rangle$ higher is the dividend and closer to 0 is $QS(\langle s, s' \rangle)$. The divisor, which is meant to keep QS in $[0, 1]$, is never equal to 0: either s has no outgoing transition ($QS(\langle s, s' \rangle) = 1$), or s has at least one outgoing transition and it corresponds to a difference ($\parallel Diff(\langle s, s' \rangle) \parallel > 0$) or not ($\parallel LC \parallel \geq 1$).

For example, in Figure 2, assuming the penalty for the deletion is set to 0.5, thus: $QS(\langle \mathsf{S3}, \mathsf{S3'} \rangle) = (1+0.5)/(1+1)=0.75$ while $QS(\langle \mathsf{S1}, \mathsf{S1'} \rangle) = (1+0)/(1+0)=1$

Eventually, to quantitatively compare P and P', we propose to calculate the mean of values returned when applying QS on each pair of states visited by the algorithm. This is done by the function MQS. $MQS(P, P') = 1$ means that P' simulates P.

$$MQS(P, P') = \sum_{p \in VStates} QS(p) / \parallel VStates \parallel$$

In the running example, if the penalty values are set to 0.5 then the mean quantitative simulation is: $MQS(P, P') = 0.875$.

5.4 Experimental results

For validation purposes, we built a test collection of 15 behavioural interfaces derived from the textual description of choreographies expressed in the standard xCBL[4]. The experiment consisted in comparing interfaces to each other.

Table 1 gives a fragment of the results obtained when comparing service interfaces. Each line reports the comparison between the interface seen as a reference and a particular interface given by its id number (see column *Interface*). In the column *MQS* is displayed the value returned when applying the function *MQS* (see above) to the list of differences built by the detection algorithm. The number of items in this list is given in column *Nb diff* while the column *States* (resp. *Transitions*) shows how many states (resp. transitions) where found in the interface to be compared. Each interface has between 3 and 16 transitions. The interface given as a reference has 11 states and 13 transitions.

Interface	*MQS*	*States*	*Transitions*	*Nb diff*
♯12	1	11	13	0
♯14	0.977	11	13	1
♯13	0.875	10	13	3
♯1	0.43	4	3	11
♯3	0.37	6	6	16
♯5	0.30	8	11	21
♯11	0.233	10	14	19

Table 1. Fragment of experimental results

The interface whose id is ♯11 has 10 states and 14 transitions. It has 19 differences with the interface given as the reference. The value returned by MQS is 0.233 which is lower then the one returned when comparing the interface whose id is ♯5. The interface ♯5 has a better score (0.30) then the one which id is ♯11, even thought ♯5 has less differences then ♯11. The interface ♯12 scores 1 and has no difference with the reference, thus it simulates the reference interface.

6 Related work

The issues tackled in this paper have been partially addressed before, with various points of view. Web service interactions may fail because of interface incompatibilities according to their structural dimension. In this context, reconciling incompatible interactions leads towards transforming message types (using for instance Xpath, XQuery, XSLT). Issues that arise in this context are similar to those widely studied in the data integration area. A mediation-based approach is proposed in [3]. While this approach relies on a mediator (called *virtual supplier*)

[4] XML Common Business Library (`http://www.xcbl.org/`).

it focuses on structural dimension of interfaces only. Detecting incompatibilities is proceeded manually.

In [14], authors introduced a technique to diagnosis message structure mismatches between service interfaces and to fix them with adaptors. An extension of this technique is applied to resolve mismatches between service protocols. The proposed iterative algorithm builds a mismatch tree to help developers to choose the suitable ad apter each time and incompatibility is detected. However, this technique can only be applied to protocols which describe a sequence of operations. More complex flow controls such as iterative or conditional compositions are not taken into consideration. The solution proposed in this text does not have this limitation. Another drawback of this approach is that adaptors have no control logic and can not resolve complicated protocol mismatches, such as extra condition, missing condition, or iteration structure, etc.

Compatibility test of interfaces has been widely studied in the context of Web service composition. Most of approaches which focus on the behavioural dimension of interfaces rely on equivalence and similarity calculus to check, *at design time*, whether or not interfaces described for instance by automata are compatible (see for example [6,11]). The behavioural interface describes the structured activities of a business process. Checking interface compatibility is thus based on bi-similarity algorithms [13]. These approaches do not deal with pinpointing exact locations of incompatibilities as our proposition does.

Recent research has addressed interface similarity measure issues. In [18], authors present a similarity measure for labelled directed graphs inspired by the simulation and bi-simulation relations on labelled transition systems. The presented algorithm returns a value of a simulation measure but does not give the location of the incompatibilities which have been detected. Its complexity is exponential or factorial to the number of states of the graphs to be compared. According to this theoretical result, our algorithm is more efficient. A similar algorithm with the same limitations and complexity has been used in service discovery as introduced in [8]. More specifically, some algorithms for detecting incompatibilities have been proposed, but they focus only on structural aspect of interfaces and do not address their behavioural dimension [7].

Change patterns have been introduced in [19] which characterise different types of business process evolution. Each pattern models a set of rules which are used by the designer to decide whether or not to propagate changes on executing instances of the modified process or to abort them. As web services are used as black boxes, this approach does not apply to web services.

Recent research has addressed interface similarity measures issues. In [12], the author presents a similarity measure for labelled directed graphs inspired by the simulation and bi-simulation relations on labelled transition systems. The author applies this technique to detect and correct deadlocks. Other algorithms based on graph-edit distances have been applied to service discovery in [8], but do not pinpoint behavioural differences between services.

In [15], authors propose an operator *match* which is a similarity function comparing two interfaces for finding correspondences between models. This function is the same as the one introduced in [18] which consider the behavioural semantics. The similarity measure is a heuristic which returns a value which calculated according to changes involved by the addition and by the deletion of an operation. However, the result do not pinpoints the exact location of these changes.

In [21], the authors propose an approach to business process matchmaking based on automata extended with logical expressions associated to states. Their algorithm determines if the languages of two automata (which model two business processes) have a non-empty intersection. This technique for detecting process differences returns a Boolean output. It does not provide detailed diagnosis.

7 Conclusion and further study

In this text we have presented both design and implementation of a tool intended to detect differences (*addition*, *deletion* or *modification* of an operation) that give rise to behavioural incompatibilities between two service interfaces. The main originality of the proposed solution is that the detection algorithm does not stop at the first incompatibility encountered but keeps searching further to identify all incompatibilities leading up to the final state of one of the interfaces to be compared. We have introduced a measure of similarity between interfaces. This measure is meant to be used to select, among a set of services, which one has the closest interface to a given service interface.

Ongoing work aims at extending the proposed solution toward two directions: (i) detecting complex types of incompatibilities (e.g. the order of two operations is swapped or an entire branch is deleted); and (ii) assisting business process designers in determining how to address an incompatibility. Also, communications are currently assumed to be synchronous. Future work will aim at extending the technique to address the asynchronous case. This extension can be achieved by maintaining a buffer of unconsumed messages during the traversal, as it is proposed in [14].

References

1. A. Ait-Bachir, M. Dumas, and M.-C. Fauvet. BESERIAL: Behavioural service analyser. In *Proc. of BPM Int. Conf., demo session*, number LNCS 5240, Milano, Italy, 2008.
2. A. Ait-Bachir, M. Dumas, and M.-C. Fauvet. Detection behavioural incompatibilities between pairs of services. In *Proc. of the 4th WESOA08 in conj. with the 6th ICSOC*, Sydney, Australia, 2008.
3. M. Altenhofen, E. Boerger, and J. Lemcke. An execution semantics for mediation patterns. In *Proc. of the BPM'2005 Workshops: Workshop on Choreography and Orchestration for Business Process Managament*, France, 2005.
4. B. Benatallah, F. Casati, D. Grigori, H. Motahari-Nezhad, and F. Toumani. Developing adapters for web services integration. In *Proc. of the 17th Int. Conf. on Advanced Information System Engineering, CAiSE*. Springer Verlag, 2005.

5. D. Beyer, A. Chakrabarti, and T. A. Henzinger. Web service interfaces. In *Proc. of the 14th WWW int. conf.*, Japan, 2005. ACM, New York, USA.
6. L. Bordeaux, G. Salan, D. Berardi, and M. Mecella. When are two web services compatible? In *Proc. 5th Int. Conf. on Technologies for E-Services (LNCS)*, Canada, 2004. Springer Verlag.
7. P.-A. Champin and C. Solnon. Measuring the similarity of labeled graphs. In *Proc. of the 5th Int. Conf. On Case-Based Reasoning*, volume LNCS 2689. Springer Verlag, 2003.
8. J. C. Corrales, D. Grigori, and M. Bouzeghoub. BPEL processes matchmaking for service discovery. In *OTM Conferences*, LNCS 4275. Springer Verlag, 2006.
9. H. Foster, S. Uchitel, J. Magee, and J. Kramer. WS-Engineer: A tool for model-based verification of web service compositions and choreography. In *Proc. of the IEEE Int. Conf. on Software Engineering*, China, 2006.
10. X. Fu, T. Bultan, and J. Su. Synchronizability of conversations among web services. *IEEE Transactions on Software Engineering*, 31(12), 2005.
11. S. Haddad, T. Melliti, P. Moreaux, and S. Rampacek. Modelling web services interoperability. In *Proc. of the 6th Int. Conf. on Enterprise Information Systems*, volume 4, Portugal, 2004. ICEIS Press.
12. N. Lohmann. Correcting deadlocking service choreographies using a simulation-based graph edit distance. In *Proc. of BPM Int. Conf.*, number LNCS 5240, Milano, Italy, 2008. Springer Verlag.
13. A. Martens, S. Moser, A. Gerhardt, and K. Funk. Analyzing compatibility of bpel processes. In *Proc. of the Advanced Int. Conf. on Telecommunications and International Conference on Internet and Web Applications and Services*, French Caribbean, 2006. IEEE.
14. H.-R. Motahari-Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-automated adaptation of service interactions. In *Proc. of the 16th Int. Conf. on World Wide Web*, Canada, 2007. ACM.
15. S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. Matching and merging of statecharts specifications. In *Proc. of the 29th Int Conf on Software Engineering*, USA, 2007. IEEE Computer Society.
16. J. Pathak, S. Basu, and V. Honavar. Modeling web service composition using symbolic transition systems. In *Proc. of the 21st Conf. on Artificial Intelligence. Workshop on AI-driven Technologies for Service-Oriented Computing*, USA, 2006.
17. S. R. Ponnekanti and A. Fox. Interoperability among independently evolving web services. In *Proc. of 5th the Int. Middleware Conf. on Middleware*, LNCS 3231, Canada, 2004. Springer Verlag.
18. O. Sokolsky, S. Kannan, and I. Lee. Simulation-based graph similarity. In *Proc. of 12th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems.*, volume 3920, Austria, March 2006. Springer Verlag.
19. B. Weber, S. Rinderle, and M. Reichert. Change patterns and change support features in process-aware information systems. In *Pro. of the 19th Int. Conf. on Advanced Information Systems Engineering*, LNCS 4495, Norway, 2007.
20. S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. Ferguson. *Web Services Platform Architecture*. Prentice Hall, 2005.
21. A. Wombacher, P. Fankhauser, B. Mahleko, and E. Neuhold. Matchmaking for business processes based on choreographies. In *Proc. of the IEEE International Conference on Multimedia and Expo, ICME 2004*, pages 359–368, Taipei, Taiwan, March 2004. IEEE Computer Society Press.