

# Assessment of Service Protocols Adaptability Using A Novel Path Computation Technique\*

Zhangbing Zhou<sup>1</sup>, Sami Bhiri<sup>1</sup>, Armin Haller<sup>1</sup>,  
Hai Zhuge<sup>2</sup> and Manfred Hauswirth<sup>1</sup>

<sup>1</sup> Digital Enterprise Research Institute, National University of Ireland at Galway  
`firstname.lastname@deri.org`

<sup>2</sup> Institute of Computing Technology, Chinese Academy of Sciences  
`zhuge@ict.ac.cn`

**Abstract.** In this paper we propose a new kind of adaptability assessment that decides whether service protocols of a requestor and a provider are adaptable, computes their adaptation degree, and identifies conditions that determine when they can be adapted. We also propose a technique that implements this adaptability assessment: (1) we construct a complete adaptation graph that captures all service interactions adaptable between these two service protocols. The emptiness or non-emptiness of this graph corresponds to whether or not they are adaptable; (2) we propose a novel path computation technique to generate all instance sub-protocols which reflect valid executions of a particular service protocol, and to derive all instance sub-protocol pairs captured by the complete adaptation graph. An adaptation degree is computed as a ratio between the number of instance sub-protocols captured by these instance sub-protocol pairs with respect to a service protocol and that of this service protocol; (3) and finally we identify a set of conditions based on these instance sub-protocol pairs. A condition is the conjunction of all conditions specified on the transitions of a given pair of instance sub-protocols. This assessment is a comprehensive means of selecting the suitable service protocol among functionally-equivalent candidates according to the requestor's business requirements.

## 1 Introduction

Given the inherent autonomy, heterogeneity, and continuous evolution of Web services, mediated service interactions are a common style of Web service interactions [8]. Hence, adaptability assessment is as important as compatibility analysis that targets direct service interactions. Following [3, 4], by adaptation we mean the act of identifying, classifying, and reconciling mismatches between service behavioral interfaces (the so-called service protocols) [5]. Adaptability assessment is further defined as the act of deciding whether or not service protocols of a requestor and a provider are adaptable without constructing an adapter,

---

\* The work presented in this paper has been funded by Science Foundation Ireland under Grant No. SFI/08/CE/I1380 (DERI Lion-2).

computing their adaptation degree, and identifying conditions that determine when these two service protocols can be adapted. This assessment enables a requestor to identify and thus to select the most suitable service provider among functionally-equivalent candidates according to her business requirements.

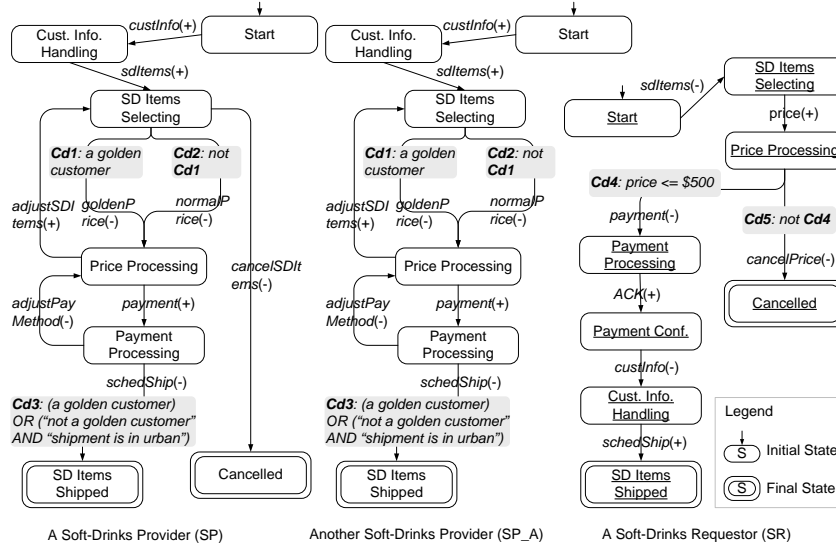
As reviewed by Dumas et al. [5], previous works related to service interaction analysis mainly focus on either compatibility analysis [1, 2] or adapter construction [3, 4, 6, 8, 12, 13]. Adaptability can somehow be studied by techniques that construct adapters. However, these approaches are inadequate to provide the level of assessment we target. Indeed, being able to build an adapter merely implies that there are some situations where service protocols are possibly adaptable, while unsupported scenarios (due to un-reconcilable deadlocks [6] for instance) are excluded from the adapter protocol specification. An adapter does not differentiate the different possibilities of adaptability (i.e., the adaptation degree) between adaptable service protocols, and hence, these protocols are assumed to be the same to the requestor although they are actually different in the adaptation possibility. In addition, an adapter does not specify conditions that determine when two service protocols are adaptable. Consequently, these two service protocols may behave in such a way that their interaction fails either because it is an unsupported scenario, or because some conditions are not satisfiable according to the exchanged message instances.

### 1.1 Motivating Example

Fig. 1 depicts, using guarded finite state automata (i.e., GFSA), the service protocols of two soft-drinks provider services (denoted  $SP$  and  $SP\_A$ ) and a possible interaction with a soft-drinks requestor service (denoted  $SR$ ). A requestor, using  $SR$ , intends to buy soft-drinks online through one of these two provider services. Guards denoted  $Cd_i$  ( $i \in [1, 5]$ ) correspond to conditions in the BPEL specification. So we use guard and condition interchangeably in this paper.

The only difference between  $SP$  and  $SP\_A$  is that  $SP$  allows canceling an interaction if some selected soft-drink is presently out of stock.  $SP$  and  $SP\_A$  may apply a discount on the price depending on a *custInfo*. Hence, a *custInfo* is expected by  $SP$  (or  $SP\_A$ ) before the price is decided. Due to privacy concerns,  $SR$  only sends *custInfo* if the *price* is acceptable. This example shows a deadlock that occurs when  $SR$  directly interacts with  $SP$  (or  $SP\_A$ ).  $SP$  (or  $SP\_A$ ) is requesting a *custInfo* before sending a *price*, while  $SR$  is expecting a *price* before deciding upon continuation and sending *custInfo* or not. Adapters, such as the ones described in [13, 6], can circumvent this problem if either (1) one of the receiving transitions is neither control nor mandatorily data dependent on one of its direct-succeeding-transitions [13], and hence, a *mock-up custInfo* message can be generated to resolve this deadlock, or (2) the adapter developer can provide a *custInfo* message using *evidences* [6], and thus, this deadlock is reconciled.

The *custInfo* is used for deciding whether or not a discount is to be applied. However, a *normalPrice* is always applied by default. For the requestor, both  $SP$  and  $SP\_A$  are adaptable with  $SR$  (according to [13, 6]) if the *price* is acceptable. Naturally, the requestor also wants to know whether the conclusion also holds



**Fig. 1.** Service protocols for soft-drink providers and requestor ( $SP$ ,  $SP_A$  and  $SR$ )

when the *price* is unacceptable, so that the interaction can be cancelled if the price is unacceptable.  $SP$  can support this scenario according to [13, 6], but,  $SP_A$  does not support this interaction because the interaction cannot be cancelled after the *Price Processing*.

Consequently, the requestor is given an initial impression that  $SP$  is more adaptable with  $SR$  than  $SP_A$ . If an expected interaction (for instance, the soft-drink items are to be shipped) can be supported by both  $SP$  and  $SP_A$ ,  $SP$  is a more suitable candidate service provider because additional interactions (for instance, the interaction is to be cancelled) are also supportable by  $SP$  but not by  $SP_A$ . An adapter is ineligible to detect this difference since, in principle, an adapter can detect whether or not, but not how far, an adaptation is possible.

The reasoning so far only explores the legal message exchanges between  $SP$  (or  $SP_A$ ) and  $SR$ . The success of an interaction in general and an adaptation in particular, however, depends also on conditions that guide which branches to follow and check whether transitions can be enabled. For instance, for an adaptation between  $SP$  (or  $SP_A$ ) and  $SR$  leading to soft-drink items being shipped, a prerequisite is that  $Cd_2$ ,  $Cd_3$  and  $Cd_4$  are able to be respected. Hence,  $Cd_2 \wedge Cd_3 \wedge Cd_4$  is a condition that determines when  $SP$ , as well as  $SP_A$ , can be adapted with  $SR$ . For another adaptation between  $SP$  and  $SR$  leading to the interaction to be cancelled, a prerequisite is that  $Cd_2$  and  $Cd_5$  are able to be respected. However, this cancellation scenario is not supportable for  $SP_A$  and  $SR$ , and hence,  $Cd_2 \wedge Cd_5$  is a condition for  $SP$  and  $SR$ , but is not a condition for  $SP_A$  and  $SR$ , that determines when they can be adapted.

To summarise, the conditions and the adaptation possibility (or the adaptation degree) are two complementary criteria to the requestor for selecting the

suitable candidate service provider. Based on the conditions, the requestor first filters candidate service providers through checking whether the expected interactions can be supported. Thereafter, the requestor can choose a suitable service provider (possibly with the highest adaptation degree) among these functionally-equivalent candidates. The selected service provider is ensured to be interacted properly with the requestor service for achieving a certain goal.

The identification of these conditions, as well as the computation of the adaptation degree as mentioned above, is beyond existing adapter construction approaches, but is the concern of this paper. In the following sections, we show how they can be addressed in this particular example and in general.

## 1.2 Overview of Our Approach

By the adaptability assessment we mean the following three aspects: deciding whether or not service protocols of a requestor and a provider are adaptable, computing their adaptation degree, and identifying conditions that determine when they can be properly adapted. In the following, we show how these three aspects are to be addressed.

1. We first construct a complete adaptation graph that captures all legal message exchanges between service protocols of a requestor and a provider (Section 3). A *complete* sequence of legal message exchanges reflects a mediated service interaction between a pair of instance sub-protocols of these two service protocols. An instance sub-protocol is a part of a service protocol that may be executed for a particular instance of this service protocol (Section 2). The emptiness or non-emptiness of the complete adaptation graph corresponds to the fact that whether these two service protocols are adaptable.
2. Assume  $m$  is the number of instance sub-protocols captured by the complete adaptation graph with respect to a certain service protocol  $p$ , and  $n$  is the number of instance sub-protocols specified by the specification of  $p$ . An adaptation degree is computed as a ratio between  $m$  and  $n$  (Section 5.1). Based on a novel path computation technique we propose in Section 4, we compute all instance sub-protocols for a service protocol specification (Section 4), and derive all instance sub-protocol pairs captured by the complete adaptation graph (Section 5.2).
3. Based on all instance sub-protocol pairs captured by the complete adaptation graph, we identify a set of conditions that determine when an adaptation is possible. For each pair, a condition is provided which corresponds to the conjunction of all conditions specified on the transitions of these two instance sub-protocols (Section 5.1).

Finally, we review the related work and conclude this paper in Section 6.

Our adaptability assessment builds upon our Space-based Process Mediator (SPM) that is detailed in our previous work [13]. It is important to notice that our technique is general and can be applied to other adapters as well.

This SPM is able to reconcile the deadlock in our motivating example in the *SD Items Selecting* state by providing a *mock-up custInfo* which is replaced

whenever the *custInfo* arrives from *SR* to enable the *schedShip(-)* transition. Generally, the SPM can circumvent this kind of deadlock, if the dependency that exists between one of the receiving transitions with the state *enabled* and one of its direct- succeeding-transitions is neither control nor mandatorily data dependent. Then, the SPM can produce a *mock-up* message that acts as the data expected by this receiving transition. This *mock-up* message is replaced (through the space-based mechanism of the SPM) by a concrete message whenever it is produced by a peer protocol. More detail about the SPM is available at [13].

## 2 Preliminaries: Service Protocol, Control and Data Dependencies, and Instance Sub-Protocol

Following [2], we adopt deterministic finite state automata for modeling service protocols, where transitions are triggered by message exchanges among partners. It should be noted that, a service protocol, or more generally a state automata, is sequentially executed. In other words, a state in a service protocol specification allows multiple transitions to follow, and hence, can lead to multiple states. However, at a certain point at runtime, only one transition can be enabled, and then be fired, and consequently, this state evolves to one of the following states.

In [2], the authors claim that different message and condition pairs are always possible to be mapped into new distinct message labels, and hence, conditions can be abstracted away. However, this simplification loses important information of conditions. For instance, the price in *SP* is mapped into *goldenPrice* and *normalPrice*. If *Cd<sub>1</sub>* and *Cd<sub>2</sub>* are not specified, the rationale for choosing *goldenPrice* or *normalPrice* is lost. Indeed, conditions are fundamental to retrieve prior known service protocols and to ensure possible interactions between these service protocols. Hence, we model service protocols in terms of GFSA. Note that the time property [17] is also an essential dimension of a service protocol specification, which is considered as our future work.

Formally, a service protocol is a tuple  $p = (M, S, s, F, C, T)$ , where  $M$  is a finite set of messages. Following [2], for each message  $m \in M$ , we use notations:  $m(+)$  and  $m(-)$ , to denote the incoming or outgoing of  $m$ .  $S$  is a finite set of states, where  $s$  is the initial state and  $F$  is a finite set of final states.  $C$  is a finite set of conditions.  $T \subseteq S^2 \times M \times C$  is a finite set of transitions. Each transition  $\tau = (s_s, s_t, m(+/-), c \mid true) \in T$  defines a source state  $s_s$ , a target state  $s_t$ , an incoming or outgoing message  $m$ , and a condition  $c \in C$  where *true* is the condition of default. Transitions are semantically described by specifying their *input*, *output*, *precondition*, and *effect* (i.e., IOPE in OWL-S specification).

As presented in [9], different kinds of dependencies can exist between transitions. These dependencies are often obfuscated by a service protocol specification that defines all possible execution sequencings of transitions. A sequencing constraint in a service protocol may originate from one or multiple kinds of dependencies [9]. We consider two kinds, namely control and data dependencies.

A transition  $\tau_b$  is control dependent on another transition  $\tau_a$  if the *completion* of  $\tau_a$  (marked by its *effect*) is a necessary condition for the *enablement* of  $\tau_b$

(guarded by its *precondition*). Data dependencies are classified as mandatory or optional.  $\tau_b$  is mandatorily data dependent on  $\tau_a$  if common data exist between the *output* of  $\tau_a$  and the *input* of  $\tau_b$ , whereas  $\tau_b$  is optionally data dependent on  $\tau_a$  if no common data exist between the *output* of  $\tau_a$  and the *input* of  $\tau_b$ , but, incoming conditions of  $\tau_b$  use the data in the *output* of  $\tau_a$ . We extract control and data dependencies from the semantic description of transitions [14].

Next, we introduce the concept of an instance sub-protocol. An instance sub-protocol represents a *valid* part of a service protocol that may be executed for a particular instance of this service protocol. *Valid* means free of dependency conflicts. Generally, if the destination transition of a dependency relation is in an instance sub-protocol, the source transition of this dependency relation must be in this instance sub-protocol as well. An instance sub-protocol itself is a *smaller* service protocol in size. For instance,  $SP_A$  is an instance sub-protocol of  $SP$ .  $SP$  itself is not since no execution can lead a service protocol to two final states.

Formally, an instance sub-protocol of a service protocol  $p = (M, S, s, F, C, T)$  is a tuple  $ISP = (M_I, S_I, s_I, f_I, C_I, T_I)$  where: (1)  $M_I \subseteq M$ ,  $S_I \subseteq S$ ,  $s_I = s$ ,  $f_I \in F$ ,  $C_I \subseteq C$ , and  $T_I \subseteq T$ ; (2)  $ISP$  is valid with respect to control and data dependencies of  $p$ ; (3) there exists an execution instance of  $p$  with  $T_e$  as the set of transitions to be enabled and executed in this instance, then,  $T_e = T_I$ .

### 3 Service Protocols Adaptation Graph

This section addresses a joint analysis of two service protocols, that of a requestor and a provider, to decide whether they are able to be adapted. Intuitively, we need to examine all pairs of instance sub-protocols in these two service protocols to study whether each pair is adaptable. However, a service protocol may have many instance sub-protocols if it contains loop segments (for instance,  $SP$  has 17 instance sub-protocols, while  $SR$  has only 2). In addition, some instance sub-protocols in a service protocol may be much similar in their specifications, i.e., they share some transitions. Hence, an examination following this brute-force strategy is usually inefficient.

Inspired by the complete interaction tree proposed in [2], we introduce the notion of an adaptation graph that explores possible mediated service interactions between two service protocols. A mediated service interaction captures a legal message exchange sequence of these two service protocols between a pair of their instance sub-protocols. As such, a mediated service interaction corresponds to a *complete* adaptation of these two service protocols leading from their initial states to a pair of their final states.

We also define an adaptation graph using GFSA, where a state is a combination of two states of participating service protocols. A transition is either a message exchange between these two service protocols, or a service protocol sending a message through an adapter (the SPM in our case), or an adapter forwarding a message (either a concrete message generated by a partner, or a *mock-up* message generated by the SPM) to a service protocol. Conditions associated with a transition in an adaptation graph are inherited from those

of relevant transitions in service protocols. Note that conditions are not to be evaluated at the adaptation graph construction phase, since the evaluation of conditions depends on the exchanged message instances.

**Definition 1 (Adaptation Graph)** Let  $p_1 = (M_1, S_1, s_1, F_1, C_1, T_1)$  and  $p_2 = (M_2, S_2, s_2, F_2, C_2, T_2)$  be two service protocols. An adaptation graph for  $p_1$  and  $p_2$  is a tuple  $adapt_{graph} = (M, S, s, F, C, T)$ .  $M \subseteq M_1 \cup M_2 \cup M_{SPM}$ , where  $M_{SPM}$  is a finite set of mock-up messages generated by SPM. The message polarity is defined as follows: messages outgoing in  $M_1$  and  $M_2$  are sent to SPM or a peer protocol, messages incoming in  $M_1$  and  $M_2$  are sent by SPM or a peer protocol, and messages in  $M_{SPM}$  are sent to  $p_1$  or  $p_2$ .  $S \subseteq S_1 \times S_2$ , where  $s = (s_1, s_2)$  and  $F \subseteq F_1 \times F_2$ .  $C \subseteq C_1 \cup C_2$ , and  $T \subseteq S^2 \times M \times C$ .

An adaptation graph is complete if it includes all possible mediated service interactions between two service protocols. As an example, Fig. 6 in Section 5 illustrates the complete adaptation graph for  $SP$  and  $SR$  (denoted  $adapt_{graph}$ ).

Due to space considerations we refer the interested reader to our technical report [16] for the algorithm to generate a complete adaptation graph, as well as its correctness proof. In a nutshell, the algorithm traverses from the initial states of two service protocols (i.e.,  $p_1$  and  $p_2$ ) to their final states, and constructs the intermediate states through combining the intermediate states of  $p_1$  and  $p_2$  on the condition that there are legal message exchanges leading to them. A legal message exchange means either (1) a message exchange between  $p_1$  and  $p_2$ , or (2) a message reordering and remembering [3, 4, 6, 8, 13] by means of SPM, or (3) a *mock-up* message generation by the SPM if one of receiving transitions (with the state *enabled*) in  $p_1$  or  $p_2$  is neither control nor mandatorily data dependent on one of its direct- succeeding-transitions [13].

The worst case time complexity of the algorithm is  $O(k^3n^6)$  where  $k$  is the upper bound of transitions between a pair of source and target states, and  $n$  is the maximum number of states in two service protocols. Notice that as observed by [7], a service protocol tends to be a fairly simple model, because a service protocol, as well as a service in general, is designed by humans. This indicates that  $k$  is usually quite small. Moreover, the number of states in a service protocol is typically not big. Concretely,  $n$  is typically less than 100 [10], and  $n$  ranging from 50 to 100 is regarded as a large service protocol [1]. These indicate that the algorithm proposed is feasible to construct a complete adaptation graph of practical relevance. We also recognize that the size of a complete adaptation graph (i.e., the number of states and transitions) is not big. Since transitions in a service protocol are constrained by control and/or data dependencies, a state in a service protocol can combine with limited states (assume that  $l$  is the upper bound) in another service protocol which construct the states and transitions in this graph. Hence, the graph has  $l \times n$  states in maximum, and there are  $k$  transitions at most between a pair of states.

We next explore how a complete adaptation graph contributes to the adaptability assessment. Given two service protocols  $p_1$  and  $p_2$ , partial adaptability specifies that, some, but not all, instance sub-protocols in  $p_1$  can be adapted with

those in  $p_2$ , whereas full adaptability mandates that all instance sub-protocols in  $p_1$  can be adapted with those in  $p_2$ . If this graph is not empty, which means that at least one pair of instance sub-protocols in  $p_1$  and  $p_2$  can be adapted, and hence,  $p_1$  and  $p_2$  are adaptable. However, to differentiate between partial and full adaptability requires checking whether or not all instance sub-protocols are reflected by the paths in this graph. To generate all instance sub-protocols of a service protocol is not a trivial task, which will be discussed in the next section. Hence, the emptiness or non-emptiness of the complete adaptation graph corresponds to the fact that whether or not  $p_1$  and  $p_2$  are adaptable.

## 4 Instance Sub-Protocol Computation

Each path in a service protocol  $p$ , which leads from the initial state to one final state and is free of dependency conflicts, constitutes an instance sub-protocol of  $p$ . Hence, the problem of instance sub-protocols generation is reducible to that of path computation of a service protocol which is a directed cyclic graph. This path computation takes exponential time to the size of the graph in general.

As far as we know, *Path\_BTC* [11] is the only promising technique that tackles this path computation problem. However, some assumptions made may not be satisfiable in our context. *Path\_BTC* requires that two functions *concatenate* and *aggregate* are distributive. An example is given by [11] to explain this requirement as follows. Consider the case where there are two paths  $p_1$  and  $p_2$  from node  $j$  to  $k$  with associated labels  $l_1$  and  $l_2$  respectively. Let there also be a path  $p_3$  from node  $i$  to  $j$  with label  $l$ . Then the path set from  $i$  to  $k$  (denoted  $P$ ) is  $\{p_3.p_1, p_3.p_2\}$ . The symbol “.” means path concatenation. This assumption may not be satisfied in our context since, in a service protocol,  $P$  may include another path that directly links  $i$  to  $k$  with label  $l_3$ . In addition, self-loops, which are excluded in *Path\_BTC*, can exist in a service protocol.

In the following, we introduce a novel path computation technique to compute all paths in a service protocol. We use the service protocol  $SP_{Sib}$  depicted in Fig. 2 as a running example.  $SP_{Sib}$  is similar to  $SP$  apart from a self-loop specified on the state *SD Items Selecting* with a transition *moreSDItems(+)*. This path computation procedure is composed of the following sequential steps:

1. Based on a service protocol we generate a FSA that (1) abstracts away conditions and self-loops, and then, (2) folds multiple transitions that share the source and target states into a single transition. Fig. 2 illustrates the FSA generated out of  $SP_{Sib}$ , namely abstracted  $SP_{Sib}$ . Thereafter, we identify back transitions in the generated FSA (detailed in Section 4.1).
2. We then compute all paths in the generated FSA. This path computation procedure is detailed in Section 4.2.

By a path, we mean a sequence of alternating states and transitions from the initial state of this generated FSA to one of its final states. To align a path with an instance sub-protocol, we represent a path in terms of an FSA. Examples of paths are shown in Fig. 3. Note that multiple paths may

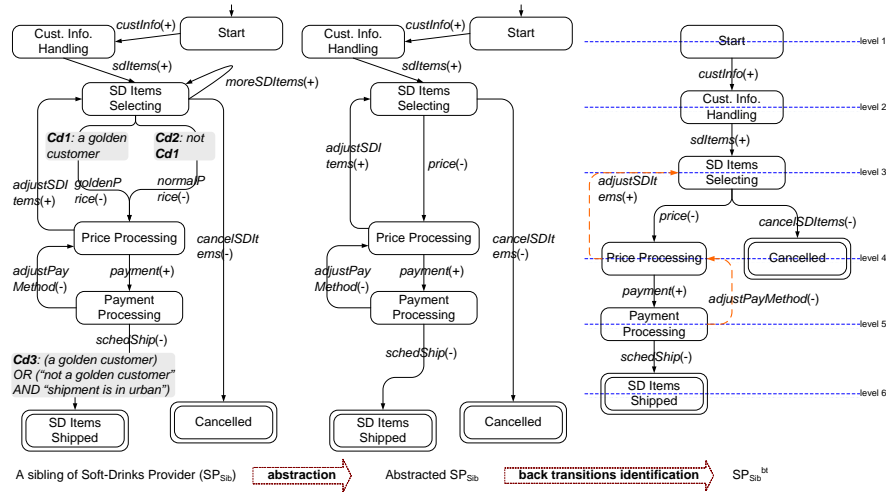


Fig. 2. Abstraction and back-transitions identification

share a same FSA representation if (1) they share the same sets of states and transitions, and (2) they are different merely because of the times to be traversed of some states and/or transitions.

- Based on the paths computed in Step 2, we generate all paths of a service protocol by (1) unfolding transitions that have been folded in the first step, by (2) considering self-loops, and by (3) integrating conditions that have been abstracted away previously. This procedure is detailed in Section 4.3. All paths free of dependency conflicts constitute the instance sub-protocols.

Except Step 2, other steps are computation light (with linear or polynomial time complexity). Regarding Step 2, as mentioned in Section 3, both a service protocol and a complete adaptation graph are typically not big in size. Hence, our technique is feasible to be used for the path computation of our context.

#### 4.1 Back-Transitions Identification

A transition  $\tau$  in the generated FSA is identified as a back-transition if (1) its target state is not a final state, and (2) the *distance* from the initial state of this FSA to the target state of  $\tau$  is not longer than that to the source state of  $\tau$ . We borrow the notion of *level* from tree automata to specify the level of the states in this generated FSA.

For instance, the level of the initial state *Start* in  $SP_{Sib}$  FSA is 1. The level of its immediately following state namely *Cust. Info Handling* is 2, and so on. The levels of states are computed using a breadth-first search, while ignoring all transitions whose level of the target state is not bigger than the level of the source state. These transitions ignored are in fact back-transitions. We refer by  $SP_{Sib}^{bt}$  in Fig. 2 to the generated FSA of  $SP_{Sib}$  where back transitions are identified and marked using dashed lines.

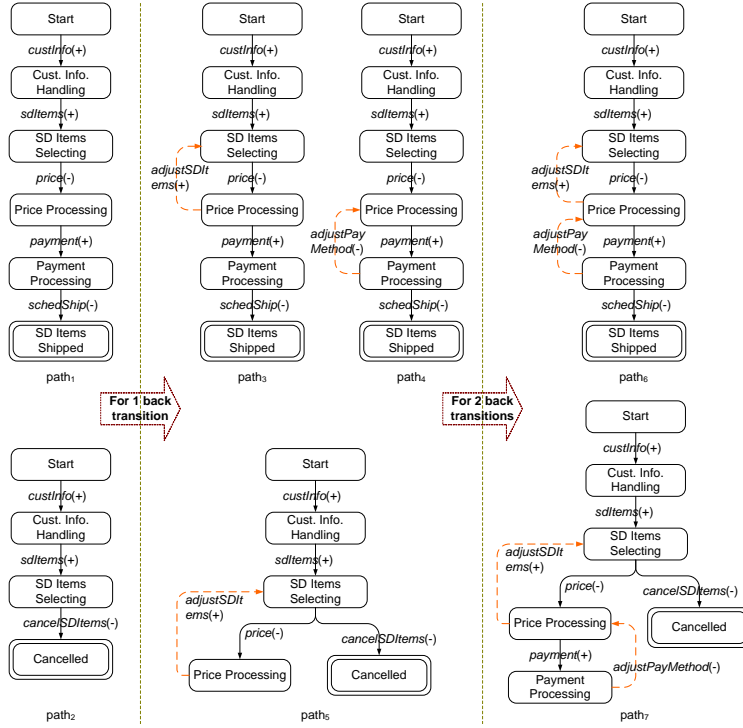


Fig. 3. Path computation

## 4.2 Path Computation of a Generated FSA

First, we generate all paths in the generated FSA while ignoring back-transitions using a breadth-first search. For instance,  $SP_{Sib}^{bt}$  has two paths  $path_1$  and  $path_2$ , as shown on the left part of Fig. 3.

Based on the paths with  $i$  back-transitions ( $i = 0, 1, 2, \dots, k-1$ , where  $k$  is the number of back-transitions in the generated FSA), we compute paths that contain  $i+1$  back-transitions. In the following we show how, given a path (denoted  $p_{th}$ ) with  $i$  back transitions and one of these  $k$  back-transition (denoted  $\tau$ ), we compute new paths with  $i+1$  back-transitions.  $s_s$  and  $s_t$  denote the source and target states of  $\tau$  respectively.

We distinguish between four situations for this path computation. The first situation corresponds to **Case 1** that (1) neither  $s_s$  nor  $s_t$  belongs to  $p_{th}$ , or (2)  $\tau$  is a transition in  $p_{th}$  already. In this situation, no new path is to be generated.

The remaining three situations correspond to **Case 2**: both  $s_s$  and  $s_t$  belong to  $p_{th}$ , **Case 3**:  $s_t$  but not  $s_s$  belongs to  $p_{th}$ , and **Case 4**:  $s_s$  but not  $s_t$  belongs to

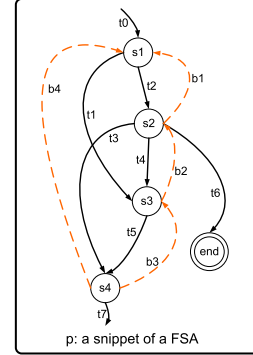


Fig. 4. A snippet of  $p$

$p_{th}$ . For page limitation, we detail **Case 2** in this paper and refer the interested reader to our technical report [16] for the description about **Case 3** and **4**.

Since  $SP_{Sib}^{bt}$  is simple and does not cover all possible situations, we use a more general FSA called  $p$ , and whose snippet is depicted in Fig. 4, to explain our computation steps. For **Case 2**, we detail different steps to compute new paths. After each step, we refer to our example to illustrate how this step applies.

**Case 2:** Both  $s_s$  and  $s_t$  belong to  $p_{th}$ . This situation is illustrated in Fig. 5 where  $pa_{th}$  corresponds to  $p_{th}$  and  $b2$  (see Fig. 5) corresponds to  $\tau$ .

*Step 1:* We initialize two state sets  $ST_{upp}$  and  $ST_{low}$  to  $\{s_t\}$  and  $\{s_s\}$  respectively.

In our example, these correspond to  $ST_{upp}^{b2} = \{s2\}$  and  $ST_{low}^{b2} = \{s3\}$ .

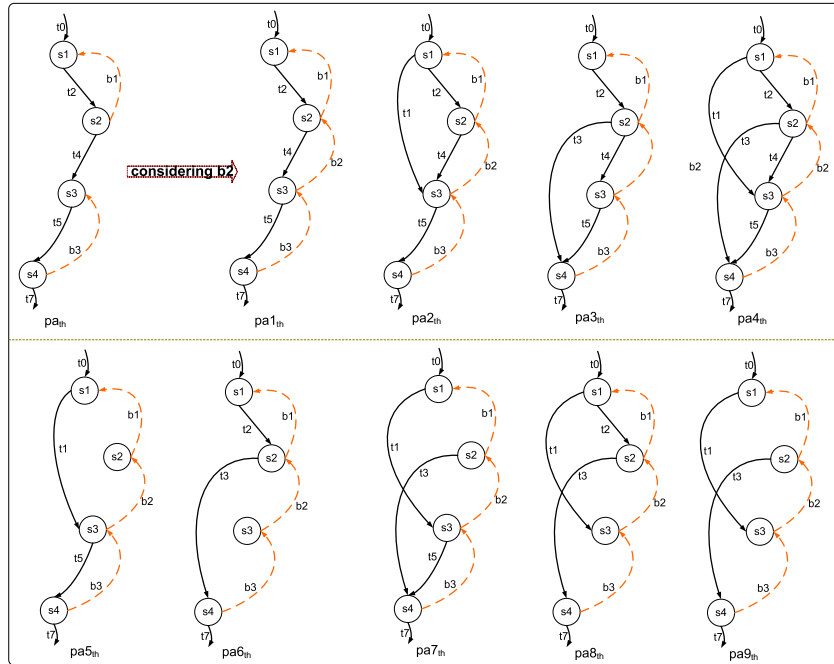
*Step 2:* We explore back-transitions in  $p_{th}$  to update  $ST_{upp}$  and  $ST_{low}$ . For each back-transition  $\tau_1$  in  $p_{th}$  (whose source state is  $s_s^1$  and whose target state is  $s_t^1$ ), if  $s_s^1 \in ST_{upp}$  then  $ST_{upp} = ST_{upp} \cup \{s_t^1\}$ , and if  $s_t^1 \in ST_{low}$  then  $ST_{low} = ST_{low} \cup \{s_s^1\}$ . This procedure stops when no back-transition can be explored anymore.

In our example, this step leads to  $ST_{upp}^{b2} = \{s1, s2\}$  and  $ST_{low}^{b2} = \{s3, s4\}$ .

*Step 3:* While ignoring back-transitions, we construct the set, denoted  $SEG$ , of all segments in  $p$  that start at one state in  $ST_{upp}$  and end at one state in  $ST_{low}$ .

In our example,  $SEG_{b2} = \{s2-t4-s3, s2-t4-s3-t5-s4, s2-t3-s4, s1-t1-s3, s1-t1-s3-t5-s4, s1-t2-s2-t4-s3, s1-t2-s2-t4-s3-t5-s4\}$ .

*Step 4:* We remove each segment  $seg \in SEG$  which all its states and transitions are contained in  $p_{th}$ . For  $SEG_{b2}$ , since some segments, such as  $s2-t4-s3$ , are



**Fig. 5.** Path computation for Case 2

already contained in  $pa_{th}$ , they are removed. This step leads to  $SEG_{b2} = \{s2-t3-s4, s1-t1-s3, s1-t1-s3-t5-s4\}$ .

*Step 5:* If two segments  $seg_1$  and  $seg_2$  in  $SEG$  are different merely because  $seg_2$  includes more states and transitions than  $seg_1$ , and these additional states and transitions are already in  $p_{th}$ ,  $seg_2$  is removed. The reason for this removal is that, if two new paths (denoted  $p_{th}^1$  and  $p_{th}^2$ ) are generated,  $p_{th}^1$  includes  $seg_1$ , and  $p_{th}^2$  includes  $seg_2$ , then  $p_{th}^2$  is actually a duplicate path with  $p_{th}^1$ .

Hence,  $SEG_{b2} = \{s2-t3-s4, s1-t1-s3\}$  since  $s1-t1-s3-t5-s4$  is excluded. We use  $m$  to denote the number of segments in  $SEG$ . In our case,  $m = |SEG_{b2}| = 2$ .

*Step 6:* Then,  $2^m$  new paths with  $i+1$  back-transitions are generated. Each new path is made through cloning  $p_{th}$ , adding  $\tau$ , and including either zero, or one, or multiple (even all) segments in  $SEG$ . The reason for this procedure is that, due to back-transitions in  $p_{th}$ , a new path with  $i+1$  back-transitions possibly loops back from one state in  $ST_{low}$  to one state in  $ST_{upp}$  for traversing some or even all segments in  $SEG$ . In our case,  $2^2 = 4$  new paths are generated as illustrated in Fig. 5, where  $pa1_{th}$  is the new path that no segment in  $SEG_{b2}$  is included,  $pa2_{th}$  and  $pa3_{th}$  show the cases that one segment in  $SEG_{b2}$  is considered, and  $pa4_{th}$  is the new path that includes both these two segments in  $SEG_{b2}$ .

*Step 7:* Each new path generated in Step 6 containing a segment  $seg$  in  $SEG$ , has another segment, alternative to  $seg$ , connecting the start state of  $seg$  to its final state. For instance, in  $pa2_{th}$ , besides the segment  $s1-t1-s3$  which belongs to  $SEG_{b2}$ , there is another segment  $s1-t2-s2-t4-s3$  starting at  $s1$  and ending at  $s3$ . For each path of this kind, we generate an additional new path by removing the segment alternative to those in  $SEG$ . For instance,  $pa5_{th}$  (see Fig. 5) is a new path generated from  $pa2_{th}$  by removing  $s1-t2-s2-t4-s3$ , which is an alternative segment to  $s1-t1-s3$  that belongs to  $SEG_{b2}$ .  $pa6_{th}$  is another new path generated from  $pa3_{th}$  following the same principal.

If a new path generated in Step 6 contains  $k$  segments in  $SEG$ ,  $2^k - 1$  additional new paths (some may be duplicate) are generated that correspond to all possible combinations of the  $k$  segments removal including the case where all of them are removed. As an example,  $pa4_{th}$  in Fig. 5 contains 2 segments in  $SEG_{b2}$ . then  $2^2 - 1 = 3$  additional new paths are generated, namely  $pa7_{th}$ ,  $pa8_{th}$  and  $pa9_{th}$ .  $pa7_{th}$  is generated by removing  $s1-t2-s2-t4-s3$ ,  $pa8_{th}$  by removing  $s2-t4-s3-t5-s4$ , and  $pa9_{th}$  by removing both  $s1-t2-s2-t4-s3$  and  $s2-t4-s3-t5-s4$ .

A new path is discarded if it is duplicate with another path generated previously. By iteratively applying the steps above, all paths are generated. As shown in Fig. 3, there are 7 paths for  $SP_{Sib}^{bt}$  in total. Since there are finite paths with  $i$  back-transitions, this procedure stops with  $k$  times recursion, and each checks all these  $k$  back-transitions with respect to all these last generated paths.

### 4.3 Path Computation of a Service Protocol

This section shows how we generate all instance sub-protocols of a service protocol based on the paths computed above. This procedure is achieved mainly by (1) unfolding transitions folded in the first step, and thereafter, by (2) taking self-loops and conditions into consideration that were abstracted away initially.

Given a transition  $\tau$  in a path  $p_{th}$ , and assume that  $\tau$  is folded from  $n$  transitions (i.e., they share the same source and target states),  $\tau$  is to be unfolded as follows: (1) if  $\tau$  is a part of a loop segment (including the case where  $\tau$  is a back transition), then  $\tau$  is unfolded to all possible combinations of these  $n$  transitions. Consequently,  $2^n - 1$  additional paths are generated, and each path is made through cloning  $p_{th}$  where  $\tau$  is replaced by a possible combination, otherwise, (2)  $\tau$  is unfolded to these  $n$  transitions. Thereafter,  $n$  additional paths are generated, and each path is made through cloning  $p_{th}$  where  $\tau$  is replaced by one of these  $n$  transitions.

A path that contains  $m$  folded transitions where each of them leads to  $t_j$  alternatives when unfolding it, is replaced by  $\prod_{j=1}^m t_j$  new paths that correspond to all possible combination of unfolding these  $m$  folded transitions. For instance, assume that  $p_{th}$  contains  $m=2$  folded transitions  $\tau_a$  and  $\tau_b$ .  $\tau_a$  is folded from two transitions  $\tau_1$  and  $\tau_2$ , and  $\tau_a$  is a part of a loop segment.  $\tau_b$  is folded from two transitions  $\tau_3$  and  $\tau_4$ , and  $\tau_b$  is not a part of a loop segment. Then, the alternatives to  $\tau_a$  are  $\tau_1$ , or  $\tau_2$ , or both  $\tau_1$  and  $\tau_2$ , i.e.,  $t_a = 3$ . The alternatives to  $\tau_b$  are  $\tau_3$  or  $\tau_4$ , i.e.,  $t_b = 2$ . Consequently,  $t_a \times t_b = 6$  new paths are generated for  $p_{th}$ ,  $\tau_a$  and  $\tau_b$ , where, as an example, one new path corresponds to the combination of (1) both  $\tau_1$  and  $\tau_2$  for  $\tau_a$ , and (2)  $\tau_3$  for  $\tau_b$ . In our case, there are 17 paths in total for  $SP_{Sib}$  after studying folded transitions in  $SP_{Sib}^{bt}$ .

After exploring folded transitions, we take self-loops into consideration. For a path  $p_{th}$  containing  $m$  self-loops,  $2^m - 1$  new paths are generated where all possible combinations of these self-loops are considered.  $p_{th}$  represents the situation that no self-loop is included. For instance, there are 34 paths in total for  $SP_{Sib}$  after studying self-loops.

Finally, we reattach conditions with associated transitions (that have been abstracted away in Step 1). Paths free of dependency conflicts are instance sub-protocols. In our case,  $SP_{Sib}$  has 34 instance sub-protocols and  $SP$  has 17.

## 5 Service Protocols Adaptability Assessment

### 5.1 Computing the Adaptation Degree and the Condition Set

For two service protocols  $p_1$  and  $p_2$ , and their complete adaptation graph  $adapt_{graph}$ , the adaptation degree is computed by means of Equation 1. The function  $instSubProtocol(p_1, adapt_{graph})$  counts instance sub-protocols in  $p_1$  that are captured by instance sub-protocol pairs of  $adapt_{graph}$ . The procedure of generating all instance sub-protocol pairs captured by  $adapt_{graph}$  is to be detailed in Section 5.2. If the parameter  $adapt_{graph}$  is set to *null*, the number of instance sub-protocols in  $p_1$  is returned. The procedure of generating all instance sub-protocols in  $p_1$  has been presented in Section 4.

$$adaptation(p_1, p_2) = \frac{instSubProtocol(p_1, adapt_{graph})}{instSubProtocol(p_1, null)} \quad (1)$$

For instance,  $adaptation(SP, SR) = 8/17 \approx 0.471$ , whereas  $adaptation(SR, SP) = 2/2 = 1$ . These show that the adaptability is an asymmetric relation

between service protocols. In our motivating example,  $adaptation(SR, SP\_A) = 1/2 = 0.5$ . These mean that the adaptation degree informs the requestor about different adaptation possibilities between candidate service providers.

We next re-explore the problem of distinguishing between partial and full adaptability (pending in Section 3) by means of the adaptation degree. If  $adaptation(p_1, p_2) = 1$ , then  $p_1$  is called fully adaptable with  $p_2$ , because each instance sub-protocol in  $p_1$  can perform a mediated service interaction with at least one instance sub-protocol in  $p_2$ . If  $adaptation(p_1, p_2) = 0$ ,  $p_1$  is assumed not adaptable with  $p_2$  since no instance sub-protocol in  $p_1$  can have an adaptable instance sub-protocol in  $p_2$ . Otherwise,  $p_1$  is regarded partially adaptable with  $p_2$ .

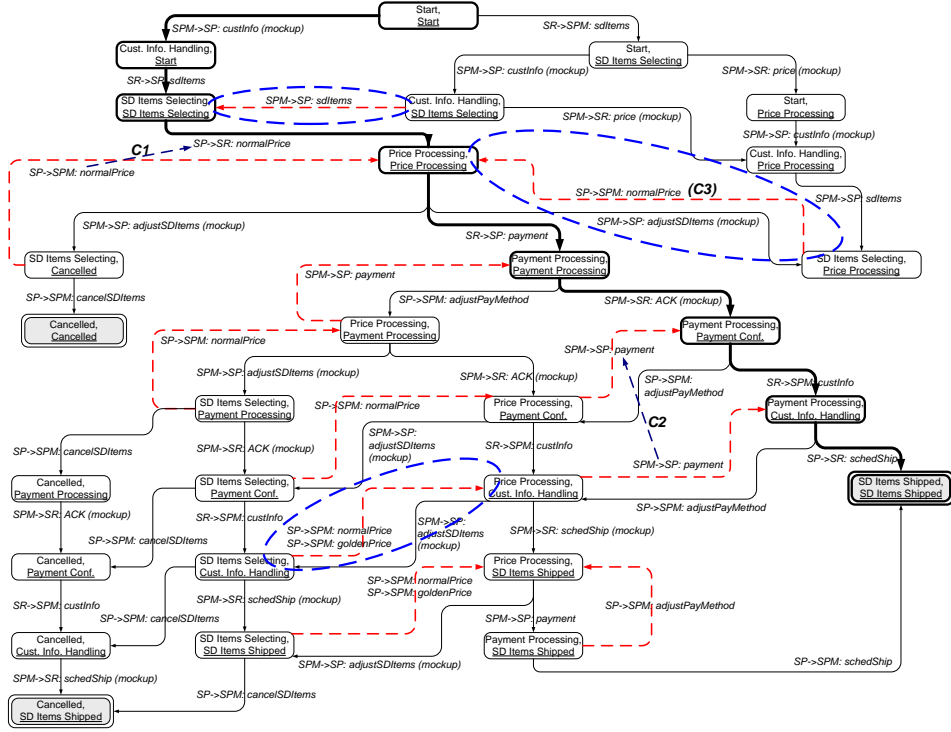
We recall that an adaptation graph reflects legal message exchanges between pairs of instance sub-protocols. For a particular pair of instance sub-protocols, the conjunction of conditions associated with their transitions constitutes another prerequisite for ensuring a proper adaptation. Such a *must-be-held* condition set is generated through studying all instance sub-protocol pairs. For instance for  $SP$  and  $SR$ , the condition set is expressed as follows:  $\{Cd_2 \wedge Cd_3 \wedge Cd_4, Cd_1 \wedge Cd_2 \wedge Cd_3 \wedge Cd_4, Cd_2 \wedge Cd_4, Cd_2 \wedge Cd_5, Cd_1 \wedge Cd_2 \wedge Cd_4\}$ . It is important to note that some conditions contain both  $Cd_1$  and  $Cd_2$ . This reflects the fact that, in some adaptation scenarios, when  $SP$  loops back to the state *SD Items Selecting*, the transition *custInfo(-)* in  $SR$  has been executed, and the message *custInfo* can make  $Cd_1$  satisfiable.

## 5.2 Instance Sub-Protocols Pairs in an Adaptation Graph

Based on the complete adaptation graph and all instance sub-protocols of a service protocol generated previously, in this section, we explore how to generate all pairs of instance sub-protocols that can conduct mediated service interactions between service protocols of a requestor and a provider. This procedure includes the following two sequential steps: we first generate all paths captured by this complete adaptation graph, and then, we project each path to a pair of instance sub-protocols and thereafter cluster these instance sub-protocol pairs.

**Path Computation for a Complete Adaptation Graph.** As the first step, we generate all paths in the complete adaptation graph using the technique presented in Section 4. For instance for  $adapt_{graph}$  as shown in Fig. 6, since it has 11 back-transitions and this graph is complex, thousands of paths are to be generated. Generally, computing all paths for a complete adaptation graph directly is usually inefficient.

Indeed, different from transitions in a service protocol where all transitions are necessary for achieving a particular business goal, some back-transitions in a complete adaptation graph may have no contribution to static analysis, and hence, can be ignored. Recall Definition 1, a transition in an adaptation graph specifies a legal message exchange either (1) between two service protocols (like  $SP \rightarrow SR: normalPrice$  in Fig. 6 that is between  $SP$  and  $SR$  through the SPM) or (2) between a service protocol and the SPM (like  $SP \rightarrow SPM: normalPrice$  in Fig. 6 that is between  $SP$  and  $SPM$ ). Given a back transition, if its message exchange has been covered by all traces from the initial state of the complete



**Fig. 6.**  $adapt_{graph}$ : the complete adaptation graph for  $SP$  and  $SR$  service protocols

adaptation graph to the source state of this back transition, which means that this back transition repeats a message exchange that has been performed before, this back transition can be ignored for static analysis purposes.

A back-transition  $\tau$  (the source state  $s_s$  and target state  $s_t$ ) in a complete adaptation graph can be ignored if the following two conditions are satisfied:

*Condition 1:* For any trace from the initial state of the complete adaptation graph to  $s_s$ , another (back) transition exists in this graph which (1) is *before*  $\tau$ , and (2) covers the message exchange by  $\tau$ . An example is the exchange for message  $normalPrice$  by the back transition  $SP \rightarrow SPM: normalPrice$  and by the transition  $SP \rightarrow SR: normalPrice$  as shown in Fig. 6 by a dashed line with a mark  $C1$ . The transition  $SP \rightarrow SR: normalPrice$  implicitly specifies the following two message exchanges through the  $SPM: SP \rightarrow SPM: normalPrice$  and  $SPM \rightarrow SR: normalPrice$ . Hence, the message exchange specified by the back transition  $SP \rightarrow SPM: normalPrice$  has been covered by that of the transition  $SP \rightarrow SR: normalPrice$ . Another example is two back transitions  $SPM \rightarrow SP: payment$  as shown in Fig. 6 by a dashed line with a mark  $C2$ . The back transition, which is the source of this dashed line, meets this condition.

*Condition 2:* Ignoring  $\tau$  does not cause that  $s_s$  is unreachable to any final state of this graph. A counterexample is the back transition  $SP \rightarrow SPM: nor-$

*malPrice* in Fig. 6 with a mark *C3*. If it is ignored, its source state (*SD Items Selecting, Price Processing*) is unreachable to any final state of this complete adaptation graph.

The first condition can be checked by a reversed breadth-first search on the graph starting at  $s_s$ , while the second condition can be verified through checking whether  $s_s$  has other outgoing transitions which are not back-transitions.

After examining all back-transitions in *adapt<sub>graph</sub>*, only three out of these eleven back-transitions, which are enclosed by means of dotted ellipses as shown in Fig. 6, are necessary. Hence, there are 432 paths to be computed in total.

**Instance Sub-Protocol Pairs Generation.** From the perspective of legal message exchange, each path in the complete adaptation graph leading from the initial state to one final state respects a mediated service interaction between a pair of instance sub-protocols. For instance, in Fig. 6, a path leading from the initial state (*Start, Start*) to one final state (*SD Items Shipped, SD Items Shipped*) (denoted  $p_{th}$ ) is marked by means of thick lines on the states and transitions. The instance sub-protocol of *SP* involving in this adaptation (denoted *ISP<sub>SP</sub>*) is  $path_1$  in Fig. 3 where the transition *price(-)* is to be unfolded to *normalPrice(-)*. The instance sub-protocol of *SR* involved is the one that leads to the final state *SD Items Shipped*.

On the other hand, for a certain instance sub-protocol pair, it possibly corresponds to more than one mediated service interaction since some messages can be exchanged in different orders. For instance, transitions *adjustPayMethod(+)* in *SP* and *custInfo(-)* in *SR* can be enabled in any order. Thereafter, multiple paths may reflect the mediated service interactions of the same instance sub-protocol pair. For instance, another path (denoted  $pd_{th}$ ) shares the same instance sub-protocol pair as that of  $p_{th}$ , if  $p_{th}$  and  $pd_{th}$  have a common segment as specified in  $p_{th}$  from the intermediate state (*SD Items Selecting, SD Items Selecting*) to the final state (*SD Items Shipped, SD Items Shipped*).

To make the instance sub-protocol pairs unique, we cluster paths if they are captured by a same pair of instance sub-protocols. This requires a technique to identify the pair of instance sub-protocols that a path reflects. We project a path to a service protocol. The result is a complete execution path [2] leading from the initial state to one final state. The projection is an operator [2] that identifies transitions in a path associated with a service protocol, and restores their polarity according to the service protocol specification. For instance, the transition *SP*  $\rightarrow$  *SR: normalPrice* as indicated in Fig. 6 by a dashed line with a mark *C1* is projected into one transition in *SP* (i.e., *normalPrice(-)*) and another transition in *SR* (i.e., *price(+)*), while *SP*  $\rightarrow$  *SPM: normalPrice* is projected into a transition in *SP* (i.e., *normalPrice(-)*).

We then identify which instance sub-protocol this projected path belongs. This is achieved by comparing the transition set in this projected path to that in an instance sub-protocol. For instance, after projecting  $p_{th}$  to *SP*, the transition set is the same as that of *ISP<sub>SP</sub>*. Hence, the instance sub-protocol pair is provided. We study other paths in such fashion. In our case, these 432 paths in *adapt<sub>graph</sub>* are clustered into 8 instance sub-protocol pairs of *SP* and *SR*.

## 6 Related Work and Conclusion

**Adaptability analysis.** A work similar to ours is the *adapter compatibility* analysis in the *Y-S* model [12] which checks whether or not two component protocols are adaptable with a particular adapter. The criteria are (1) no *unspecified receptions*, and (2) *deadlock free*. No *unspecified receptions* is restrictive, since message production and consumption in mediated service interactions are time-decoupled, and extra messages are often allowed. Our approach does not have this limitation. We give an adaptation degree which is more accurate than a binary answer. Since conditions in protocols are not explored, this work does not specify conditions that determine when two protocols are adaptable. Our approach specifies such necessary conditions. In addition, this work depends on the synchronous semantics. This assumption simplifies the problem, but it fails to capture most Web service interactions, since they are normally asynchronous. Our approach does not depend on such an assumption.

**Adapter construction.** Adapters are important for supporting interactions in the context of both software components [12] and Web services [3, 4, 6, 8, 13].

As shown in [12], an adapter is automatically constructed for two incompatible protocols. The adapter tackles order mismatches with *unspecified receptions*, but considers any deadlock as unresolvable. The same limitation exists in the adaptation mechanisms of mediation-aided service composition approaches [8].

In [3, 4], possible mismatches are categorized into several mismatch patterns, and adaptation templates [3] or composable adaptation operators [4] are proposed for handling these mismatch patterns. However, mismatches between two protocols are identified by a developer and an adapter is constructed manually.

Besides the mismatches covered by adapters above, [6] handles a deadlock through *evidences*. The choice of an *evidence* for resolving a given deadlock is decided by adapter developers, and hence, this method is not generic. This technique presumes that recommended business data is consistent with a certain interaction context, but data recommended by some *evidences*: such as *enumeration with default* and *log based value/type interface*, may not satisfy this assumption, since enumeration may not be the default value, and some business data may differ in different interactions. Whereas a *mock-up* message generated by the SPM [13] is consistent with a certain interaction context.

In short, adapter construction means that service protocols are adaptable in some case. The possibility and the conditions are not specified. Adapter building constitutes a starting point, but is inadequate, for assessing the adaptability. This paper builds upon the adapter construction presented in our previous work [13] and provides adaptability assessment, whose result is a key criterion to a requestor for identifying and selecting a suitable service provider from functionally-equivalent candidates according to her specific business requirements.

In conclusion, the two major contributions of this paper are as follows: first, our adaptability assessment, as reflected by the motivating example, is a comprehensive means of selecting the appropriate service provider among functionally-equivalent candidates. To the best of our knowledge, this assessment is new. Besides the technique presented in this paper, we have proposed another approach

in [15] that assesses the adaptability using protocol reduction and graph-search with backtracking techniques. Compared with [15], this paper brings the second major contribution of a novel path computation technique which computes all instance sub-protocols in a service protocol as well as their pairs in a complete adaptation graph. This computation is general and can be applied, besides to the adaptability assessment, also to the compatibility assessment where a numerical compatibility degree and a set of conditions are still missing.

## References

1. Backer, M.D., Snoeck, M., Monsieur, G., Lemahieu, W. and Dedene, G.: A scenario-based verification technique to assess the compatibility of collaborative business processes. *Data & Knowledge Engineering*. 68, 6, 531–551 (2009)
2. Benatallah, B., Casati, F. and Toumani, F.: Representing, analysing and managing web service protocols. *Data & Knowledge Engineering*. 58, 3, 327–357 (2006)
3. Benatallah, B., Casati, F., Grigori, D., Nezhad, H.R.M. and Toumani, F.: Developing Adapters for Web Services Integration. *Proc. of CAiSE*, 415–429 (2005)
4. Dumas, M., Spork, M. and Wang, W.: Adapt or perish: Algebra and visual notation for service interface adaptation. *Proc. of BPM*, 65–80 (2006)
5. Dumas, M., Benatallah, B. and Nezhad, H.R.M.: Web Service Protocols: Compatibility and Adaptation. *IEEE Data Engineering Bulletin*. 31, 3, 40–44 (2008)
6. Nezhad, H.R.M., Benatallah, B., Martens, A., Curbera, F. and Casati, F.: Semi-Automated Adaptation of Service Interactions. *Proc. of WWW*, 993–1002 (2007)
7. Nezhad, H.R.M., Saint-Paul, R., Benatallah, B. and Casati, F.: Deriving Protocol Models from Imperfect Service Conversation Logs. *IEEE Trans. on Knowledge and Data Engineering*. 20, 12, 1683–1698 (2008)
8. Tan, W., Fan, Y. and Zhou, M. : A Petri Net-Based Method for Compatibility Analysis and Composition of Web Services in Business Process Execution Language. *IEEE Trans. on Automation Science and Engineering*. 6, 1, 94-106 (2009)
9. Wu, Q., Pul, C., Sahai, A. and Barga, R.: Categorization and Optimization of Synchronization Dependencies in Business Processes. *Proc. of ICDE*, 306–315 (2007)
10. van der Aalst, W.M.P., Weijters, T. and Maruster, L.: Workflow Mining: Discovering Process Models from Event Logs. *IEEE Trans. on Knowledge and Data Engineering*. 16, 9, 1128–1142 (2004)
11. Ioannidis, Y., Ramakrishnan, R. and Winger, L.: Transitive closure algorithms based on graph traversal. *ACM Trans. on Database Systems*. 18, 3, 512-576 (1993)
12. Yellin, D.M. and Strom, R.E.: Protocol Specifications and Component Adaptors. *ACM Trans. on Programming Languages and Systems*. 19, 2, 292-333 (1997)
13. Zhou, Z., Bhiri, S., Gaaloul, W. and Hauswirth, M.: Developing Process Mediator for Supporting Mediated Service Interactions. *Proc. of ECOWS*, 155–164 (2008)
14. Zhou, Z., Bhiri, S. and Hauswirth, M.: Control and Data Dependencies in Business Processes Based on Semantic Business Activities. *Proc. of iiWAS*, 257–263 (2008)
15. Zhou, Z., Bhiri, S., Zhuge, H. and Hauswirth, M.: Assessing Service Protocols Adaptability Using Protocol Reduction and Graph-Search with Backtracking Techniques. *Proc. of SKG (To appear, 2009)*
16. Zhou, Z. and Bhiri, S.: Assessment of Service Protocols Adaptability. DERI technical report at [http://www.deri.ie/about/team/member/zhangbing\\_zhou/](http://www.deri.ie/about/team/member/zhangbing_zhou/) (2009)
17. Zhuge, H., Cheung, T.Y. and Pung, H.P.: A timed workflow process model. *Journal of Systems and Software* 55, 3, 231–243 (2001)