

Compatibility and Reparation of Web Service Processes

Yuhong Yan^{1*}, Ali Ait-Bachir², Min Chen¹, Kai Zhang³
 Concordia University, Montreal, Quebec, Canada¹
 University of Grenoble, LIG (MRIM), Grenoble, France²
 Huazhong Normal University, Wuhan, Hubei, China³
 yuhong@encs.concordia.ca, Ali.Ait-Bachir@imag.fr

Abstract

When two Web services work together, they exchange messages in a predefined interface process. Two interface processes should be compatible when they can work properly. Our idea to fix incompatibility problem in service processes is to change an incompatible process so that the new process can simulate a compatible process. We consider not only the control flow but also the data flow in modeling the processes into FSMs. We present a technique that not only detects the incompatibility, but also provides resolution strategies to generate the new process.

1 Introduction

The work presented here focuses on structural and behavioural interfaces and is complementary to other work which has studied the problem of interface incompatibility [1]. We also consider data dependencies of variables in web service processes to diagnose incompatibilities [8]. The study described in this paper aims at providing a tool which is capable of reporting incompatibilities between two service interfaces. Its main contribution is an algorithm which detects *all* differences that cause two service interfaces not to be compatible from a behavioural viewpoint. Among all detected differences, the algorithm detects those which are reconciliable and generates an interface to fix it.

The paper is structured as follows. Section 2 models service interfaces according to their behavioural and structural dimensions. Section 3 presents the principle of the proposed approach while Section 4 details the detection and resolution technique. Section 5 compares the proposal with related ones, and Section 6 concludes and sketches further work.

*This work is supported by project Building Self-Manageable Web Service Process of Canada NSERC Discovery Grant, RGPIN/298362-2007

2 Modelling Control Flow and Data Flow

Following [5], 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 operations (i.e. messages to be sent or received). When a message is sent or received, the corresponding transition is fired. To model data dependency, we introduce pre and post conditions on transition function. As a motivating example, we consider services that handle purchase orders processed either online or offline. Figure 1 depicts FSMs of provided interfaces P and P' .

The operation has polarity $>$ when the corresponding message m is to be sent, otherwise its polarity is $<$ (the corresponding message is to be received). The names given to states do not have any semantics. Each conversation initiated by a client starts an execution of the corresponding FSM. The figure shows also all differences between P and P' . How to detect and localise these differences is discussed in the next section.

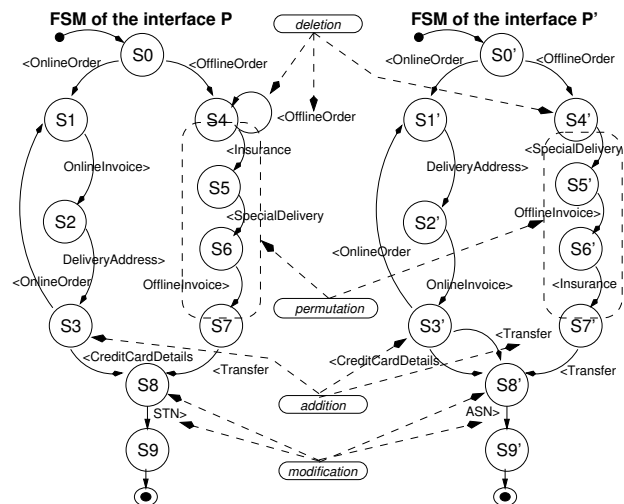


Figure 1. FSMs modelling P and P' .

Definitions and notations: We consider not only the control flow but also the data flow in a BPEL process. Using the notation in [8], assume a BPEL process has a finite set of variables $V = \{v_1, \dots, v_n\}$, and the domain $\mathcal{D} = \{D_1, \dots, D_n\}$ for V is real values \mathbb{R} or arbitrary strings. $C = \{c_1, \dots, c_m\}$ is a finite set of constraints. A constraint c_j of some arity k is defined as a subset of the cartesian product over variables $\{v_{j_1}, \dots, v_{j_k}\} \subseteq V$, i.e. $c_j \subseteq D_{j_1} \times \dots \times D_{j_k}$, or a first order formula over $\{v_{j_1}, \dots, v_{j_k}\}$. A constraint restricts the possible values of the k variables. A Web service process is a tuple (S, E, C, T, s_0, F) :

Definition 1 A Web service process can be defined as an FSM (S, E, C, T, s_0, F) such that S is a finite set of states, E is a set of events (operations), C is a constraint set that defines states S , T is a set of transition functions ($T : S \times 2^C \times E \rightarrow S \times 2^C$), s_0 is the initial state such as $s_0 \in S$ and F the set of final states such as $F \subseteq S$.

We define a labeling function $L : S \rightarrow 2^C$ that associates a state s with a set of constraints. Then the precondition of a transition t is $pre(t) \subseteq L(\circ t)$ and its post condition $post(t) \subseteq L(t \circ)$. The \circ operator (respectively \bullet) is defined in the following notations: $s \bullet$ is the set of outgoing transitions of s , $t \circ$ is the target state of the transition t , $Label(t)$ is the label of t and $|X|$ is cardinality of X .

The \circ operator (respectively \bullet) is generalized to a set of transitions (respectively states). For example, if $T = \bigcup_{i=1}^n \{t_i\}$ then $T \circ = \bigcup_{i=1}^n \{t_i \circ\}$ and $pre(T) = \bigcup_{i=1}^n \{pre(t_i)\}$; where $n = |T|$. Similarly, operator $Label$ is generalized to a set of transitions.

We can also write a transition as follows: $t : (s_1 \wedge pre(t)) \xrightarrow{et} (s_2 \wedge post(t))$. We note $t.InVar$ for the set of input variables in $pre(t)$, and $t.OutVar$ for the set of output variables in $post(t)$.

Variable dependency. When considering data flow, one transition can depend on the its upper stream transitions, if its input variables are modified by these transitions. Formally, if $t_1.InVar \cap t_2.OutVar \neq \emptyset$, and t_2 is executed before t_1 , t_1 and t_2 are dependent. Otherwise, they are independent. Variable dependency analysis is known in program slicing [7]. We can consider each transition in a process as a function that has a set of input parameters and produces a set of output parameters. Then a transition is a definer of the output variables and a utilizer of the input variables. [8] has a more complete analysis to apply program slicing to BPEL.

3 Diagnosing Incompatibilities

We know that if P' simulates P , P' is also a compatible interface. If they are not the same or do not have the

simulation relation, we need to detect the discrepancy and diagnose if they are fixable. In this section, we discuss how we interpret the discrepancy and fixing strategies.

3.1 Fixable incompatibilities

Figure 1 illustrates a situation where a fixable incompatibility can be diagnosed between two interfaces (see a permutation fragment of P and P' interfaces). This situation occurs when all the transitions on a path starting in P with $\langle \text{Insurence} \rangle$ and finishing with $\langle \text{OfflineInvoice} \rangle$ are also enabled in FSM P' , but in a different order. In other words, the path in P defined by the sequence of transition labels $[\langle \text{Insurence}, \langle \text{SpecialDelivery}, \text{OfflineInvoice} \rangle]$ is a permutation of the path in P' defined by $[\langle \text{SpecialDelivery}, \text{OfflineInvoice} \rangle, \langle \text{Insurence} \rangle]$.

A trace p of an FSM between two states s_1 and s_2 is a sequence of contiguous transitions starting from s_1 and ending at s_2 . Multiple traces can exist between s_1 and s_2 . If a trace $p(s_1, s_2)$ can be permuted into another trace $p'(s'_1, s'_2)$, the first condition to satisfy is that they should have the same number of transitions. Transitions can be one-to-one paired by the same event they emit (Property 1).

Property 1 The transitions $\{t\} \in p(s_1, s_2)$ and $\{t'\} \in p'(s'_1, s'_2)$ can be matched by an bijection function $f: \exists f, f : t \mapsto t'$, where $e = e'$ that $e \in t$ and $e' \in t'$.

When considering data flow, the inputs of a transition should not be affected by permutation operation. For a process P , we can build a variable dependency table. The table records which operation/transition defines or utilizes a variable. From the table, we can get all the transitions that can change the input variables of t : $D(t) = \bigcup_{v \in t.InVar} D_v$ where D_v is the set of definers of v . Define $PathSet(\circ t)$ is the set of all possible traces from the initial state s_0 to t , and $PathSet'(t \circ)$ is the set of traces after t is put in a new place by permutation. The slice of $t.InVar$ is the set of traces that affects the values of $t.InVar$ when t is triggered. It can be calculated by a simplified version of program slicing applied to Web service process [8]. If t is moved by a permutation and t is triggered at its new position then input values of $t.InVar$ should be the same as before performing the permutation. This means the following property should be staisfied.

Property 2 The slice of $t.InVar$ on $PathSet(\circ t)$ and the slice of $t.InVar$ on $PathSet'(t \circ)$ have the same set of traces.

Proposition 1 A trace $p(s_1, s_2)$ can be permuted to $p'(s'_1, s'_2)$, iff $p(s_1, s_2)$ and $p'(s'_1, s'_2)$ satisfy Property 1 and each transition t in $p(s_1, s_2)$ satisfies Property 2.

3.2 Deletion of an operation

Figure 1 depicts a situation where an operation appears in P and not in P' . We observe that all operations enabled in state $S4'$ are also enabled in state $S4$. The reverse is not true, because there is an operation (namely $\langle \text{OfflineOrder} \rangle$) enabled in state $S4$ that has no match in state $S4'$.

Once this difference has been detected, the pair of states to be examined next in the process of comparing P and P' is $\langle S5, S5' \rangle$.

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:

Property 3 $(s \bullet \supseteq s' \bullet) \vee \exists t \in s \bullet : t \notin s' \bullet \wedge (t \circ) \bullet \subseteq s' \bullet$

3.3 Addition of an operation

Addition problem is symmetric to the deletion problem that some transitions in interface P' are not in P . The fixing strategy is to delete them from P' . However, we need to consider that the removal should not change the inputs of its following transitions. We need to satisfy the following property to fix the addition problem:

Property 4 $(s' \bullet \supseteq s \bullet) \wedge \forall t \in s' \bullet \setminus s \bullet : t.OutVar \cap (s' \bullet).InVar = \emptyset$ or $\exists t \in s' \bullet : t \notin s \bullet \wedge (t \circ) \bullet \subseteq s \bullet \wedge t.OutVar \cap ((t \circ) \bullet).InVar = \emptyset$

3.4 Modification of an operation

Figure 1 shows a situation where we can diagnose that operation $\text{STN} \rangle$ has been replaced by operation $\text{ASN} \rangle$ (i.e. a modification). The reason is that the operation $\text{STN} \rangle$ is enabled in $S8$ but not in $S8'$, and conversely $\text{ASN} \rangle$ is enabled in $S8'$ but not in $S8$. Moreover, the transition labelled $\text{STN} \rangle$ does not match to any transitions t' in state $S9'$ such that operation $\text{STN} \rangle$ occurs downstream along the branch starting with t' , and symmetrically, $\text{ASN} \rangle$ does not match any transitions t of state $S9$ such that $\text{ASN} \rangle$ occurs downstream along the branch starting with t . Thus we can not diagnose that $\text{STN} \rangle$ has been deleted, nor can we diagnose that $\text{ASN} \rangle$ has been added.

The pair of states to be visited next in P and P' is such that both transitions involved in the modification are traversed simultaneously ($\langle S9, S9' \rangle$). Formally, a modification is diagnosed in state pair $\langle s, s' \rangle$ if the following condition holds:

Property 5 $\exists t1 \in s \bullet, \exists t1' \in s' \bullet : t1 \notin s' \bullet \wedge t1' \notin s \bullet \wedge \neg \exists t2 \in s \bullet : t1' \in (t2 \circ) \bullet \wedge \neg \exists t2' \in s' \bullet : t1 \in (t2' \circ) \bullet$

4 Detection and Resolution

In this section, we want to use the strategies of *permutation, deletion, addition and modification* discussed in the previous section to fix an interface. This is a process to detect the discrepancy of the given process P_1 to a correct standard interface P_2 and repair it using the four listed strategies. **The objective of incompatibility resolution is to change P_1 so that it can simulate P_2 .** We constrain the following discussion to Acyclic Deterministic Finite State Automata (ADFA). Here are some guidelines when we apply the four repair strategies: i) When P_1 and P_2 have shared operations, try to use permutation. Permutation is considered low cost of repair, because it tries to keep most of the transitions; ii) Topological differences are resolved by deleting and adding transitions; iii) Any other differences are resolved by modification.

4.1 Get initial deletion and addition sets

First, we need Definition 4.1 for calculation.

Definition 2 A *projection of an automaton* $P = (S, E, T, s0, F)$ on a subset of its events $E' \subseteq E$ is an automaton $P' = (S', E', T', s0, F)$ that all the transitions with $E \setminus E'$ are removed by merging their end and starting states: i) $S' \subseteq S$; ii) $T' \subseteq S' \times E' \times S'$. If $t = (q, e, q') \in T'$, there exist a path $q \xrightarrow{u_1} q_1 \cdots \xrightarrow{u_m} q_m \xrightarrow{e} q'$ in P with $u_i \in E \setminus E'$ and $e \in E'$.

Assume two ADFAs, $P_1 = (S_1, E_1, C_1, T_1, s0_1, F_1)$ is an interface under inspection and $P_2 = (S_2, E_2, C_2, T_2, s0_2, F_2)$ is a correct standard interface. We project P_1 and P_2 on their shared events $E_1 \cap E_2$ and get the following definitions:

- The addition set for P_1 is $T_a = \{t|t : (s, e, s'), e \in E_2 \setminus E_1, t \in T_2\}$;
- The deletion set for P_1 is $T_d = \{t|t : (s, e, s'), e \in E_1 \setminus E_2, t \in T_1\}$;

T_a and T_d are the transitions removed from T_2 and T_1 due to projection respectively.

4.2 Detect the permutation segments

We project both P_1 and P_2 on $E_1 \cap E_2$ to obtain two AFDAs that have the same events (aka the operations). We still use P_1 and P_2 after their projection. Now we check if we can permute P_1 to P_2 .

Definition 3 The *labeling function* $l(s, s_1)$ of an AFDA $P = (S, E, T, s0, F)$ is a mapping between a state $s \in S$ to a multiset that each element of the multiset is a set of event names on one trace from s_1 to s : $l(s, s_1) : s \rightarrow \{\{e \in E\}\}$.

A trace is mapped to a label. We use multiset because a trace can contain multiple transitions of the same event (the same operation multiple times) and multiple traces between s and s_1 can contain the same event sequence.

Definition 4 A binary relation $\mathcal{R} \subseteq S_1 \times S_2$ is such that if $(s_1, s_2) \in \mathcal{R}$, and labeling functions $l(s_i, s_1)$ of P_1 and $l(s_j, s_2)$ of P_2 label s_i and s_j with at least one common label, then $(s_i, s_j) \in \mathcal{R}$. And there exists a matching pair of traces $(s_1 \xrightarrow{p_1} s_i)$ and $(s_2 \xrightarrow{p_2} s_j)$ with the same events.

Definition 4 is similar to binary relation in simulation, except that states and transitions between two matching states do not match. Applying Definition 4 to P_1 and P_2 using labeling function from the initial states s_{01} and s_{02} , we can get matching pairs of segments that the starting and the ending segments are matching states, and the segments are composed by the same set of transitions but in different sequential order. These segments are fixed by permutation.

Proposition 2 Matching traces $(s_1 \xrightarrow{p_1} s_i)$ and $(s_2 \xrightarrow{p_2} s_j)$ can be fixed by permutation if they satisfy Property 2.

If some segments cannot be fixed by permutation, we put them into a modification transition set T_m .

4.3 Reexamine deletion and addition sets

After the labeling process, we can re-examine the deletion and the addition sets to check if there are matching traces with the following conditions:

Definition 5 Assume a trace $p_1 : s_1 \xrightarrow{p_1} s_2$ composed by transitions from T_a , and a trace $p_2 : s'_1 \xrightarrow{p_2} s'_2$ composed by transitions from T_d , if $(s_1, s'_1) \in \mathcal{R}$ and $(s_2, s'_2) \in \mathcal{R}$ then p_1 maps to p_2 .

We also consider to fix them using modification strategy instead of deleting them and adding a new set of operations to replace them. Therefore, we put p_2 into the modification set $T_m = T_m \cup p_2$ and remove them from addition set and deletion set: $T_a = T_a \setminus p_1$ and $T_d = T_d \setminus p_2$.

5 Related Work

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 [2]). Checking interface compatibility is thus based on bi-similarity algorithms [4]. These approaches do not deal with pinpointing exact locations of incompatibilities as ours

does. In [6], authors propose an operator *match* which is a similarity function comparing two interfaces for finding correspondences between them. The similarity measure is an heuristic which relies on penalty scores associated with the type of change (addition vs. deletion). However, the result does not pinpoint the exact location of these changes. Similar algorithms with the same limitations and complexity has been used for service discovery purpose as introduced in [3]. Authors use graph matching algorithms based on graph edit distance and propose heuristics to limit the exponential complexity of exhaustive search algorithms.

6 Conclusion and Further Study

We present a technique to detect and resolve incompatibility in service interface processes. We consider not only the control flow but also the data flow when we model the interface processes into FSM. Our resolution strategy tries to change an incompatible process by permuting, adding, deleting, and modifying the transitions in the original process, so that the fixing process can simulate a compatible interface process. Yet, we need to extend our work on cyclic FSMs and present criteria to evaluate the impact of changes.

References

- [1] A. Ait-Bachir and M.-C. Fauvet. Diagnosing and measuring incompatibilities between pairs of services. In *Proc. of the 20th Int. Conf. on Database and Expert Systems Applications*, number 5690 in LNCS, Austria, 2009. Springer.
- [2] L. Bordeaux, G. Salaün, D. Berardi, and M. Mecella. When are two web services compatible? In *Proc. 5th Int. Conf. on Technologies for E-Services*, Canada, 2004. Springer.
- [3] R. M. Dijkman, M. Dumas, and L. García-Bañuelos. Graph matching algorithms for business process model similarity search. In *Proc. of the 7th Int. Conf. on Business Process Management*, LNCS 5701, Germany, 2009. Springer.
- [4] A. Martens, S. Moser, A. Gerhardt, and K. Funk. Analyzing compatibility of bpm processes. In *Proc. of the Advanced Int. Conf. on Telecommunications and Int. Conf. on Internet and Web Applications and Services*, French Caribbean, 2006. IEEE.
- [5] 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.
- [6] 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.
- [7] M. Weise. Program slicing. *IEEE Transactions on Software Engineering*, 10(4), 1984.
- [8] Y. Yan, P. Dague, Y. Pencole, and M.-O. Cordier. A model-based approach for diagnosing faults in web service processes. *Int. Journal on Web Service Research*, 6(1), 2009.