Assessment of Service Protocols Adaptability

Zhangbing Zhou and Sami Bhiri

DERI Technical Report 2009-03-15
March 2009
Assessment of Service Protocols Adaptability

Zhangbing Zhou\textsuperscript{1} and Sami Bhiri\textsuperscript{1}

Abstract. Protocol adaptation between services is a key functionality for ensuring successful interactions. Previous work has mainly interested in either compatibility analysis which targets at the direct service interaction, or constructing adapters for service protocols. In this paper, we are rather interested in characterizing whether two service protocols are adaptable without constructing an adapter, quantifying their adaptability degree, and identifying conditions providing which they can be properly adapted. We believe such an assessment is a key criteria for selecting the appropriate service among functionally-equivalent candidates. We firstly introduce a generic method that adapts service protocols without requiring to construct an adapter at design-time. Then we present a technique that enables to explore our interests mentioned above.

\textsuperscript{1}Digital Enterprise Research Institute (DERI), National University of Ireland, Galway, IDA Business Park, Lower Dangan, Galway, Ireland. E-mail: \{zhangbing.zhou, sami.bhiri\}@deri.org.

Acknowledgements: The work presented in this paper was supported (in part) by the EU funded TripCom Specific Targeted Research Project under Grant No. FP6-027324, (in part) by the EU funded SUPER Specific Targeted Research Project under Grant No. FP6-026850, (in part) by the Lion project supported by Science Foundation Ireland under Grant No. SFI/02/CE1/I131, and (in part) by the Lion-2 project supported by Science Foundation Ireland under Grant No. SFI/08/CE/I1380.

Copyright © 2009 by the authors
1 Introduction

Given the inherent autonomy, heterogeneity, and continuous evolution of Web services, mediated service interactions [24] are more likely to happen than direct ones. Therefore we believe that adaptability assessment is as important as compatibility checking which focuses on direct service interactions. Following [1] [8], by adaptation we mean the act of identifying, classifying, and reconciling mismatches between service behavioral interfaces which are called service protocols in this paper. By assessing the adaptability we mean characterizing whether two service protocols are adaptable without generating an adapter, computing the degree of adaptability, and perceiving conditions providing which they are able to be adapted. This assessment enables a client to select the most suitable candidate according to his/her requirements.

Most work done about service protocol analysis focuses on either compatibility analysis [2] [15] or adapter construction [1] [5] [8] [17] [23]. Adaptability can somehow be checked by approaches that construct adapters. However these approaches are ineligible for providing the level of assessment we target. Indeed, being able to build an adaptor means only that there are some cases where service protocols are possibly adaptable. However it does not differentiate between the degree of adaptability. Hence the least and the most adaptable protocols are all seen at the same level. Moreover these approaches do not explicitly specify conditions providing which two service protocols are adaptable. Service protocols may behave in such a way that they fall in non-adaptable situations.

In this paper, we assess the adaptability in service protocols. Besides a binary answer that decides whether two service protocols are adaptable, we compute the adaptability degree that corresponds to different capabilities of candidate services that a client service can interact with through an adapter, and provide conditions that specify the prerequisites for a successful adaptation. In general, the adaptability is assessed according to the following three sequential steps. For two service protocols:

1. First, we build a complete adaptation graph that captures all legal message exchanges between them through SPM. If this graph is not empty, which means that there exists some legal message exchanges between them, then they are adaptable. Otherwise they are not adaptable.

2. Then, for a service protocol, we generate a set of instance sub-protocols using path computation technique. An instance sub-protocol represents a valid part that may be executed for a particular instance of this service protocol. Valid means free of dependency confliction between transitions. The reason that we propose the notion of instance sub-protocol, rather than reuse existing similar concepts, is that a service protocol can have finite instance sub-protocols. This enables computing the adaptability degree and providing the condition set. Indeed each legal message exchange in an adaptation graph reflects an adaptation situation between a pair of instance sub-protocols of two service protocols. Then we compute all paths in an adaptation graph using path computation technique, and project each path to an instance sub-protocol pair. Consequently we get all instance sub-protocol pairs that can be properly adapted through SPM.

3. Finally, we compute the adaptability degree through scaling the instance sub-protocols captured by the adaptation graph with those in a service protocol, and generating the condition set where a condition is the conjunction of conditions associated with transitions in a certain instance sub-protocol pair.
Adaptability assessment depends tightly on the used adaptation mechanisms, i.e., what protocol mismatch patterns can be resolved. Two service protocols can be seen adaptable by one approach while they can be evaluated as unadaptable by another approach, simply because the first approach is able to resolve some existing mismatch patterns but the second cannot. In this paper we present an approach for assessing adaptability in accordance to our space-based process mediator (SPM for short) [25]. Our SPM is able to resolve all mismatch patterns identified by [1] [5]. The originality of our SPM is its generic way for adaptation where no adapter is required to be built at design-time. In addition our SPM is able to resolve a kind of deadlock considered unresolvable so far. On the other hand, our approach is generic and can be applied to other adapters as well.

It should be noted that path computation for a service protocol, and more generally for a directed cyclic graph, takes time exponential in the size of service protocol in the most general case. Hence, the adaptability degree and the condition set can be generated providing that service protocols to be studied are recommended not very large in size.

The main contributions this paper brings are as follows:

1. Firstly, to the best of our knowledge, this is the first adaptability assessment that provides the adaptability degree and identifies the condition set. In [7], the authors reviews a number of techniques of service protocol compatibility and adaptation. For the techniques related to adaptation, besides [23] where an approach is proposed for checking the existence of an adapter for incompatible protocols (i.e., a binary answer for adaptability between two component protocols), other techniques focus on constructing an adapter, and hence, on resolving signature and protocol mismatches. Following the research line, the technique proposed in this paper continues the adapter construction, and progresses on assessing the adaptability.

2. Secondly, the adaptability degree and the condition set resulting from our adaptability assessment are the key criteria to the client for choosing the most suitable service among functionally-equivalent candidates.

3. Finally, the context of this paper is the adaptability assessment. However, the techniques proposed in this paper, such as the path computation technique, the generation of the adaptability degree, and the identification of the condition set, is generic, and can be applied for other assessments like compatibility, replaceability [2], and the change support.

The outline of this paper is as follows. Section 2 motivates our work through an example that is used throughout the paper. Section 3 briefly introduces SPM. Section 4 introduces service protocol modeling and discusses the concepts of control and data dependencies. Section 5 generates a complete adaptation graph for two service protocols. Section 6 proposes a novel path computation technique to generate all instance sub-protocols for a service protocol, and details how service protocol assessment is conducted based on an adaptation graph. Section 7 presents our prototype. Section 8 reviews the related work and concludes.

2 Motivating Example

Fig. 1 depicts three service protocols and a potential interaction that a client, using a toy requestor service (TR), chooses from two toy shop services (TS and TS,A) to interact with for buying some

---

\(^1\)The symbol “+” means message incoming and “-” means message outgoing.
toys online. Service protocols are modeled through guarded finite state automata (GFSA for short) [9]. Guards denoted \( C_d_i \ (i \in [1,5]) \) correspond to conditions in BPEL specification, so that we do not differentiate guard and condition in this paper. We defend our choosing GFSA in Section 4. The only difference between TS and TS_A is that TS allows canceling an interaction if some selected toy items are out of stock.

\[
\begin{align*}
&\text{Cd}_1 \colon \text{a golden customer} \\
&\text{Cd}_2 \colon \text{not } \text{Cd}_1 \\
&\text{Cd}_3 \colon \text{(a golden customer)} \\
&\text{OR ("not a golden customer" AND "shipment is national")} \\
&\text{Cd}_4 \colon \text{price} \leq \$500 \\
&\text{Cd}_5 \colon \text{not } \text{Cd}_4
\end{align*}
\]

Figure 1: Service protocols for two toy shops (TS and TS_A) and one toy requestor (TR)

TS may apply a discount on the price depending on custInfo, so that custInfo is expected before the price is decided. Due to privacy concern, TR sends custInfo only if the client is convinced by price and is committed to buy. This example shows a deadlock where TS is requesting custInfo before sending price while TR is expecting price before deciding upon continuing and sending custInfo or not.

This kind of deadlock is resolvable according to our SPM adaptation mechanisms leveraging the dependencies that exist between transitions, but is beyond the capability of other adapters [1] [8] [17] [23]. Back to the example, custInfo is used for deciding whether a discount is to be applied or not. However a normalPrice is always applied by default. On the contrary, custInfo is mandatory for schedShip. Knowing that, SPM is able to send a mock-up custInfo message to TS which will then execute the following transition and provide a normalPrice. SPM will update custInfo for TS, in order to achieve schedShip, whenever custInfo is provided by TR. More detail about our SPM is given in Section 3.

The discussion above tells the client that both TS and TS_A are adaptable with TR if price is
acceptable. Then s/he wants to know whether the similar situation holds when price is unacceptable, so that s/he is able to cancel an interaction afterwards. Indeed, TS is able to support this situation where SPM generates mock-up custInfo and mock-up adjustToyItems messages, and then TS cancels the interaction. But TS_A cannot since the transition schedShip(-) cannot be enabled with a mock-up custInfo. Consequently the client perceives that TS is more adaptable with TR than TS_A, and is a more suitable candidate. However SPM is ineligible to detect such a difference since, in principle, an adapter is to recognize whether or not, but not how far, an adaptation is possible.

The reasoning above does not apply for other existing adapters since they cannot resolve the deadlock confronted. Hence TS and TS_A are both unadaptable with TR based on their adaptation mechanisms. But their limitation for assessing adaptability remains true if we consider a more general case.

In fact, the reasoning above explores legal message exchanges between TS and TR through our SPM. However, success of interactions in general and adaptation in particular depend on conditions that decide which path to follow and whether transitions are able to be enabled. In other words, only a legal message exchange cannot guarantee the success of an interaction or adaptation. For instance, for an adaptation between TS and TR leading to toy items been shipped, a prerequisite is that Cd2, Cd3 and Cd4 are able to be respected. To identify those conditions is beyond existing adaptation approaches.

In the following, we explore that, for two service protocols, how we can identify whether or not and how far they are adaptable, and the conditions they are to be respected for ensuring a successful adaptation.

3 Space-Based Process Mediator

This section briefly describes our SPM and its adaptation mechanisms. We start by introducing the principles that our SPM relies on. Then we present its architecture and explain how SPM implements these principles. We refer the interested reader to our previous work [25] for further details.

Like [1] [23], our SPM is built on the principle of time-decoupling between message production by one service protocol and consumption by other service protocols. This principle is supported by our SPM in a uniform way through a space-based architecture, rather than building an adaptor that encodes which message has to be stored and when to be resubmitted afterwards.

Our SPM relies also on the semantic description of service protocols and the kinds of dependency that may exist between their transitions (see section 4 for service protocol modeling) to resolve a kind of deadlock considered unresolvable so far. Such deadlock occurs if all service protocols expect concrete business data which cannot be inferred from previously exchanged ones. Our SPM can go around such kind of deadlock if the dependency that exist between one of the receiving transitions and one of its immediately following transitions is not mandatory. In that case, our SPM can produce a mock-up message which acts as the data expected by this receiving transition. This mock-up message is replaced by a concrete one whenever it is produced by a peer protocol.

Fig. 2 depicts the architecture of our SPM. A service protocol has a local space that saves its incoming and outgoing messages. Service protocols interact through SPM, rather than communicating in a direct manner. When a service protocol deposits a to-be-sent message into its local space, an event is generated and the Mediation Engine is notified. Based on the current state of the
adaptation, and using a set of communication and adaptation rules, the Mediation Engine either dispatches the corresponding message or drop it out. A local space is a shared space between the service protocol and the SPM.

The SPM has knowledge about the description of the interacting protocols, the schema of messages they have to exchange, and also has access to their state during the adaptation. Leveraging those information, the SPM can check at each state of the adaptation (1) if a protocol instance is expecting a certain message, (2) if it is sending a message, (3) if it is expecting an ACK for a sent message, or (4) if it is expecting a business data and if this data is mandatory or optional for the immediately following transitions. Based on these functionalities our SPM applies a set of adaptation mechanisms, in a uniform way, that are able to resolve all mismatch patterns identified in [1] [5].

4 Service Protocol, Control and Data Dependencies

4.1 Service Protocol Model

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, as well as a state automata, is sequentially executed. In other words, a state in a service protocol specification allows multiple transitions to follow and hence leads to multiple states. However for a certain point at runtime, only one transition can be enabled and then fired and consequently this state is to be evolved to one of the following states.

As presented in [2], different message and condition pairs are always possible to be mapped onto different message labels, and hence conditions are abstracted away. However this simplification loses important information of conditions. For instance in TS, the price is mapped onto goldenPrice and normalPrice. If Cd_1 and Cd_2 are not specified, the rationale for choosing goldenPrice or normalPrice is lost. Indeed, conditions are fundamental knowledge for getting known service protocols themselves as well as ensuring possible interactions between partners. Consequently we model service protocols in terms of GFSA. During a run of a service protocol, a transition is taken providing that the associated condition is evaluated to be satisfiable.

Definition 1 (Service Protocol) A service protocol \( p \) is a tuple \((M, S, s, F, C, T)\) where \( M \) is a finite set of messages. Following [2], for each message \( m \in M \), we define a function \( \text{Polarity}(p, m) \) which will be “+” if incoming or “−” if outgoing. \( S \) 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 $\tau = (s_s, s_t, m, c \mid \text{true}) \in T$ 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 by default if no condition associates with $\tau$.

In our approach, transitions are semantically described by specifying their input, output, precondition, and effect. For instance, BPEL4SWS [12] proposed by Frank Leymann’s team enables describing activity implementations semantically by means of Semantic Web services (i.e. OWL-S or WSMO), and BPEL\textsuperscript{light} [19] is able to support the execution of BPEL4SWS processes. The input and output define consumed and produced messages respectively, while the precondition and effect represent the state of world before and after the execution of a transition.

4.2 Control and Data Dependencies

As presented by [22], different kinds of dependencies, such as control, data, service and cooperation dependencies, exist between transitions. These dependencies are often obfuscated by a service protocol that defines all possible execution sequencing of transitions. A sequencing constraint in a service protocol may result from one or multiple kinds of dependencies [22]. In our approach, we consider two kinds of dependencies namely control and data dependencies.

A transition $B$ is control dependent on another transition $A$ if the completion of $A$ (marked by its effect) is a necessary condition for the enablement of $B$ (guarded by its precondition). Data dependencies are classified as mandatory or optional. $B$ is mandatorily data dependent on $A$ if common data exist in the output of $A$ and the input of $B$. Whereas $B$ is optionally data dependent on $A$ if no common data exist in the output of $A$ and the input of $B$, but incoming conditions of $B$ uses the data in the output of $A$.

In principle, a transition in a service protocol is allowed to be control and/or data dependent on any other transition. But in practice, service protocols cannot be modeled at will because a dependency transition is not able to be enabled at runtime if its precondition cannot be satisfied (i.e. control dependent transitions have not been completed) and/or its required input is unavailable (i.e. source transitions for required data have not been completed). In general, any two transitions are not allowed to be dependent on each other, and one transition cannot depend on another transition that is possibly to be executed afterwards. However, the correctness of service protocol modeling is out of the focus of this paper, and we require that service protocols are properly modeled. We refer the interested reader to a promising work [6] which relies on the IOPE (Input, Output, Preconditions, and Effects) paradigm to specify operation semantics, and then, is able to verify various semantic properties of a e-service.

[22] claims that different kinds of dependencies between transitions can be extracted from design documents. [16] [13] propose how to extract data dependencies from BPEL process models (also referred to data links in [13]). In our approach we extract control and data dependencies from the semantic description of transitions [26].

Hence, we distinguish control and data dependency graphs of a service protocol that specify a finite set of asymmetric, irreflexive and transitive relations among transitions. Fig. 3 depicts control and data dependency graphs of $TS$. 
4.3 Instance Sub-Protocol

Before moving into the next section, we need to define a concept of 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 confliction. For instance, cancelToyItems(-) in TS always coexists with toyItems(+). Indeed, an instance sub-protocol itself is a service protocol, where all states, transitions and conditions contained may be visited in a particular execution. For instance, TS_A is an instance sub-protocol of itself as well as of TS, but TS itself is not an instance sub-protocol since no execution can lead to two final states. We explain how to generate all instance sub-protocols for a service protocol in Section 6.

The reason that we propose the notion of instance sub-protocol, rather than reuse existing concepts of complete execution path/tree proposed in [2], lies in that, there are finite instance sub-protocols for any service protocol. However for a service protocol that contains even one loop, complete execution paths/trees are infinite. Indeed, an instance sub-protocol corresponds to a category of complete execution paths/trees where they are different only because of the repetition of loop segments. This observation enables us computing the degree of adaptability, rather than providing a binary or ternary answer.

Indeed, our instance sub-protocol is similar to the notion of instance subgraph proposed in [21], which represents a subset of workflow tasks to be executed in a particular workflow instance, where a workflow is modeled by means of a directed acyclic graph (DAG for short). Hence instance subgraphs are independent on each other, in other words, assuming TSK_1 and TSK_2 are sets of workflow tasks of two different instance subgraphs, then the relation of TSK_1 ⊂ TSK_2 or TSK_2 ⊂ TSK_1 is impossible. In our approach, a part of an instance sub-protocol of a service protocol p may be also an instance sub-protocol of p. [21] has calculated the number of instance subgraphs in a workflow graph. In fact, to calculate the number of subgraphs in a DAG between a pair of vertices takes time polynomial in the size of DAG in the most general case. However this procedure in the context of a direct cyclic graph takes time exponential in general. These observations promote us for proposing the concept of instance sub-protocol, rather than reusing the notion of instance subgraph.
5 Service Protocols Adaptation Graph

This section addresses a joint analysis of two service protocols, that of a requestor (like TR) and a provider (like TS), to see whether they are able to be adapted by means of our SPM. Intuitively, we need to examine all pair of instance sub-protocols in these two service protocols for studying whether they are adaptable. However a service protocol may have many instance sub-protocols if it contains loop segments (for example TS has seventeen instance sub-protocols but TR has two), and some instance sub-protocols for a service protocol may be much similar, hence it is not efficient to examine following this strategy.

In this paper, we introduce a notion of adaptation graph that captures possible adaptation traces between two service protocols, where an adaptation trace captures a legal message exchange by means of SPM between a pair of instance sub-protocols of these two service protocols.

5.1 Definition of an Adaptation Graph

We define an adaptation graph using GFSA where a state is a combination of two states of participating service protocols, and a transition is either a message exchange between these two service protocols through SPM, or a service protocol sends a message to SPM, or SPM forwards a message (a concrete message from the partner, or a mock-up message generated by SPM) to a service protocol. Conditions associated with a transition in an adaptation graph are inherited from those of relevant transitions in participating service protocols. However conditions are not to be evaluated during the adaptation graph building phase since the evaluation of conditions relies on runtimely exchanged message instances.

Definition 2 (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 between \( p_1 \) and \( p_2 \) by means of SPM is a tuple \( (M, S, s, F, C, T) \) where:

- \( M \subseteq M_1 \cup M_2 \cup M_{SPM} \), where \( M_{SPM} \) is a finite set of mock-up messages generated by SPM. The polarity of messages are defined as follows: messages outgoing in \( M_1 \) and \( M_2 \) is sent to SPM or the partner, messages incoming in \( M_1 \) and \( M_2 \) is sent by SPM or the partner, and messages in \( M_{SPM} \) is sent to either \( p_1 \) 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 characterizes all possible adaptation traces. For instance, except the segments marked by dashed lines, Fig. 4 shows a complete adaptation graph for TS and TR by means of SPM.

5.2 Generating a Complete Adaptation Graph

In the following we propose an operator \( genAdaptationGraph() \) by Algorithm 1 that generates a complete adaptation graph for two service protocols. \( p_1 = (M_1, S_1, s_1, F_1, C_1, T_1) \) and \( p_2 = (M_2, S_2, s_2, F_2, C_2, T_2) \) are two input service protocols, and \( adaptGraph = (M, S, s, F, C, T) \) represents a complete adaptation graph generated for \( p_1 \) and \( p_2 \).
Figure 4: An adaptation graph for the toy shop TS and the toy requestor TR

In general, it traverses from the initial states of the input service protocols (line 2) to their final states (line 28), and constructs the intermediate states through combining the intermediate states of the input service protocols providing that there are legal message exchanges by means of SPM leading to them (line 3-27). The procedure clean(adapt_graph) used at line 28, not detailed in the algorithm, removes from adapt_graph all the states, as well as related messages, transitions and conditions, that cannot lead to the final states, for instance the segments in Fig. 4 that are marked as dashed lines. The remaining graph forms a DFSA since line 9 and 20 guarantee that each transition relates to one source state and one target state.

The set Cand initially contains the initial states of p₁ and p₂ with empty message sets since no message exchange has been conducted (line 2). For each candidate \((s_{p1}, M_{p1}, s_{p2}, M_{p2})\) in Cand,
Algorithm 1: genAdaptationGraph

Variable: Cand: a finite set of quadruples in terms of \((s_{p1}, M_{p1}, s_{p2}, M_{p2})\) that is composed of a state \(s_{p1}\) (or \(s_{p2}\)) for \(p_1\) (or \(p_2\)), and a message set \(M_{p1}\) (or \(M_{p2}\)) that \(p_1\) (or \(p_2\)) sends and receives when \(p_1\) (or \(p_2\)) evolves to \(s_{p1}\) (or \(s_{p2}\)) by means of SPM.

Function: getEnabledT(p, s, M): to get a set of transitions in a service protocol \(p\) in terms of \((s, s^t, m, c)\) so that (1) \(s\) is the source state, and (2) each \(\tau = (s, s^t, m, c)\) is able to be enabled according to \(M\), in other words, if \(\tau\) is control or mandatorily data dependent on \(\tau_1 = (s_1, s_1^t, m_1, c_1)\), then \(m_1 \in M\), and if \(\tau\) is optionally data dependent on \(\tau_2 = (s_2, s_2^t, m_2, c_2)\), then a concrete or a mock-up \(m_2\) is in \(M\),

1 begin
2 \(M, F, C, T \leftarrow \emptyset; s \leftarrow (s_1, s_2); S \leftarrow \{s\}; \text{Cand} \leftarrow \{(s_1, \emptyset, s_2, \emptyset)\}\)
3 foreach \((s_{p1}, M_{p1}, s_{p2}, M_{p2}) \in \text{Cand}\) do
4 \(T_{s_{p1}} \leftarrow \text{getEnabledT}(p_1, s_{p1}, M_{p1}); T_{s_{p2}} \leftarrow \text{getEnabledT}(p_2, s_{p2}, M_{p2})\)
5 foreach \(m_i\) such that \(\exists (s_{p1}, s_{p1}^t, m_1, c_1) \in T_{s_{p1}}\) and \(\exists (s_{p2}, s_{p2}^t, m_i, c_2) \in T_{s_{p2}}\) and
6 \(\text{Polarity}(p_{1}, m_i) \neq \text{Polarity}(p_2, m_i)\) do
7 \(m \leftarrow p_2 \rightarrow p_1 : m_i\)
8 if \(\text{Polarity}(p_1, m_i) = "-"\) then
9 \(m \leftarrow p_1 \rightarrow p_2 : m_i\)
10 if \((s_{p1}, s_{p2}), (s_{p1}^t, s_{p2}^t), m, c_{p1} \land c_{p2}\) \(\notin T\) then
11 \(S \leftarrow S \cup \{(s_{p1}, s_{p2})\}\)
12 \(T \leftarrow T \cup \{(s_{p1}, s_{p2}), (s_{p1}^t, s_{p2}^t), m, c_{p1} \land c_{p2}\}\)
13 \(M \leftarrow M \cup \{m\}; C \leftarrow C \cup \{c_{p1} \land c_{p2}\}\)
14 \(\text{Cand} \leftarrow \text{Cand} \cup \{(s_{p1}, M_{p1} \cup \{m\}, s_{p2}, M_{p2} \cup \{m\})\}\)
15 \(T_{s_{p1}} \leftarrow T_{s_{p1}} \setminus \{(s_{p1}, s_{p1}^t, m_i, c_1)\}; T_{s_{p2}} \leftarrow T_{s_{p2}} \setminus \{(s_{p2}, s_{p2}^t, m_i, c_2)\}\)
16 foreach \((s_{p1}, s_{p1}^t, m_1, c_1) \in T_{s_{p1}}\) do
17 \(m \leftarrow \text{SPM} \rightarrow p_1 : m_1\) (mockup); \(M_{p2}^{*p1} \leftarrow M_{p2}\)
18 if \(\text{Polarity}(p_1, m_1) = "-"\) then
19 \(m \leftarrow p_1 \rightarrow \text{SPM} : m_1; M_{p2}^{*p1} \leftarrow M_{p2} \cup \{m_1\}\)
20 if \((s_{p1}, s_{p2}), (s_{p1}^t, s_{p2}^t), m, c_{p1}\) \(\notin T\) then
21 \(S \leftarrow S \cup \{(s_{p1}, s_{p2})\}\)
22 \(T \leftarrow T \cup \{(s_{p1}, s_{p2}), (s_{p1}^t, s_{p2}^t), m, c_{p1}\}\)
23 \(M \leftarrow M \cup \{m\}; C \leftarrow C \cup \{c_{p1}\}\)
24 \(\text{Cand} \leftarrow \text{Cand} \cup \{(s_{p1}, M_{p1} \cup \{m_1\}, s_{p2}, M_{p2}^{*p1})\}\)
25 Repeat the procedure from line 16 to line 25 for each transition in \(T_{s_{p2}}\)
26 \(\text{Cand} \leftarrow \text{Cand} \setminus \{(s_{p1}, M_{p1}, s_{p2}, M_{p2})\}\)
27 adapt_graph \(\leftarrow \text{clean}(\text{adapt_graph}); F \leftarrow F_1 \times F_2 \land S\)
28 end
we retrieve all transitions in \( p_1 \) (or \( p_2 \)) that \( s_{p1} \) (or \( s_{p2} \)) is the source state and each transition can be enabled (see the description of the function \( getEnabledT(p, s, M) \)) leveraging control and data dependency graphs (line 4). Indeed to check whether a transition \( \tau \) is able to be enabled is a heavy operation since it needs to visit control and data dependency graphs for finding the transitions \( T_{dep} \) that \( \tau \) depends on, and then to check whether the messages operated by \( T_{dep} \) are consistent with \( M_{p1} \) (or \( M_{p2} \)). To lighten this operation, two techniques are introduced. Firstly we apply a tag on each transition \( \tau \) in \( T_1 \) and \( T_2 \) for specifying whether it is enabled or not. This tag is initially set to \textit{not}, and is set to and kept as \textit{yes} once \( \tau \) is enabled. The reason is that, if \( \tau \) can be enabled at a state \( s_1 \), \( \tau \) will be always enabled at the following states (such as \( s_2 \)) since the evolution from \( s_1 \) to \( s_2 \) will generate new messages and/or update mock-up messages using concrete ones. Consequently for \( \tau \), once enabled, this checking is unnecessary anymore. Secondly, as presented in [26], control and data dependency graphs are merged and then optimized into a minimal dependency graph. Hence to find the transitions that \( \tau \) depends on is related to this minimal dependency graph only. However the time complexity of \( getEnabledT(p, s, M) \) in the worst case is \( O(m) \) where \( m \) is the number of transitions in \( p \), since all transitions in \( p \) except \( \tau \) are possibly being visited.

We firstly deal with paired transitions in \( T_{sp1} \) and \( T_{sp2} \) that operate same messages but with different polarities (line 5-15). In this case, these messages are exchanged from \( p_1 \) to \( p_2 \) or \textit{vice versa} through SPM (line 6-8). If such a pair of transitions have not been handled (line 9), they as well as related resources are incorporated into adapt\(_{graph} \) (line 10-13). The target states as well as updated message sets are an candidate for the future exploration (line 14).

Then other transitions in \( T_{sp1} \) and \( T_{sp2} \) are studied afterwards (line 16-26). We explain the handling of \( T_{sp1} \) as the example. For an incoming transition in \( T_{sp1} \), SPM generates a mock-up message (line 17). Otherwise \( p_1 \) sends this message \( m_1 \) to SPM (line 18-19). The message set of \( s_{p2} \) is also updated by incorporating \( m_1 \) since \( m_1 \) may be used either for updating mock-up \( m_1 \) in \( p_2 \) or for possible future consumption (line 19). If such a transition \( \tau \) has not been handled (line 20), \( \tau \) with related resources are incorporated into adapt\(_{graph} \) (21-24). The target states as well as updated message sets are an candidate for the future exploration (line 25).

Thereafter this candidate is removed from the candidate list (line 27). Finally \textit{clean()} is called and final states of adapt\(_{graph} \) are derived from \( S \) (line 28).

During the handling of transitions (line 9-13 and 20-24), the target state pair may be already in \( S \). There are two possible reasons: the target state pair has been explored by other source state pairs, or multiple transitions are specified in \( p_1 \) and/or \( p_2 \) that connect the source state and the target state. For instance, for two transitions \textit{glodenPrice(-)} and \textit{normalPrice(-)} in \( TS \), they share the source and target state pair as \textit{Toy Items Selecting} and \textit{Price Processing}.

Possibly some state pairs in \( p_1 \) and \( p_2 \) are to be explored multiple times since new candidate generation (line 14 or 25) is not restricted by the If segment (line 10-11 or 21-22). The reason lies in that \( getEnabledT(p, s, M) \) returns currently enabled transitions according to \( M \). Hence when \( M \) is updated, possibly some transitions, which cannot be enabled previously, can be enabled at this moment. For instance, if there is a transition \( \tau \) in \( TR \) with \textit{Cust. Info. Handling} as the source state and \textit{Payment Processing} as the target state. When the state pair (\textit{Payment Processing}, \textit{Payment Processing}) is evolved from (\textit{Price Processing}, \textit{Price Processing}), the transition \textit{schedShip(-)} in \( TS \) cannot be enabled using a mock-up \textit{custInfo}. However when (\textit{Payment Processing}, \textit{Payment Processing}) is evolved from (\textit{Payment Processing}, \textit{Cust. Info. Handling}) through \( \tau \), the transition \textit{schedShip(-)} in \( TS \) can be enabled since \textit{custInfo} has been provided by the transition \textit{custInfo(-)} in \( TR \).
5.3 Correctness Proof

The following three properties prove the correctness of Algorithm 1.

- **Termination.** Algorithm 1 iteratively handles candidates in \(C\). Consequently the termination of Algorithm 1 leverages two facets: the termination of each candidate handling, and finite number of candidates in \(C\).

  Firstly the proceeding of a candidate is bounded by finite transitions in \(p_1\) and \(p_2\) that are retrieved using \(getEnabledT(p, s, M)\) (line 4). Hence the procedure of line 5-27 terminates when these transitions are dealt with.

  Secondly candidates in \(C\) is finite. The reason is that new candidate generation (line 14 and 25) depends on new transitions for \(adapt_{graph}\) (line 12 and 23), which depend on transitions in \(p_1\) and \(p_2\). Since transitions in \(p_1\) and \(p_2\) are finite, new transitions for \(adapt_{graph}\) are finite, and then candidates for \(C\) are finite.

  A consequence of the reasoning above is that Algorithm 1 can terminate.

- **Soundness.** Algorithm 1 is sound in the sense that any path in \(adapt_{graph}\) leading from the initial state to one of the final states reflects a legal message change between a pair of instance sub-protocols of two service protocols \(p_1\) and \(p_2\) by means of SPM. Line 12 and 23 in Algorithm 1 specify that such a path is composed of alternating labels from states and transitions in \(adapt_{graph}\), and a transition is either a message exchange between \(p_1\) and \(p_2\), or \(p_1\) (or \(p_2\)) sends a message to SPM, or SPM generates a mock-up message and forwards it to \(p_1\) (or \(p_2\)).

  We prove by contradiction that there is a path in \(adapt_{graph}\) which does not reflects a legal message exchange between a pair of instance sub-protocols in \(p_1\) and \(p_2\). Then at least one transition \(\tau = ((s_{p_1}, s_{p_2}), (s'_{p_1}, s'_{p_2}), m, c)\) in this path is invalid. However this argument is impossible because:

  1. if \(s_{p_1}\) and \(s'_{p_1}\) are different states in \(p_1\), and \(s_{p_2}\) and \(s'_{p_2}\) are different states in \(p_2\), then a transition \(\tau_1 = (s_{p_1}, s'_{p_1}, m_1, c_1)\) exists in \(T_1\), another transition \(\tau_2 = (s_{p_2}, s'_{p_2}, m_2, c_2)\) exists in \(T_2\), and the function \(getEnabledT(p, s, M)\) ensures that both \(\tau_1\) and \(\tau_2\) can be enabled according to messages exchanged previously. In addition \(m_1 = m_2\) but with different polarity, otherwise line 5 in Algorithm 1 cannot be satisfied and thus \(\tau\) cannot be generated. Therefore \(\tau\) cannot be a transition that refers to a message exchange between \(p_1\) and \(p_2\).

  2. Assume \(\tau\) reflects that a service protocol like \(p_1\) sends a message to SPM. Then \(s_{p_2}\) and \(s'_{p_2}\) refer to a same state in \(p_2\), and a transition \(\tau_1 = (s_{p_1}, s'_{p_1}, m_1, c_1)\) exists in \(T_1\) where \(Polarity(p_1, m_1) = "-"\). Then a natural consequence is that \(\tau\) is valid. Therefore \(\tau\) cannot be a transition that refers to a situation that \(p_1\) (or \(p_2\)) sends a message to SPM.

  3. Similar to the above, \(\tau\) cannot be a transition that refers to a situation that SPM generates a mock-up message and forwards it to \(p_1\) (or \(p_2\)).

The reasoning above shows that \(\tau\) cannot be a possible transition in \(adapt_{graph}\), in other words, such a \(\tau\) does not exist. Hence Algorithm 1 is sound.
In the worst case, the time complexity of Algorithm 1 is $O(p_1 + p_2)$ possible number of candidates to be examined and the complexity of each candidate handling.

Transitions for a pair of source and target states, and

$$\text{The complexity of handling one (pair of) transition(s) depends on line 9 and 20, where there are } O(kn)$$

$O(kn)$ candidates are possibly to be generated, and hence there are $2kn^2(n - 2)$ candidates in maximum to be examined. In the following we consider the time complexity for each candidate handling. The time complexity for line 4 is $O(kn^2)$ as discussed previously. There are at most $k(n - 1)$ transitions in $T_{sp1}$ and $T_{sp2}$ for each candidate. The complexity of handling one (pair of) transition(s) depends on line 9 and 20, where there are at most $2k(n - 1)(n - 2)$ transitions in $T$. Hence the time complexity of handling a candidate is $O(kn^2)$. Consequently the overall time complexity in the worst case is $O(k^3n^6)$. It is worth noting that, since the function clean() in line 28 can be reduced to a graph reachability problem, it can be implemented efficiently [11]. The worst case time complexity is a little high. However it is time polynomial and hence Algorithm 1 is tractable. In addition, as presented in [18], service protocols in general are designed by humans, and hence, they tend to be fairly simple models.

On the other hand, a service protocol is usually not so complex as what we assumed above. We next explore the time complexity in the best case. Considering a service protocol such that: (1) this service protocol has no loop segments, and (2) for any two neighboring transitions $\tau_1 = (s_1, s_2, m_1, c_1)$ and $\tau_2 = (s_2, s_3, m_2, c_2)$, $\tau_2$ is either control or mandatorily data dependent on $\tau_1$, then the time complexity is $O(k^3n^2)$. The reason is that, there are at most $4kn$ candidates to be explored, since any state in a service protocol can be paired with two neighboring states in another service protocol, each state pair can be explored only once and can have at most $2k$ kinds of message combination. For the handling of each candidate, the time complexity is $O(kn)$. The reason is as follows. The time complexity for line 4 is $O(kn)$, and the number of transitions in $T_{sp1}$ and $T_{sp2}$ is at most $k$. For each transition in $T_{sp1}$ and $T_{sp2}$, the time complexity depends on line 9 and 20, so that, is $O(kn)$ since there are at most $2k(n - 1)$ transitions in one service protocol and at most $4k(n - 1)$ transitions in an adaptation graph.

• Completeness. The completeness of Algorithm 1 means that the paths in adapt_graph captures all legal message exchanges by means of SPM between pairs of instance sub-protocols in $p_1$ and $p_2$.

The reasoning of Termination tells us that the complexity of Algorithm 1 depends on the possible number of candidates to be examined and the complexity of each candidate handling.

5.4 Time Complexity of Algorithm genAdaptationGraph

In the worst case, the time complexity of Algorithm 1 is $O(k^3n^6)$ where $k$ is the upper bound of transitions for a pair of source and target states, and $n$ is the maximum number of states in $p_1$ and $p_2$. Since in the worst case, $p_1$ and $p_2$ are strongly connected, in other words, for any two states $s_1$ and $s_2$ in a service protocol except the initial and final states, there are $k$ transitions connecting $s_1$ to $s_2$ and vice versa. Then there are $k(n - 1)(n - 2)$ transitions in each service protocol, and $n^2$ state pairs to be considered. For each state pair, $2k(n - 2)$ candidates are possibly to be generated, and hence there are $2kn^2(n - 2)$ candidates in maximum to be examined. In the following we consider the time complexity for each candidate handling. The time complexity for line 4 is $O(kn^2)$ as discussed previously. There are at most $k(n - 1)$ transitions in $T_{sp1}$ and $T_{sp2}$ for each candidate. The complexity of handling one (pair of) transition(s) depends on line 9 and 20, where there are at most $2k(n - 1)(n - 2)$ transitions in $T$. Hence the time complexity of handling a candidate is $O(kn^2)$. Consequently the overall time complexity in the worst case is $O(k^3n^6)$. It is worth noting that, since the function clean() in line 28 can be reduced to a graph reachability problem, it can be implemented efficiently [11]. The worst case time complexity is a little high. However it is time polynomial and hence Algorithm 1 is tractable. In addition, as presented in [18], service protocols in general are designed by humans, and hence, they tend to be fairly simple models.

On the other hand, a service protocol is usually not so complex as what we assumed above. We next explore the time complexity in the best case. Considering a service protocol such that: (1) this service protocol has no loop segments, and (2) for any two neighboring transitions $\tau_1 = (s_1, s_2, m_1, c_1)$ and $\tau_2 = (s_2, s_3, m_2, c_2)$, $\tau_2$ is either control or mandatorily data dependent on $\tau_1$, then the time complexity is $O(k^3n^2)$. The reason is that, there are at most $4kn$ candidates to be explored, since any state in a service protocol can be paired with two neighboring states in another service protocol, each state pair can be explored only once and can have at most $2k$ kinds of message combination. For the handling of each candidate, the time complexity is $O(kn)$. The reason is as follows. The time complexity for line 4 is $O(kn)$, and the number of transitions in $T_{sp1}$ and $T_{sp2}$ is at most $k$. For each transition in $T_{sp1}$ and $T_{sp2}$, the time complexity depends on line 9 and 20, so that, is $O(kn)$ since there are at most $2k(n - 1)$ transitions in one service protocol and at most $4k(n - 1)$ transitions in an adaptation graph.
5.5 An Adaptability Graph and the Adaptability Analysis

Next we explore how a complete adaptability graph contributes to the adaptability analysis. For two service protocols $p_1$ and $p_2$, partial adaptability specifies the situation that some instance sub-protocols in $p_1$ can be adapted properly with those in $p_2$ by means of SPM, while full adaptability requires that all instance sub-protocols in $p_1$ can be adapted properly with those in $p_2$. Indeed adaptability is an asymmetric relation between service protocols.

If $adapt_{graph}$ is not empty, which means that at least one pair of instance sub-protocols in $p_1$ and $p_2$ can be adapted properly by means of SPM, hence $p_1$ and $p_2$ are adaptable. However to differentiate between partial and full adaptability requires checking whether all instance sub-protocols are reflected by the paths in $adapt_{graph}$. Indeed, to generate all instance sub-protocols for a service protocol is not a trivial task which will be addressed in the next section. Consequently we conclude this section by providing a black-white answer for adaptability.

6 Service Protocols Adaptability Assessment

Leveraging the complete adaptation graph $adapt_{graph}$ generated for two service protocols $p_1$ and $p_2$, this section assesses the adaptability from two facets: computing the degree of adaptability, and identifying a set of conditions providing which service protocols are able to be adapted by means of SPM. In general, for a service protocol like $p_1$, the adaptation degree with respect to $p_2$ is reflected by the ratio of instance sub-protocols captured by $adapt_{graph}$ with all instance sub-protocols in $p_1$. As presented previously, a path in $adapt_{graph}$ leading from the initial state to one of the final states denotes an adaptation scenario between a pair of instance sub-protocols in $p_1$ and $p_2$ from the perspective of legal message exchange, however conditions associated with transitions cannot be verified statically. On the other hand, a conjunction of these conditions acts as another prerequisite that this pair of instance sub-protocols must be satisfied at runtime for ensuring a successful adaptation.

For exploring these two facets, in this section, we firstly generate all instance sub-protocols in a service protocol, and derive all pairs of instance sub-protocols from a complete adaptation graph where each pair captures a legal message exchange between two service protocols by means of SPM. Then the degree and the condition set are provided afterwards.

6.1 Generating Instance Sub-Protocols for a Service Protocol

Instance sub-protocol generation is reducible to the problem of computing all paths between a pair of vertices in a directed cyclic graph. However this path computation takes time exponential in the size of the graph in general. To the best of our knowledge, Algorithm $Path_{BTC}$ proposed in [11] is the only promising work that takes this path computation problem. However some assumptions made is not satisfiable to our context. For instance, Algorithm $Path_{BTC}$ requires that the function $CON$ (i.e. concatenate) is distribute over the function $AGG$ (i.e. aggregate). More clearly, assuming $\tau_1$, $\tau_2$ and $\tau_3$ are transitions in a service protocol $p$, $P_{12}$ are the paths in $p$ that contains $\tau_1$ and $\tau_2$, $P_{13}$ for $\tau_1$ and $\tau_3$, and $P_{1[23]}$ for $\tau_1$ with $\tau_2$ and/or $\tau_3$, then $P_{12} \cup P_{13} = P_{1[23]}$ is required. This distribute condition may not be satisfied in our context since $P_{1[23]}$ may include paths containing $\tau_1$, $\tau_2$ and $\tau_3$. In addition, self-loops, which are not considered in Algorithm $Path_{BTC}$, may exist in service protocols.
In the following we propose a technique to compute all paths in a service protocol that each path leads from the initial state to one of the final states. In general the procedure is composed of the following steps in sequential:

1. For simplicity, we firstly abstract away conditions and self-loops, and replace multiple transitions for a source state to a target state as a single transition. We will take them into account afterwards.

2. Then we identify back transitions in an FSA, where a back transition reflects a loop segment in this FSA specification, and the FSA without back transitions forms a DAG.

3. Without considering back transitions, we traverse this FSA for generating all paths leading from the initial state to one of the final states. Path means an alternating sequence of states and transitions in this paper. However, such a sequence is relatively not easy to capture its relation of difference and similarity with other paths. We then represent a path in terms of an FSA with one initial and one final states. Examples are shown in Fig. 6.

4. Leveraging the paths generated previously, back transitions are considered in the sense that more back transitions are contained in the paths.

5. After all paths are generated for the abstracted service protocol, we unfold the transitions in FSA that represent multiple ones in the service protocol between a source state and a target state. Afterwards self-loops are taken into account, and conditions are re-associated with relevant transitions. Consequently all paths for the service protocol are available.

6. Finally we examine each path with respect to dependency requirements. The paths free of dependency confliction are the instance sub-protocols.

We detail these steps in the following sections. $TS_{Sib}$, a sibling of $TS$ where a self-loop is specified upon the state Toy Items Selecting with a transition moreToyItems(+), is chosen as the running example. Because of the nature of time exponential, the size of the service protocol (i.e. the number of states and transitions) to be studied is recommended not to be very large.

### 6.1.1 Service Protocol Abstraction and Back Transitions Identification.

The abstraction of a service protocol is straightforward as depicted at Fig. 5.

Then we identify back transitions in this abstracted $TS_{Sib}$ where a transition is marked as a back transition if (1) the target state of this transition is not one of the final states, and (2) the distance between the initial state of abstracted $TS_{Sib}$ and the target state of this transition is not longer than that of the source state of this transition. Let us further explain this as follows. We borrow the concept of level from tree automata that specify the level of states in abstracted $TS_{Sib}$, where the initial state of abstracted $TS_{Sib}$ sits at level 1, and its neighboring states are at level 2, and so on. The knowledge of state level can be derived through a breadth-first search on the abstracted $TS_{Sib}$, while ignoring all transitions whose level of the target state is no less than that of the source state. Indeed the transitions ignored during the state level identification phase are the back transitions. As shown in Fig. 5, we denote the abstracted $TS_{Sib}$ with identified back transitions as $TS_{Sib}^{old}$, where back transitions are marked using dashed lines. Consequently, if back transitions are not considered, the remaining part of the abstracted $TS_{Sib}$ forms a DAG.
For instance in $TS_{Sib}^{bt}$, $\text{adjustToyItems}(+) \text{ is identified as a back transition since its source state Price Processing sits at level } 4 \text{ while its target state Toy Items Selecting, sitting at level } 3, \text{ is not one of the final states of the abstracted } TS_{Sib}^{bt}. \text{ If the target state is one final state, this transition is not a back transition. For instance, the transition } TS \rightarrow SPM : \text{ schedShip shown in Fig. 4, whose source state is (Payment Processing, Toy Items Shipped) leveled at } 10 \text{ and whose target state is (Toy Items Shipped, Toy Items Shipped) leveled at } 8, \text{ is not a back transition because the target state is a final state.}

It should be noted that, while ignoring back transitions, the remaining part of an abstracted service protocol may not be a finite state tree automata, although this is the case for $TS_{Sib}^{bt}$. Assuming that a service protocol is specified like $\text{adapt}_\text{graph}$ (note that the polarity of transitions must be changed to incoming or outgoing), then the state (Cust. Info. Handling, Price Processing), leveled at 5, is the child of both (Cust. Info. Handling, Toy Items Selecting) and (Start, Price Processing). For converting into a tree, we have to clone the states such as (Cust. Info. Handling, Price Processing). This technique can make the tree built, however, it may bring many new states and transitions, and hence, complicate the problem. On the other hand, either a tree automata or a DAG has not impact on the path computation that we detail as below.

6.1.2 Path Computation for $TS_{Sib}^{bt}$

We firstly generate all paths in $TS_{Sib}^{bt}$, without considering back transitions. This procedure is achieved by means of breadth-first search on $TS_{Sib}^{bt}$ and the result is two paths $path_1$ and $path_2$ shown on the left part of Fig. 6. It should be noted that a service protocol is sequentially executed, hence without considering back transitions, either Price Processing or Canceled, but not both, can

Figure 5: Abstracting $TS_{Sib}$ and back transitions identification.
be the next state of Toy Items Selecting.

Leveraging paths with $i$ ($i = 0, 1, 2, \ldots, k-1, k$ is the number of back-transitions in an FSA) back-transitions, we compute paths that contains $i+1$ back-transitions. For a back-transition $\tau$ ($s_s$ and $s_t$ denote its source and target states respectively) to be considered, we explore each path with $i$ back-transitions (denoted $p_{th}$) according to the following three situations. Since $TS^{bt}_{Sib}$ is simple and cannot cover all possible scenarios, we use a more general FSA (denoted $p$), whose snippet is shown in Fig. 7, to explain the procedure of our path computation technique.

1. If $p_{th}$ does not contain $s_s$ and $s_t$, then ignore.

2. First, we explore the situation that $p_{th}$ contains both $s_s$ and $s_t$. For instance, path in Fig. 7 contains both $s_3$ and $s_2$, which are the source and the target states of the back-transition.
We initialize two state sets $ST_{\text{upp}} = \{s_1\}$ and $ST_{\text{low}} = \{s_s\}$. In our case, $ST_{\text{upp}}^{b2} = \{s_2\}$ and $ST_{\text{low}}^{b2} = \{s_3\}$. We then explore back-transitions in $p_{th}$ for updating these two sets. For each back-transition $\tau_1$ (whose source state is $s_1$ and whose target state is $s_1^\bot$) in $p_{th}$, if $s_1^\bot \in ST_{\text{upp}}$ then $ST_{\text{upp}} = ST_{\text{upp}} \cup \{s_1^\bot\}$, and if $s_1^\bot \in ST_{\text{low}}$ then $ST_{\text{low}} = ST_{\text{low}} \cup \{s_1^\bot\}$. This process stops when no back-transition in $p_{th}$ can be explored anymore. The result for our case is that $ST_{\text{upp}}^{b2} = \{s_1, s_2\}$ and $ST_{\text{low}}^{b2} = \{s_3, s_4\}$. Indeed, $ST_{\text{upp}}$ and $ST_{\text{low}}$ include all back-transition
4. We then consider the situation that \( p_{th} \) contains \( s_t \) but not \( s_s \). For instance, \( pb_{th} \) in Fig. 7 contains \( s_2 \), the target state of the back transition \( b_2 \). We build \( ST_{upp} \) as presented above, and in our case, \( ST_{upp}^{b_2} = \{s_1, s_2\} \). \( ST_{upp} \) includes all back-transition chains in \( p_{th} \) that are extensible using \( \tau \). In our case, \( b_1 \) in \( pb_{th} \) is extensible using \( b_2 \), and thus, a longer back-transition chain: \( b_2 \) and \( b_1 \), is constructed.

As above, without considering back-transitions, we retrieve all segments (denoted \( SEG \)) in the FSA \( p \), such that, each segment starts at one state in \( ST_{upp} \) and ends at one state in \( ST_{low} \). In our case, \( SEG_{b_2} = \{s_2-t_4-s_3, s_2-t_4-s_3-t_5-s_4, s_2-t_3-s_4, s_1-t_1-s_3-t_5-s_4, s_1-t_2-s_2-t_4-s_3, s_1-t_2-s_2-t_4-s_3-t_5-s_4\} \). But for a segment \( seg \in SEG \), if all transitions in \( seg \) are contained in \( p_{th} \), \( seg \) is excluded. For \( SEG_{b_2} \), since some segments, such as \( s_2-t_4-s_3 \), are already contained in \( pa_{th} \), they are excluded, afterwards, \( SEG_{b_2} = \{s_2-t_3-s_4, s_1-t_1-s_3, s_1-t_1-s_3-t_5-s_4\} \). In addition, if two segments \( seg_1 \) and \( seg_2 \) in \( SEG \) are different only because \( seg_2 \) includes more transitions than \( seg_1 \), and these transitions are already in \( p_{th} \), \( seg_2 \) is excluded. The reason 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 \( p_{th}^1 \). Thereafter, \( SEG_{b_2} = \{s_2-t_3-s_4, s_1-t_1-s_3\} \), since the segment: \( s_1-t_1-s_3-t_5-s_4 \), is excluded. We use \( m \) to denote the number of segments in \( SEG \). In our case, \( m = 2 \).

Consequently, \( 2^m \) 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-transition chains 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 shown in Fig. 7. \( pa_{1_{th}} \) corresponds to the new path that no segment in \( SEG_{b_2} \) is included, \( pa_{2_{th}} \) and \( pa_{3_{th}} \) show the cases that one segment in \( SEG_{b_2} \) is considered, and \( pa_{4_{th}} \) is the new path that includes all these two segments in \( SEG_{b_2} \).

3. We then consider the situation that \( p_{th} \) contains \( s_t \) but not \( s_s \). For instance, \( pb_{th} \) in Fig. 7 contains \( s_2 \), the target state of the back transition \( b_2 \). We build \( ST_{upp} \) as presented above, and in our case, \( ST_{upp}^{b_2} = \{s_1, s_2\} \). \( ST_{upp} \) includes all back-transition chains in \( p_{th} \) that are extensible using \( \tau \). In our case, \( b_1 \) in \( pb_{th} \) is extensible using \( b_2 \), and thus, a longer back-transition chain: \( b_2 \) and \( b_1 \), is constructed.

As above, without considering back-transitions, we retrieve all segments (denoted \( SEG \)) in the FSA \( p \), such that, each segment starts at one state in \( ST_{upp} \) and ends at \( s_s \). In our case, \( SEG_{b_2} = \{s_2-t_4-s_3, s_1-t_1-s_3\} \), and \( m = 2 \), since the segment: \( s_1-t_2-s_2-t_4-s_3 \), is excluded. Consequently, \( 2^m - 1 \) new paths with \( i+1 \) back-transitions are generated. Each new path is made through cloning \( p_{th} \), adding \( \tau \), and including either one or multiple (even all) segments in \( SEG \). For \( pb_{th} \) and \( SEG_{b_2} \), \( 2^2 - 1 = 3 \) new paths (denoted \( pb_{1_{th}}, pb_{2_{th}} \), and \( pb_{3_{th}} \)) are generated as shown in Fig. 7.

4. We then consider the situation that \( p_{th} \) contains \( s_s \) but not \( s_t \). For instance, \( pc_{th} \) in Fig. 7 contains \( s_3 \), the source state of the back transition \( b_2 \). This situation is explored according to the following two sequential steps:

First, we build \( ST_{low} \) and construct \( SEG \). Each segment starts at \( s_t \) and ends at one state in \( ST_{low} \). In our case, \( ST_{low}^{b_2} = \{s_3, s_4\} \) and \( SEG_{b_2} = \{s_2-t_4-s_3, s_2-t_3-s_4\} \). Consequently, \( 2^m - 1 \) new paths with \( i+1 \) back-transitions are generated. Each new path is made through cloning \( p_{th} \), adding \( \tau \), and including either one or multiple (even all) segments in \( SEG \). For \( pc_{th} \) and
Without considering back-transitions, we retrieve all segments (denoted \(SEG_f\)) in the FSA \(p\). Each segment starts at \(s_i\) and ends at one final state of this FSA \(p\). Same as \(SEG\), some segments in \(SEG_f\), which are parts of \(p_{th}\), or are different merely because some include more transitions and these transitions are already in \(p_{th}\), are excluded. We use \(n\) denote the number of segments in \(SEG_f\). In our case, \(SEG_f^{2} = \{s2-t6-end\}\), \(n = 1\). Then, \(2^m \times n\) new paths with \(i+1\) back-transitions are generated. We denote \(s_{end}^{old}\) as the final state in \(p_{th}\), and \(s_{end}^{new}\) as the final state in a segment of \(SEG_f\). Each new path (denoted \(p_{th}^{new}\)) is made through cloning \(p_{th}\), adding \(\tau\), including either zero, or one, or multiple (even all) segments in \(SEG\), including one segment in \(SEG_f\), and removing \(s_{end}^{old}\) and other states that are not reachable to \(s_{end}^{new}\) in \(p_{th}^{new}\). As shown in Fig. 7, \(2^2 \times 1 = 4\) new paths (denoted \(pc_{th}^{4}, pc_{th}^{5}, pc_{th}^{6}, and pc_{th}^{7}\)) are generated for \(pc_{th}, SEG_{b2}\) and \(SEG_f^{2}\).

A new path is discarded if it is duplicate with another one that is generated previously. Path comparison is reducible to string comparison for implementation efficiency. By iteratively applying the rules above, all paths are generated. As shown in Fig. 6, there are 7 paths for \(TS_{Sib}^{old}\) in total. Since there are finite paths with \(i\) back-transitions, this process terminates with \(k\) times recursion, and each will check \(k\) back-transitions with respect to the last generated paths. Since path generation with \(i+1\) back-transitions relies on paths with \(i\) back-transitions, we append a queue with each transition in \(TS_{Sib}^{old}\) for recording the path number this transition belongs to. Then, resource for saving other paths with less than \(i\) back-transitions can be released.

Possibly a new path is duplicate with another that is generated previously. This requires path comparison which is not a trivial work. To lighten this job, we number the transitions in \(TS_{Sib}^{old}\), and alternatively list states and transitions in this new path according to transitions’ numbering. Then path comparison is reduced to string comparison which is implemented efficiently.

The number of paths is highly impacted by that of back transitions in \(TS_{Sib}^{old}\), and this number is possibly huge and hence requires large memory or disk space if all paths are stored individually. Indeed, path generation with \(i+1\) back transitions depends on the paths with \(i\) back transitions, while other paths are not relevant. Based on this observation, we number the paths generated, and append a queue with each transition in \(TS_{Sib}^{old}\) for recording the path number that this transition belongs to. Consequently the resource for saving paths with less than \(i\) back transitions can be released.

### 6.1.3 Instance Sub-Protocols Generation for a Service Protocol.

We firstly deal with folded transitions. In general, for a path that contains \(m\) folded transitions, and a folded transition \(\tau_j\) can be unfolded into \(t_j\) \((j \in [1, m])\) scenarios, then this path in \(TS_{Sib}^{b}\) (denoted as \(path\)) corresponds to \(\prod_{j=1}^{n} t_j\) paths in \(TS_{Sib}\) where \(\prod\) means multiplication. Then \(t_j\) is to be explored beforehand. Assuming \(\tau_j\) is folded from \(n\) transitions with the source state \(s_s\) and the target state \(s_t\), then if (1) \(\tau_j\) acts as a back transition in \(TS_{Sib}^{b}\), or (2) \(\tau_j\) is embedded in a loop segment, more clearly, there exists a back transition \(\tau_b\) in \(path\) with the source state \(s_s\) and the
target state $s_i^b$ providing that, $s_s^b$ is either the same state as $s_t$ or a descendant\(^2\) of $s_t$ in $TS_{Sib}^{bt}$, and $s_j^b$ is either the same state as $s_s$ or an ancestor of $s_s$ in $TS_{Sib}^{bt}$, then $t_j = 2^n - 1$ since either one, or two, ..., or all these $n$ transitions are possibly to be enabled in a certain execution. Otherwise $t_j = n$. Consequently the paths in $TS_{Sib}$ are generated through cloning $path$ but folded transitions are replaced by their combinatorial scenarios.

![Diagram](image)

Figure 8: Unfolding $price(-)$ in $path_6$ for computing paths of $TS_{Sib}$

For instance, $path_6$ has one folded transition $price(-)$ that is folded from two transitions $goldenPrice(-)$ and $normalPrice(-)$, and $price(-)$ is embedded in a loop segment, then as depicted by Fig. 8, $path_6$ corresponds to $2^2 - 1 = 3$ paths in $TS_{Sib}$. On the other hand, $path_4$ corresponds to 2 paths in $TS_{Sib}$. There are totally 17 paths for $TS_{Sib}$ after studying folded transitions.

Next we study the impact of self-loops to path computation. In general, for a path $path$ that contains $m$ self-loops on different states, $2^m - 1$ new paths are generated by means of cloning $path$ with combinational self-loops since either one, or two, ..., or $m$ self-loops are possibly enabled in a certain execution. $path$ represents the situation that no self-loop is to be considered. For instance, one self-loop is specified on the state $Toy Items Selecting$ with the transition $moreToyItems(+) in TS_{Sib}$, hence new paths are generated when this self-loop is considered and there are totally 34 paths for $TS_{Sib}$.

Thereafter we verify paths with respect to the minimal dependency graph, and reattach conditions with associated transitions. All paths free of dependency confliction are instance sub-protocols of service protocol. Assuming $TS_{Sib}$ shares the same dependency specification with $TS$, then $TS_{Sib}$ has 34 instance sub-protocols, whereas $TS$ has 17.

\(^2\)We borrow the notions of descendant and ancestor from tree automata to specify the following relations: for two states $s_1$ and $s_2$ in $TS_{Sib}^{bt}$, $s_2$ is called a descendant of $s_1$ if, without considering back transitions, there exists a path in $TS_{Sib}^{bt}$ leading from $s_1$ to $s_2$. The relation of ancestor is defined in a similar way.
From the procedure above, we see that the number of instance sub-protocols in a service protocol is tightly dependent on the complexity of this service protocol specification, rather than its size. For instance, one more self-loop is specified in $TS_{Sib}$ compared with $TS$, however the number of instance sub-protocol in $TS_{Sib}$ is double of that in $TS$. Another example is that $TS$ and $TR$ are not differ much from the size perspective. However, $TS$ has 17 instance sub-protocols but $TR$ has 2. Conclusively back transitions, self-loops, as well as multiple transitions sharing the source and target states, make a service protocol quite complex with many possible instance sub-protocols, and prevent the end user from grasping its possible behaviors.

6.2 Generating Instance Sub-Protocols Pairs from an Adaptation Graph

Leveraging an adaptation graph, this section explores how to generate pairs of instance service protocols that can make legal message exchanges through SPM. In general, this procedure includes the following two steps:

Firstly, we generate all paths in $adapt_{graph}$ using the technique presented in Section 6.1. Since there are eleven back transitions in $adapt_{graph}$, which is also quite complex, there are thousands of paths in this adaptation graph. Hence to compute all paths for an adaptation graph are usually inefficient.

Indeed, unlike transitions in a service protocol that all of them are necessary for achieving a particular business requirement, some back transitions in an adaptation graph may have no contribution to the static analysis and hence can be ignored. In fact, a back transition specifies another execution of a loop segment, however that may have been covered by the message exchange ahead of this back transition. The reason lies in that, a complete adaptation graph aims at capturing all possible legal message exchanges between service protocols. Consequently a loop segment may be enabled once again although it has been enabled beforehand. For instance, as shown in Fig. 9, a back transition $SPM \rightarrow TS : payment$, whose source state is (Price Processing, Payment Processing) and whose target state is (Payment Processing, Payment Processing), is unnecessary for the static analysis since this back transition contributes to another execution of following transitions: $adjustPayMethod(-)$ in $TS$ and $payment(+) in TS$. In principle, a back transition $\tau$ (with the source state $s_s$ and the target state $s_t$) can be ignored for the static analysis if:

1. Ignoring $\tau$ does not result in that the source state of $\tau$ is a dangling state. In other words, this back transition ignoring does not cause that some states in an adaptation graph are not reachable to some final states, whereas previously they can. An example is the back transition $TS \rightarrow SPM: normalPrice$, whose source state is (Toy Items Selecting, Price Processing) and whose target state is (Price Processing, Price Processing).

2. Another back transition $\tau_1$ exists in all message exchange traces leading from the initial state of the adaptation graph to $s_s$ providing that $\tau$ and $\tau_1$ associate with the same message set. An example is a back transition $TS \rightarrow SPM: \{normalPrice, goldenPrice\}$ whose source state is (Toy Items Selecting, Toy Items Shipped) and whose target state is (Price Processing, Toy Items Shipped).

3. Or this back transition exists in all message exchange traces leading from the initial state of adaptation graph to $s_s$ without considering back transitions. An example is the back transition $SPM \rightarrow TS: payment$ mentioned above.
The first rule can be verified through checking if \( s_s \) has other non-back outgoing transitions. While the last two rules can be checked by means of reversed breadth-first search on \( adapt_{graph} \) starting from \( s_s \). For a trace, once either of these two conditions is satisfiable, the search on this trace stops.

After checking for all back transitions, only two back transitions, which are marked by means of a dotted ellipse in Fig. 9, are necessary. Consequently there are totally 324 paths to be computed.

From the perspective of legal message exchange, each path leading from the initial state to one of the final states respects an adaptation trace between a pair of instance sub-protocols. However on the other hand, for a pair of instance sub-protocols, their legal message exchange traces are possibly multiple because some messages can be exchanged in different orders. For instance, transitions \( adjustPayMethod(+) \) in \( TS \) and \( custInfo(-) \) in \( TR \) are contained in different instance sub-protocols of \( TS \) and \( TR \) respectively, however they can be enabled in any order. Consequently, multiple paths may reflect the adaptation traces between the same pair of instance sub-protocols.

To make the instance sub-protocol pairs unique, as the second step, we cluster paths providing that they are captured by a same pair of instance sub-protocols. This requires a technique for
identifying the pair of instance sub-protocols that a path reflects. In general, we firstly project a path to a service protocol, and 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 the transitions in a path associating with a service protocol, and restores the polarity of these transitions according to the service protocol specification. Then we identify which instance sub-protocol this projected path belongs to. This is achieved by means of comparing the transition set in this projected path with that in an instance sub-protocol. Hence the instance sub-protocol pair is provided. Consequently we study other paths following this way. It should be noted that, during the path clustering procedure, the problem of path comparison is reduced to that of string comparison for complexity consideration. Back to our example, 324 paths in adapt\_graph are clustered into 8 instance sub-protocol pairs.

### 6.3 Computing the Adaptability Degree and the Condition Set

After studying the instance sub-protocol set in a service protocol and their pairs captured by an adaptation graph, in this section, we explore how they can be applied for computing the degree of adaptability and the condition set.

For two service protocols $p_1$ and $p_2$ with a complete adaptation graph adapt\_graph, the adaptability degree is defined as Equation 1 where the function $\text{instSubProtocol}(p_1, \text{adapt}\_\text{graph})$ aims at counting the instance sub-protocols in $p_1$ that are captured by instance sub-protocol pairs of adapt\_graph. If adapt\_graph is set to null, then the number of instance sub-protocols in $p_1$ is returned.

$$adaptation(p_1, p_2) = \frac{\text{instSubProtocol}(p_1, \text{adapt}\_\text{graph})}{\text{instSubProtocol}(p_1, \text{null})}$$  \hspace{1cm} (1)$$

For instance, $adaptation(TS, TR) = 8/17 \approx 0.471$, whereas $adaptation(TR, TS) = 2/2 = 1$. This shows that the adaptability is an asymmetric relation between service protocols. Back to our motivating example, $adaptation(TR, TS_A) = 1/2 = 0.5$, hence the adaptability degree provides to the client the knowledge of different adaptability possibility of candidates.

Next we re-explore the partial and full adaptability problem, which is pending in Section 5, 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 have a legal message exchange with at least one instance sub-protocol in $p_2$. Otherwise, $p_1$ is partially adaptable with $p_2$.

We recall that an adaptation graph captures legal message exchanges between pairs of instance sub-protocols. However for a particular pair, the conjunction of conditions associated with their transitions serves as another prerequisite for ensuring a successful adaptation. For instance, $path^2_6$ depicted in Fig. 8, which is actually an instance sub-protocol of TS, is adaptable with an instance sub-protocol of TR that leads TR to the final state Toy Items Shipped. This adaptation requires that $Cd_2 \land Cd_3 \land Cd_4$ is satisfiable. Such a must-be-held condition set for two service protocols are generated through studying the instance sub-protocols pairs provided by Section 6.2. For TS and TR, the condition set is as follows: \{Cd_2 \land Cd_3 \land Cd_4, Cd_3 \land Cd_2 \land Cd_3 \land Cd_4, Cd_2 \land Cd_4, Cd_2 \land Cd_5, Cd_1 \land Cd_2 \land Cd_4\}. It should be noted that some conditions contain both $Cd_1$ and $Cd_2$. This reflects the fact that, in some situation, when TS loops back to the state Toy Items Selecting, the transition custInfo(-) in TR has been fired and the message custInfo can make $Cd_1$ satisfiable.
Indeed, the behavior of a service protocol at runtime depends on the evaluation of its branching and other conditions while these conditions depend on the exchanged message instances. A *must-be-held* condition includes branching conditions (such as $Cd_1$ and $Cd_2$ in $TS$, and $Cd_4$ and $Cd_5$ in $TR$) that guide service protocols to evolve as desired at runtime, and other conditions (such as $Cd_3$ in $TS$) that guard transitions to be enabled in a proper situation. For instance, path $p_6$ is to be adapted with an instance sub-protocol of $TR$ that leads $TR$ to *Toy Items Shipped*, $Cd_2$ and $Cd_4$ leads $TS$ and $TR$ to the desired branch, while $Cd_3$ ensures that relevant business requirements are respected. This example demonstrates the importance and the contribution of conditions for static analysis like service protocol adaptability.

7 Prototype Implementation

This section briefly describes the architecture of the prototype we developed to implement our approach. As shown in Fig. 10 we distinguish three layers.

*User Interface.* This layer supports service protocol modeling, and conducts the adaptability assessment by calling the functionality provided by the logical layer. *JGraphPad* ([http://www.jgraph.com/jgraphpad.html](http://www.jgraph.com/jgraphpad.html)) is plugged into this layer which acts as the GUI for displaying resources including service protocols, the adaptation graph, instance sub-protocols, and dependency graphs.

*Persistence Layer.* This layer aims at serializing resources, and supporting the access of resources by the logical layer.

*Logical Layer.* This is the main part of prototype where (1) an adaptation graph is generated for two service protocols, (2) paths are computed for a service protocol as well as an adaptation graph, (3) back transitions are analyzed for an adaptation graph, and (4) adaptability is assessed according to the work above.

![Diagram of the layered architecture of the prototype](http://www.jgraph.com/jgraphpad.html)

Figure 10: The layered architecture of our prototype

The context of this paper is the mediated Web service interaction. However our technique is generic and can be applied for supporting the interaction of component protocols. Due to this concern, service protocols are represented using *GXL* ([http://www.gupro.de/GXL/](http://www.gupro.de/GXL/)) which is an XML-based standard exchange format for graphs. *GXL* is one input format of *JGraph* which is a Java-based, powerful, easy-to-use, feature-rich and standards-compliant open source graph component. *JGraph* has been widely used by academic and industry.
8 Related Work and Conclusion

We discuss the related work in respect to adaptability analysis, adapter construction, and compatibility analysis.

**Adaptability analysis.** The work similar to us is the adapter compatibility analysis in Y-S model [23], which checks whether or not two component protocols are adaptable with a certain adapter. The criteria are no unspecified receptions and 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 such a limitation, and we compute an adaptability degree which is more accurate than a binary answer. Since conditions in protocols are not explored, this work does not specify conditions providing which two protocols are adaptable. Whereas our approach specifies such necessary conditions. In addition, this work depends on the synchronous semantics, which makes the problem simple but fails to capture most Web service interactions since they are normally asynchronous. Our work does not depend on such an assumption.

**Adapter construction.** Adapters are important for supporting interactions of both software components [3] [23] and Web services [1] [5] [8] [17] [25].

In [3] [23], an adapter is automatically built with respect to two incompatible protocols. The adapter tackles order mismatch with unspecified receptions, but considers any deadlock as unsolvable. In [1] [5] [8], possible mismatches are categorized into several classes (called mismatch patterns), and then adaptation templates [1] [5] or adaptation operators [8] are proposed for handling these mismatch patterns. However, for two protocols, mismatches between them are identified by a developer and an adapter is constructed manually.

Besides mismatches covered by the adapters above, [17] can handle a deadlock through evidences. To choose which evidence for resolving a certain deadlock is up to adapter developers’ decision. So this method is somehow not generic. This deadlock resolution assumes that recommended business data is consistent with a certain interaction context. However, recommended business data by some evidences: such as enumeration with default and log based value/type interface, may not hold this assumption. The reason is that enumeration business data may not be the default in a certain interaction, and some business data may differ in different interactions. On the contrary, our SPM produces a mock-up message [25] to automatically resolve a deadlock relating to optional data dependencies, and this mock-up message is consistent with the certain interaction context.

In [14] [4], adapters are automatically constructed if service protocols are not compatible. However, mismatches resolved are only the ordering mismatch [1] between messages. Consequently, these adapters are limited compared with the other approaches as discussed above.

In short, adapter construction means only that protocols are adaptable in some cases. However the possibility, and the conditions providing which they are adaptable, are not specified. In general, adapter building constitutes a starting point, but is insufficient, for assessing adaptability. Whereas our approach gives a degree which differentiates higher or lower adaptability possibility, and a set of must-be-held conditions prescribing allowed behaviors of participating protocols.

**Compatibility analysis.** Compatibility can be analyzed either through user-defined approaches [2] [20] or by means of formal methods [9] [10].

Partially and fully protocol compatibilities are defined in [2] which depend on general protocol operators, and forward and backward compatibilities are proposed in [20] for supporting protocol
evolution. Rather than a binary answer provided by [10] [15] [23], partial compatibility represents the situation of neither fully nor not compatible. But it is still inaccurate since it does not specify the possibility that protocols can interact. On the contrary, our adaptability degree captures the difference of adaptation possibility between protocols.

Another class of method uses formal methods like model-checking. In [9], protocol interactions are analyzed using SPIN. In [10], Web processes are modeled by petri nets, then a problem of compatibility is converted into that of reachability of the composed process. An advantage of formal methods is that a property like compatibility is thoroughly verified to see if protocols are always or sometime satisfied. But same as [2], partial compatibility is not further studied.

In general, the context of compatibility analysis approaches is the direct service interaction. They result in a binary or ternary answer which is inaccurate. Besides [9] which considers guards (conditions in fact), conditions in protocols are not considered. But even in [9], no conditions are given where protocols can interact. These approaches inspire our adaptability assessment, albeit they do not give the level of assessment that our adaptability assessment provides.

In summary, three innovative contributions this paper brings are as follows. For two service protocols, we firstly characterize if they are adaptable through constructing an adaptation graph. Secondly we compute instance sub-protocols in a service protocol, and generate all instance sub-protocol pairs in an adaptation graph. They enable to quantify the adaptability degree. Finally, we provide the condition set where a condition is the conjunction of conditions specified on transitions in a pair of instance sub-protocols. We believe that all of these are valuable knowledge to the client for judging and then selecting the most suitable candidate according to certain requirements.

As to the future work, we are optimizing the path computation using the single-entry-single-exit fragment technique and hence following the divide and conquer strategy.

References


