Detecting behavioural incompatibilities between pairs of services*

Ali Aït-Bachir[1]**, Marlon Dumas[2]***, Marie-Christine Fauvet[1]

[1] LIG, University of Grenoble, France
{Ali.Ait-Bachir, Marie-Christine.Fauvet}@imag.fr
[2] University of Tartu, Estonia
Marlon.Dumas@ut.ee

Abstract. We present a technique to analyse successive versions of a service interface in order to detect changes that cause clients using an earlier version not to interact properly with a later version. We focus on behavioural incompatibilities and adopt the notion of simulation as a basis for determining if a new version of a service is behaviourally compatible with a previous one. Unlike prior work, our technique does not simply check if the new version of the service simulates the previous one. Instead, in the case of incompatible versions, the technique provides detailed diagnostics, including a list of incompatibilities and specific states in which these incompatibilities occur. The technique has been implemented in a tool that visually pinpoints a set of changes that cause one behavioural interface not to simulate another one.

1 Introduction

Throughout its lifecycle, the interface of a software service is likely to undergo changes. Some of these changes do not cause existing clients or peers to stop interacting properly with the service. Other changes give rise to incompatibilities. This paper is concerned with the identification of these latter changes.

Service interfaces can be seen from at least three perspectives: structural, behavioural and non-functional. The structural interface of a service describes the schemas of the 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 [14]. The behavioural interface describes the order in which the service produces or consumes messages. This can be described for example using BPEL [14] business protocols, or more simply using state machines as discussed in this paper. The work presented here focuses on behavioural interfaces and is complementary to other work dealing with structural interface incompatibility [12].

In this paper, we present a technique for comparing two service interfaces in order to detect a series of changes that cause them not to be compatible from

* Work partly funded by the Web Intelligence Project, Rhône-Alpes French Region
** The author was supported by a visiting PhD scholarship at University of Tartu.
*** The author is also affiliated with Queensland University of Technology, Australia

a behavioural viewpoint. We consider three types of differences between two services S1 and S2: an operation that produces a message is enabled in a state of service S2 but not in the equivalent state in S1; an operation that consumes a message is enabled in a state in S1 but not in the equivalent state in S2; and and operation enabled in a state of S1 is replaced by another operation in the equivalent state in S2. It may be that S2 allows operation Op to be invoked, but at a different point in time than S1 does. So the diagnosis our technique provides goes beyond what can be provided based purely on structural interfaces.

The paper is structured as follows. Section 2 frames the problem and introduces an example. Section 3 defines a notion of behavioural interface while Section 4 presents the incompatibility detection algorithm. Section 5 compares the proposal with related ones and Section 6 concludes and sketches future work.

2 Motivation

As a running example, we consider a service that handles purchase orders processed either online or offline. Figure 1 depicts three behavioural interfaces related to this example. These behavioural interfaces are described using UML activity diagrams notations that capture 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, the figure shows the provided interface P of an existing service (see left-hand side of the figure). This service interacts with a client application that requires an interface R (shown in the centre of the figure). We consider the scenario where another service which satisfies similar needs, but whose interface is P' (shown in the right-side of the figure). In this setting, the questions that we address are: (i) does the differences between P and P' cause incompatibilities between P' and P's existing client(s); and (ii) if so, which specific changes lead to these incompatibilities. As mentioned above, we consider three types of changes: addition and deletion of an operation enabled in a given state of P, and replacement of an operation enabled in a state of P with a different operation enabled in a corresponding state in P'.[3]

In Figure 1, we observe that the flow that loops from *Receive OfflineOrder* back to itself in P does not appear in P'. In other words, according to P''s interface, customers 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 but the service (with interface P') does not expect a new order after the first order. On the other hand, message *Shipment Tracking Number* (STN) has been replaced in P' by message *Advance Shipment Notice* (ASN). This difference will certainly cause an incompatibility vis-a-vis of existing client applications and peer services. Note also that the possibility of paying by bank transfer has been added to the branch of the behavioural interface that deals with online orders. However, this addition does not lead

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

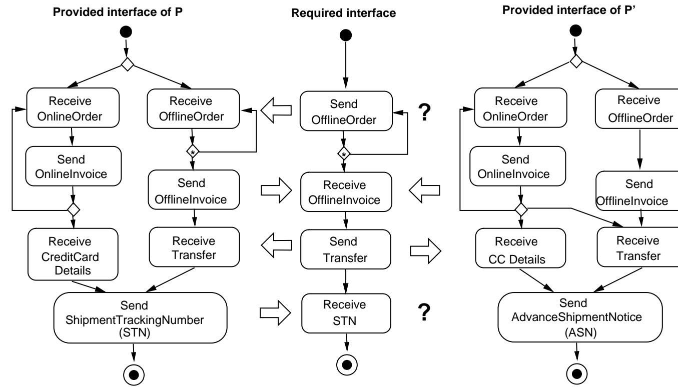

Fig. 1. P and P' provided interfaces.

to an incompatibility since existing client applications or peer services are not designed to use this option. This later case shows that an incompatibility only arises when P' offers less options than P. In technical terms, a change between P' and P only leads to an incompatibility if it causes P' not to simulate P.

3 Modeling service behaviour

The proposed technique for detecting incompatibilities is based on the comparison of behavioural interfaces capturing order dependencies between messages sent and received by a service. We only consider message names, without inspecting the schema of these messages.

Following [3,10], we adopt a simple yet effective approach to model service behaviour using *Finite State Machines* (FSMs). Techniques exist to transform behavioural service interfaces defined in other languages (e.g. BPEL) into FSMs (see for example the WS-Engineer tool [7]), and therefore the choice of FSM should not be seen as a restriction. What we can note is that during the transformation from behaviour description languages that support parallelism (e.g. BPEL) into FSMs, parallelism is encoded in the form of interleaving of actions. For example, the parallel execution of activities a and b is encoded as a choice between 'a followed by b' and 'b followed by a'. In the FSMs we consider, transitions are labelled with message exchanges. 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 is denoted by $>m$ (resp. $<m$) when it is sent (resp. received). Each conversation initiated by a client starts an execution of the corresponding FSM.

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 that $s_0 \in S$, and F the set of final states such that $F \subset S$. The transition function T associates a source state

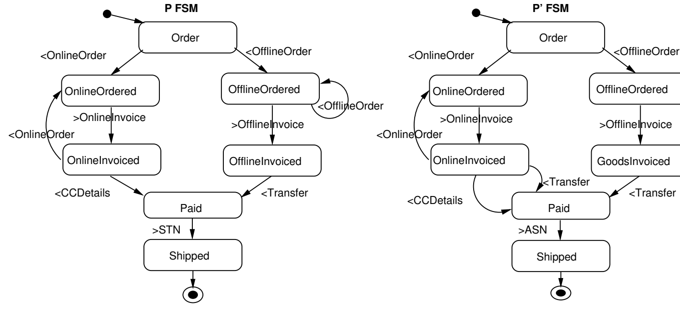

Fig. 2. FSMs of two provided interfaces.

$s_1 \in S$ and an event $l_1 \in L$ to a target state $s_2 \in S$. In this model, a transition is defined as a tuple containing a source state, a label and a target state.

We assume synchronous communication. While in reality Web service communication is not always synchronous, synchronous communication 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 [6].

Another assumption is that 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 externalized as messages.

Below, we use the following notations (examples refer to the FSM P depicted in the left side in Figure 2):

- $s\bullet$ is the set of outgoing transitions from s.
  (e.g. OnlineInvoiced$\bullet$ = {(OnlineInvoiced,<CCDetails,Paid),(OnlineInvoiced, <OnlineOrder,OnlineOrdered) }).
- $t\circ$ is the target state of the transition t.
  (e.g. (OnlineInvoiced,<CCDetails,Paid)$\circ$ =Paid).
- Label(t) is the label of the transition t.
  (e.g. Label((OnlineInvoiced,<CCDetails,Paid))=<CCDetails)
- $\parallel X \parallel$: set cardinality of a set 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 generalized to a set of transitions.

4 Detection of changes

To detect changes, P and P' 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') such that the sub-automaton starting from

s in P and the one starting from s' in P' are *incompatible*. The traversal algorithm is described in section 4.4. But first, we discuss the conditions that need to be evaluated to diagnose each type of change: deletion (see Section 4.1), addition (see Section 4.2) and modification of an operation (see Section 4.3).

4.1 Deletion of an operation

Figure 3 illustrates a situation where a deletion can be diagnosed. It shows two FSMs: one corresponding to a service (P) and the other to another service P'. We observe that all operations enabled in state $S1'$ are also enabled in state S. On the other hand, there is an operation (namely $>R(m)$) enabled in state S that has no match in state $S1'$. So we can conclude that operation $>R(m)$ has been deleted from this particular state.

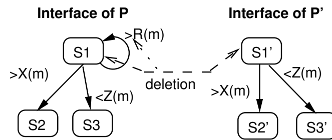

Fig. 3. First case where a deletion is diagnosed

Figure 4 depicts a second scenario where a deletion can be diagnosed. We first note that the above condition does not hold: not all operations enabled in $S1'$ are enabled in $S1$. Indeed, operation *<Z(m)* is enabled in $S1'$ but not in $S1$. At the same time, operation $>X(m)$ is enabled in $S1$ but it is not enabled in $S1'$. There are two possibilities for this mismatch: either operation $>X(m)$ has been modified and has become *<Z(m)*, or operation $>X(m)$ has been deleted altogether. In this example, we can discard the former possibility because *<Z(m)* appears downstream in the interface FSM of P (it is the label of the outgoing transition of state $S2$). Thus, *<Z(m)* can not be considered to be a replacement for $>X(m)$. So we conclude that $>X(m)$ has been deleted.

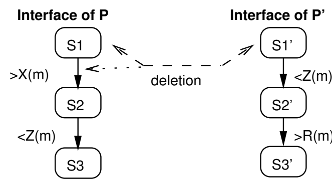

Fig. 4. Second case where a deletion is diagnosed

Once this deletion is detected, the state pair to be examined next in the comparison of P and P' is $(S2, S1')$. In other words, when deleting a transition, we jump to its target state and continue looking for other changes that may be

sources of incompatibilities. For reporting purposes, the deletion is denoted by a tuple $(S1, >X(m), S1', null)$, meaning that operation $>X(m)$ enabled in $S1$ is replaced by the 'null' value in $S1'$. Formally, when comparing two interface FSMs P and P', a deletion is diagnosed in a pair of states s and s' (respectively belonging to P and P') if the following condition holds (each part of this condition is explained below):

$$\|Label(s\bullet) - Label(s'\bullet)\| \geqslant 1 \wedge \|Label(s'\bullet) - Label(s\bullet)\| = 0 \tag{1}$$

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

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).[4] 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.

4.2 Addition of an operation

We now consider the diagnosis of an incompatibility resulting from the addition of an operation in state pair $(S1, S1')$. The simplest case is when all the operations enabled in state $S1$ are also enabled in $S1'$ but not the opposite. This is the case for example of *<Z(m)* in Figure 5. This particular addition however does not lead to an incompatibility because what it does is that it allows a service implementing P' to accept an additional message that a service implementing P would not accept. An existing client of P would simply not send this message. Thus, existing clients of P can interact with P', even though such clients would never use the branch starting with transition labelled $<Z(m)$. On the

[4] 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.

other hand, if we replaced $<Z(m)$ with $>Z(m)$, the addition would give rise to an incompatibility, because the service implementing P' may try to produce a message *Z(m)* that a client of P would not accept.

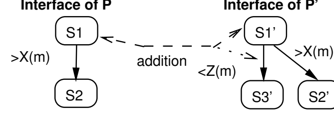

Fig. 5. First case where an addition is diagnosed

Figure 6 illustrates another case where an addition can be diagnosed. Operation *>X(m)* is enabled in state $S1$ and is not enabled in $S1'$. On the other hand, operation *<Y(m)* is enabled in state $S1'$ but not in $S1$ and operation *>X(m)* is enabled in a state downstream along the transition labelled *<Y(m)*. Thus we can conclude that operation *<Y(m)* has been added. This addition constitutes an incompatibility regardless of whether *Y(m)* is sent or received, because the non-occurrence of *Y(m)* would prevent the execution of a service implementing P' to progress along the branch leading to the state where *>X(m)* can occur.

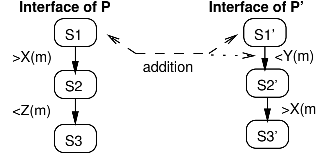

Fig. 6. Second case where an addition is diagnosed

When an addition is detected in a state pair (S, S'), the synchronous traversal of the two FSMs advances along the added transition. In the case of Figure 5 this means that $(S1, S2')$ should be visited next, while in the case of Figure 6, state pair $(S1, S3')$ should be visited next – in addition to $(S2, S2')$ since this latter state pair can be reached by taking transitions $>X(m)$ synchronously. For reporting purposes, the addition of an operation $<Y(m)$ is denoted by a tuple $(S1, null, S1', <Y(m))$. Formally, an addition of an operation is diagnosed in state pair (s, s') if the following condition holds:

$$(\|Label(s\bullet) - Label(s'\bullet)\| = 0 \wedge \|Label(s'\bullet) - Label(s\bullet)\| \geqslant 1) \tag{3}$$

$$\vee\ \exists t \in s\bullet, \exists t' \in s'\bullet : Label(t') \notin Label(s\bullet) \wedge ExtIn(t, (t'\circ)\bullet) \tag{4}$$

An addition is detected in two cases. The first case (line 3) is when there exists an outgoing transition of s' whose label does not match any of the labels of the outgoing transitions of s, while at the same time, every outgoing transition of s can be matched to an outgoing transition of s'. In this case, we need to

additionally check whether the added operation corresponds to a “send” or a “receive”, since an added “receive” does not constitute an incompatibility in this case. The second case (line 4) is when there exists a pair of outgoing transitions t and t' (of states s and s' respectively) with different labels and such that the label of t appears in the FSM rooted at the target of transition t'.

4.3 Modification of an operation

Figure 7 shows a situation where we can diagnose that operation *>X(m)* has been replaced by operation *>Y(m)* (i.e. a modification). We can make this diagnosis because operation *>X(m)* is enabled in $S1$ but not in $S1'$, and conversely *>Y(m)* is enabled in $S1'$ but not in $S1$. Moreover, the transition labelled *>X(m)* can not be matched to a transition t' in state $S1'$ such that operation *>X(m)* occurs downstream along the branch starting with t', and symmetrically, *>Y(m)* can not be matched with a transition t of state $S1$ such that *>Y(m)* occurs downstream along the branch starting with t. Thus we can not diagnose that *>X(m)* has been deleted, nor can we diagnose that *>Y(m)* has been added.

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

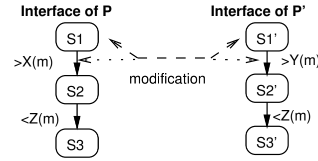

Fig. 7. Diagnosis of a modification/replacement

For reporting purposes, a modification (replacement) of $>X(m)$ into $>Y(m))$ is denoted by tuple $(S1, >X(m), S1', >Y(m))$. 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, $(S2, S2')$ should be visited next. Formally, a modification is diagnosed in state pair (s,s') if the following condition holds:

$$\exists t1 \in s\bullet, \exists t1' \in s'\bullet : \mathit{Label}(t1) \notin \mathit{Label}(s'\bullet) \wedge \mathit{Label}(t1') \notin \mathit{Label}(s\bullet) \quad (5)$$

$$\wedge\neg\exists t2 \in s\bullet : \mathit{ExtIn}(t1', (t2\circ)\bullet)) \wedge \neg\exists t2' \in s'\bullet : \mathit{ExtIn}(t1, (t2'\circ)\bullet)) \quad (6)$$

4.4 Detection algorithm

The algorithm implementing the detection of changes is detailed in Figure 8. Given two interface FSMs P and P', the algorithm traverses P and P' syn-

chronously 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 the algorithm progresses along the transition of the added operation in P' only. Conversely, if the change is a *deletion*, the algorithm progresses along the transition of the deleted operation in P only. 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 changes represented as tuples of the form (s, t, s', t'), as explained previously.

```
1  Detection (Pi: FSM; Pj: FSM ): {Change}
2  { Detection(Pi,Pj) returns a set of tuples of changes represented as tuple of the form < si, ti, sj, tj >
   where si and sj are states of Pi and Pj respectively, while ti and tj are either null values or outgoing
   transitions of si and sj respectively }
3  setRes: {Change}; { result variable }
4  si,sj: State ; { auxiliary variables }
5  visited, toBeVisited : Stack of statePair; { pairs of states that have been visited / must be visited }
6  si ← initState(Pi) ; sj ← initState(Pj)
7  toBeVisited.push((si,sj))
8  while notEmpty(toBeVisited)
9     (si, sj) ← toBeVisited.pop();
10    visited.push( (si, sj) ) { add the current state pair to the visited stack }
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 outgoing transitions of si and sj that do not have a
                                      match }
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. 8. Detection algorithm

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 (line 5). These state pairs are such that the first state belongs to the FSM of Pi while the second state belongs to the one 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 memorize the already visited state pairs (line 10).

Labels that appear both in the outgoing transitions of si and in the outgoing transitions of sj are considered as unchanged. 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 belong to $Label(si\bullet)$ but do not belong 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 $Label(sj\bullet)$.

Lines 14 to 16 detect a deletion when an outgoing transition of si does not match any transition in $sj\bullet$. The result is a 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. The detection of an addition is quite similar to the detection of a deletion (lines 17 to 20).

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 such transition pair, 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 progresses along pairs of matching transitions, i.e. pairs of transitions with identical labels (line 31). The algorithm has a worst-case complexity quadratic on the total number of transitions in both FSMs.

The detection algorithm is implemented in a tool called *BESERIAL* [1] available at `http://www-clips.imag.fr/mrim/User/ali.ait-bachir/webServices/webServices.html`. Figure 9 shows the output of the compatibility analysis performed by BESERIAL on the example introduced in Section 2. Here, *Process2* is the more recent version of the interface. The operation that allows clients to update an offline order has been deleted (<*OfflineOrder*). We can see a state pair ($offlineOrdered$, $_offlineOrdered$) linked by a dashed edge labelled with the change *deletion*. The deleted operation is <*OfflineOrder* shown by a dotted arrow. Other changes (*addition* of <*Transfer* and modification of >*STN* by >*ASN*) are pinpointed as well.

5 Related work

Compatibility test of interfaces has been widely studied in the context of Web service composition. Most approaches that deal with the behavioural dimension of interfaces rely on equivalence and similarity techniques to check, *at design time*, whether or not interfaces described for instance by automata are compatible [4,2]. These techniques usually rely either on trace equivalence checking or on (bi-)simulation algorithms [9]. However, these approaches do not deal with pinpointing exact locations of incompatibilities. In [13], the authors address the issue of runtime replaceability of services by extending the notions of design-

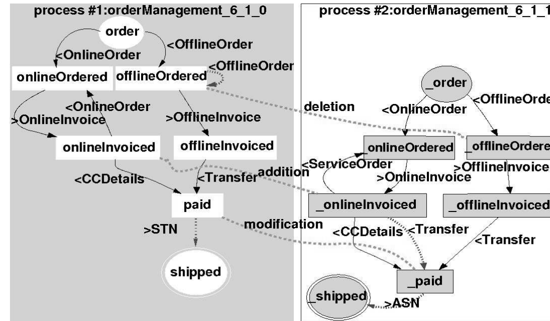

Fig. 9. Graphical output of BESERIAL on the running example.

time replaceability defined in [2] which are based on trace comparison. Again, this work does not aim at pinpointing a complete set of differences between service behaviours as we do in our work.

Recent research has addressed interface similarity measures issues. In [8], the author presents a similarity measure for labeled directed graphs inspired by the simulation and bi-simulation relations on labeled 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 [5], but do not pinpoint behavioural differences between services as our work does.

In [11], the authors propose an operator *match* which is a similarity function comparing two interfaces by finding correspondences between models. The similarity measure is a heuristics which returns a value calculated according to changes involving the addition or the deletion of an operation. However, the result does not pinpoint the exact location of these changes. In [15], 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 diagnostics such as pinpointing specific states of the two services are different, which is the goal of our work.

6 Conclusion and future work

We presented a technique to detect changes (*addition*, *deletion* or *modification* of an operation) that give rise to behavioural incompatibilities between two services. The originality of this technique is that the detection algorithm does not stop at the first incompatibility encountered but tries to seek further to identify a series of incompatibilities between two services.

Ongoing work aims at extending BESERIAL towards 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 service designers in determining how to address an incompatibility. Also, BESERIAL currently as-

sumes synchronous communication. Future work will aim at supporting asynchronous communication. We foresee that the incompatibility detection algorithm can be extended in this direction by maintaining a buffer of unconsumed messages during the traversal, along the lines of [10].

References

1. A. Aït-Bachir, M. Dumas, and M.-C. Fauvet. BESERIAL: Behavioural Service Interface Analyser. In *Proc. of the 6th International Conference on Business Process Management (BPM) – Prototype Demonstration Track, Italy.* Springer, 2008.
2. B. Benatallah, F. Casati, and F. Toumani. Representing, analysing and managing web service protocols. *Data and Knowledge Engineering, Elsevier*, 58(3):327–357, September 2006.
3. 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.
4. 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.
5. J. C. Corrales, D. Grigori, and M. Bouzeghoub. BPEL processes matchmaking for service discovery. In *OTM Conferences*, LNCS 4275, France, 2006. Springer.
6. X. Fu, T. Bultan, and J. Su. Synchronizability of conversations among web services. *IEEE Transactions on Software Engineering*, 31(12), 2005.
7. 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.
8. N. Lohmann. Correcting deadlocking service choreographies using a simulation-based graph edit distance. In *Proc. of the Int. Conf. on BPM*, number 5240 in LNCS, Milano, Italy, September 2008. Springer.
9. A. Martens, S. Moser, A. Gerhardt, and K. Funk. Analyzing compatibility of bpel processes. In *Proc. of the Advanced Int. Conf. on Telecom. and Int. Conf. on Internet and Web Applications and Services*, French Caribbean, 2006. IEEE.
10. H. R. Motahari-Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-automated adaptation of service interactions. In *Proc. of the 16th WWW Int. Conf.*, Canada, 2007. ACM.
11. 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.
12. 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.
13. 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. *ACM Transactions on the Web*, 2(2):46, April 2008.
14. S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. Ferguson. *Web Services Platform Architecture.* Prentice Hall, 2005.
15. A. Wombacher, P. Fankhauser, B. Mahleko, and E. Neuhold. Matchmaking for business processes based on choreographies. In *Proc. of the Int. Conf. on Multimedia and Expo*, Taipei, Taiwan, March 2004. IEEE Computer Society Press.