Formal Verification of Mediated Web Service Interactions Considering Client's Expected Behaviours

Zhangbing Zhou, Sami Bhiri, Lei Shu and Manfred Hauswirth
Digital Enterprise Research Institute, National University of Ireland, Galway
{firstname.lastname}@deri.org

Abstract—**This paper proposes a formal technique to verify whether or not an expected interaction is adaptable. We first present our observation that a mediated service interaction is synchronizable. This fact is a prerequisite of our approach. Hereafter, we formally model a protocol scenario (i.e., a part of a service protocol to be enacted in an expected interaction) and an adapter, generate an adaptation logic, and formalize a mediated service interaction and its conversation. These formalization enables one to perform a formal verification that checks whether or not, as well as under which condition, an expected interaction is achievable. The technique presented in this paper complements the efforts of adapter synthesization for ensuring the achievability of a certain expected interaction.**

I. Introduction

Given the inherent autonomy, heterogeneity, and continuous evolution of Web services, mediated service interactions are a common style of service interactions [1]. Nowadays, standards such as BPEL and WS-CDL lay out the foundation that industry can build upon. Research efforts relating service interactions either analyze service compositions [2], [3] that target direct service interactions, or synthesize adapters [1], [4], [5] that facilitate mediated service interactions through identifying and reconciling mismatches. Generally, these techniques support service interactions from a global behavioral perspective without considering the client's expectations. Concretely, for service protocols of a client and a provider, it is common that they can form many possible (direct or mediated) service interactions. However, according to the client's requirement, just a few (in percentage) are interesting whereas the others are somehow not relevant. For instance in the motivating example, assume that only the interaction leading *Toy Items* to be delivered is expected. Then the selecting criterion for a suitable service provider is the support of these expected interactions, whereas the other interactions are complementary but not mandatory. To support this selection criterion, in this paper, we propose a formal technique that verifies whether or not, and provides conditions that determine when, the expected interactions can be properly mediated [6].

As reviewed in [6], previous efforts have been conducted for synthesizing adapters. However, synthesizing an adapter does not provide the level of evaluation we target. Generally, the existence of an adapter indicates that there are some interactions possible for service protocols of a client and a provider, but, an adapter itself does not inform the client about whether an expected interaction is supportable, although this knowledge may be derived through simulating the message exchange by means of the adapter. In addition, an adapter does not prescribe the condition that determine when an interaction is possible. Hence, even if an expected interaction is achievable from message exchange perspective, it may fail because of unsatisfiability of some conditions according to exchanged message instances.

A. Motivating Example

Figure 1 depicts three service protocols including two toy shop services (denoted TS_1 and TS_2) and one toy requestor service (denoted REQ). *Rec.*, *Rep.*, and *Inv.* means *receive*, *reply*, and *invoke* respectively. *Switch* and transition conditions (denoted Cd_i ($i \in [1,9]$)) are associated to links. *Switch* conditions refer to the conditions specified on conditional branches in *Switch* blocks.

The difference between TS_1 and TS_2 is that, TS_1 may apply a discount on the price depending on *Cust. Info.*, and hence, *Cust. Info.* is expected before the price is decided. However, *Normal Price* always applies by default. On the other hand, TS_2 can provide a price only if the client is confirmed to be an adult. In addition, some conditions specified on the links of TS_1 and TS_2 (such as Cd_1 in TS_1 and Cd_6 in TS_2) are different.

Without the loss of generality, we assume that a client, using REQ, chooses from TS_1 or TS_2 to interact with for buying some toys online as a gift for her kid. The expected result is *Toy Items* being delivered.

We first explore the support of the expected interaction between TS_1 and REQ. Due to privacy concerns, REQ sends *Cust. Info.* only if she is convinced by the price and is committed to buy. Consequently, a deadlock occurs that TS_1 is requesting *Cust. Info.* before sending the *Price*, while REQ is expecting the *Price* before deciding upon continuation and sending *Cust. Info.* or not. Such (kind of) deadlock is reconcilable according to the adaptation mechanisms of [4] (through providing missing *Cust. Info.* using *evidences*) and [5] (through generating *mock-up Cust. Info.* message), but is beyond the capacity of other adapters [1], [7] since they consider any deadlock as an unreconcilable mismatch.

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8270
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8270

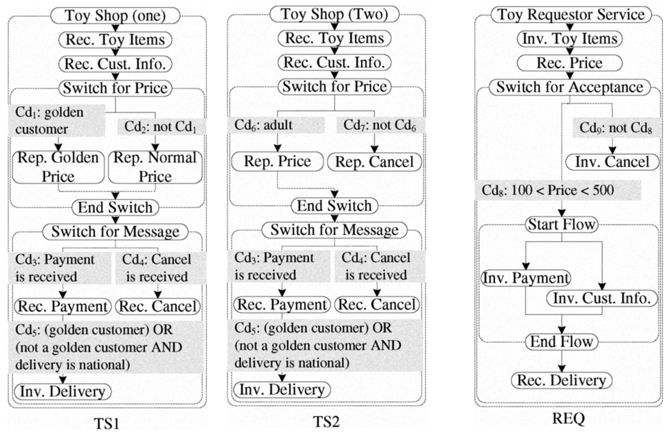


Figure 1. Service protocols for two toy shop services (i.e., TS_1 and TS_2) and one toy requestor service (i.e., REQ). A client, using REQ, can choose from TS_1 or TS_2 to interact with for buying some toys online

We thereafter explore the support of the expected interaction between TS_2 and REQ. As discussed in the paragraph above, [4] can support it through providing missing *Cust. Info.* using *evidences*, although this *Cust. Info.* may not be consistent with the interaction context since it may be different from what is provided by REQ afterwards (through *Inv. Cust. Info.* activity). On the other hand, the deadlock is unresolvable according to [5] since the condition Cd_6 cannot be enabled using a *mock-up Cust. Info.* message.

From the reasoning above, the client got to know that TS_1 supports the expected interaction using an adapter of [4], [5], but TS_2 supports it using an adapter of [4] only.

Naturally, the client is also interesting in whether she can cancel the interaction in case the price is unacceptable. According to the reasoning above, through an adapter of [4], [5], both TS_1 and TS_2 can support this expected interaction that leads both TS_1 (or TS_2) and REQ to their cancellation.

Consequently, the client is informed that TS_1 and TS_2 are both suitable candidate service providers if an adapter is used according to [4], while TS_1 is more suitable if an adapter is used according to [5]. However, an adapter such as [4], [5] can specify whether or not an interaction is possible, but it cannot indicate whether a specific (maybe also an expected) interaction is supportable.

The discussion above explores possible message exchanges between service protocols. However, the success of an adaptation in particular and an interaction in general depends on *conditions* that decide which branches to follow in *Switch* blocks and guard transitions between activities. For instance, TS_1 and REQ are adaptable such that *Toy Items* can be delivered. A prerequisite is that the condition $Cd_2 \bigwedge Cd_3 \bigwedge Cd_5 \bigwedge Cd_8$ is satisfiable, such that Cd_2, Cd_3 and Cd_8 lead TS_1 and REQ to choose the desired branches, while Cd_5 ensures meeting specific business requirements. Identifying these conditions is beyond the existing capability of adapter synthesization techniques.

B. Overview of Our Approach

1) Context: Studying service interactions formally is an active research area and several methods like [2], [3] have been proposed. [2] proposes an inspiring method to study interacting BPEL processes, which identifies sufficient conditions that determine when the conversation set for bottom-up specified service compositions remains the same for synchronous and asynchronous communication semantics. In this paper we use the technique proposed in [2] to perform our verification, thanks to our observation that a mediated service interaction is synchronizable. Another major concern is that we can reuse the tool developed in [2].

We conduct the verification in accordance to our Space-based Process Mediator (SPM) which is detailed in [5]. Note that the technique proposed in this paper is general and can be applied to other adapters as well.

2) Approach: We first present our observation in Section III that a mediated service interaction is synchronizable. Hereafter in Section IV, we formalize a protocol scenario and an adapter protocol in terms of Guard Finite State Automata (GFSA) [2]. Note that BPEL is not suitable as a service protocol modeling method in formal approaches. GFSA is a finite state automata with guards specified on transitions [2], where guards corresponds to conditions in BPEL specification, and hence, we use the notions of guard and condition interchangeably afterwards. A protocol scenario is a part of a service protocol, which can be enacted in a particular (maybe expected) interaction depending on the evaluation of conditions in *Switch* branches. We study mediated service interactions at a protocol scenario level since a protocol scenario can represent the part of a service protocol involving in an expected interaction. In addition, the conjunction of transition guards in these interacting protocol scenarios constitutes a condition that determines when these service protocols can properly perform the expected interaction (possibly through adaptation).

Then, an adaptation logic in respect to interacting protocol scenarios is generated. An adaptation logic is closely coupled with the specific protocol scenarios, albeit the adaptation mechanism of an adapter (like the SPM) is general.

Consequently, we formalize mediated service interactions and their conversations, and represent relevant properties in terms of LTL formulae (in Section V), which can be verified through Web Service Analysis Tool (WSAT) [2]. The reason of reusing existing WSAT is that, WSAT is mature for analyzing service compositions with synchronous semantics. In addition, a translation from BPEL to GFSA and then to Promela (the input language of SPIN generated by WSAT) can be reused which is not a trivial task.

II. Preliminaries

A. Control and Data Dependencies

A service protocol specifies sequencing constraints between a finite set of semantic activities [8] using *Sequence*, *Switch*, *Flow*, and *Loop* control structures. An activity sends or receives a message that contains business data. An activity is semantically described by specifying its *input*, *output*, *precondition*, and *effect* (so-called IOPE, and refer to OWL-S specification for details). The *input* and *output* define consumed and produced messages respectively. The *precondition* and *effect* represent the respective state of the world before and after the execution of an activity. For instance, as proposed in BPEL4SWS, one can describe the activity implementation using semantic Web services (i.e., OWL-S or WSMO), and can apply $BPEL^{light}$ [9] to support the execution of BPEL4SWS processes. A service protocol may specify conditions that guard transitions between activities. These conditions refer to transition guards in GFSA.

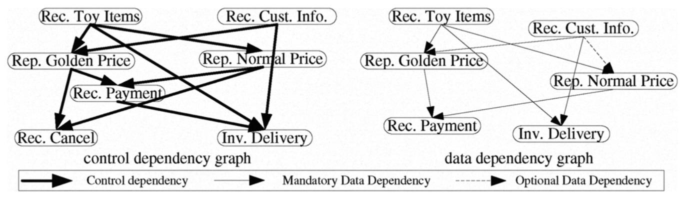


Figure 2. Control and data dependency graphs for TS_1 service protocol

As discussed in [10], different kinds of dependencies exist between activities, which are often obfuscated by a service protocol specification that defines all possible activity execution sequencing. A sequencing constraint in a service protocol may result from one or multiple kinds of dependency [10]. In our approach, we consider two kinds, namely control and data dependencies. Generally,

- An activity act_b is control dependent on another activity act_a if the *completion* of act_a (marked by its *effect*) is a necessary condition for the *enablement* of act_b (guarded by its *precondition*).
- Data dependencies are classified as mandatory or optional. act_b is mandatorily data dependent on act_a if common data exist between the *output* of act_a and the *input* of act_b, while act_b is optionally data dependent on act_a if no common data exist between the *output* of act_a and the *input* of act_b, but incoming conditions of act_b use the data in the *output* of act_a.

[10] claims that different kinds of dependencies between activities can be extracted from design documents. In our approach, we extract control and data dependencies from the semantic description of activities [8]. As illustrated in Figure 2, we distinguish between control and data dependency graphs of a service protocol that specify a finite set of *asymmetric*, *irreflexive* and *transitive* relations between activities. Dependency relations between activities are used in constructing an adaptation logic (see Section IV-C).

B. WSAT: Web Service Analysis Tool

WSAT (http://www.cs.ucsb.edu/~su/WSAT/) is a formal specification, verification and analysis tool for analyzing Web service compositions that tackles the following challenges: (1) automata with XPath guards (GFSA) is used as the intermediate representation of BPEL processes, (2) synchronizability and realizability analyses, and (3) handling XML data manipulation.

We next present sufficient conditions of synchronizability analysis in brief since a mediated service interaction follows a bottom-up manner. A GFSA service composition is synchronizable if it is (1) synchronous compatible and (2) autonomous. A synchronous compatible condition requires that each sent message should be received by the peer(s) immediately or reachable through ϵ-transitions (ϵ means a local transition). An autonomous condition requires that each peer, at any moment, can either terminate or send/receive a message. An autonomous condition can be relaxed for a *Flow* block as a single-entry single-exit permutation block. More details about the synchronizability analysis and WSAT can be found in Fu's Ph.D. thesis [2].

III. Mediated Service Interactions

A mediated service interaction follows a centralized architecture where an adapter acts as a centric *arbiter* to collaborate with interacting service protocols [1]. Generally, the adapter intercepts all messages sent by service protocols, stores or transforms these messages or even generates new messages (such as the acknowledgement (i.e., ACK) [4], [5], or new messages using *evidences* [4], or *mock-up* messages [5]), and thereafter, sends a message to a service protocol when this protocol is ready to receive this message.

In the following sections, we first justify our observation that a mediated service interaction is synchronizable, and we then analyze the computation complexity of our technique which shows that our technique is applicable to service protocols of practical relevance.

A. Synchronizability of Mediated Service Interactions

In this section, we present Theorem 1 which prove our observation that a mediated service interaction is synchronizable. This fact is important to our approach because, for a bottom-up specified service composition like a mediated service interaction, if it is not synchronizable, the asynchronous communication and unbounded input queues may cause the undecidability of LTL verification [2]. Hence, Theorem 1 is indeed a prerequisite for our formal verification of a mediated service interaction.

Theorem 1. *A mediated service interaction is synchronizable according to the sufficient conditions proposed in [2].*

Proof. *A mediated service interaction can be regarded as a service composition which is composed of multiple service protocols and a centric adapter. We prove that such a service composition satisfies sufficient conditions of synchronizability analysis [2]:*

(1) ***Synchronous compatible condition****. In a mediated service interaction, each service protocol sends its messages to the adapter, rather than to peer protocols directly. Afterwards, the adapter forwards these messages to the desired peer protocols whenever they are ready to receive them. This message exchange mechanism indicates that the synchronous compatible condition is satisfiable since (1) any message, which is to be sent by any service protocol, is ready to be received by the adapter immediately or reachable via ϵ-transitions, and (2) the same situation holds for any message that is to be sent by the adapter to a service protocol.*

(2) ***Autonomous condition****. Here, a relaxed autonomous condition is taken into account. After abstracting any* Flow *block into a single-entry single-exit permutation block,*

1) *any service protocol satisfies relaxed autonomous condition since, at any moment, it either (1) sends a message to the adapter, or (2) receives a message from the adapter, or (3) executes ϵ-transitions for performing its local (also known as internal) logic. Note that activities in* Switch *blocks do not violate this conclusion, since only one branch can be enabled in a certain interaction.*
2) *the adapter satisfies relaxed autonomous condition since, at any moment, it either (1) receives a message from a service protocol, or (2) sends a message (either a message produced by a service protocol, or a new message generated by the adapter itself) to a service protocol, or (3) executes ϵ-transitions for performing its local (also known as internal) logic.*

Hence, a relaxed autonomous condition is satisfiable.

The reasoning above indicates that any mediated service interaction satisfies sufficient conditions of synchronizability analysis, and hence, we conclude that a mediated service interaction is synchronizable.

Note that there exist some interactions which can be properly mediated and are synchronizable, but they may not bring any practical effort. This kind of interactions are outside the scope of this paper. For instance, considering two service protocols that send messages only, these two service protocols are adaptable according to [1], [4], [5], [11], and the interaction between them is synchronizable according to Theorem 1. However, this interaction does not bring any value-added effect of practical relevance.

In this paper, we study mediated service interactions at a protocol scenario level. A protocol scenario is indeed a *smaller* service protocol in size. A definition and examples for the protocol scenario are presented in Section IV. Hence, Theorem 1 holds for our approach.

B. Computational Complexity Analysis

In this section, we explore the computation complexity of our approach. For service protocols of a client and a provider, we conduct the verification of all combinatorial protocol scenarios that are interesting to the client. A service protocol is observed by [12] to be a fairly simple model because a service protocol, as well as a service in general, is designed by humans. This means that the number of protocol scenarios in a service protocol is typically not large. In addition, there are only a few (in percentage) interactions of all combinatorial interactions that are interesting to the client. For instance for TS_1 and REQ, there are $4 \times 2 = 8$ combinatorial interactions since TS_1 has four protocol scenarios and REQ also has two. However, there are only four interactions interesting to the client which lead TS_1 and REQ to either (1) *Toy Items* to be delivered, or (2) both TS_1 and REQ to their cancellation.

To support the discussion of computation complexity above, we have conducted a survey about the size of BPEL service protocols of practical relevance and the number of protocol scenarios in these service protocols, which are presented in Table I. The column "Source" shows where the samples come from and "BPEL Process" shows the name of the sample processes. The columns "#activity" and "#proSce" represent the number of activities, and protocol scenarios, in these sample BPEL protocols, respectively.

Sample Set			
Source	BPEL Process	#activity	#proSce
[1]	eBay service	6	1
	TPC service	4	1
SUPER project	Fulfilment	12	4
	OrderFulfillment	16	2
	ContentProvision	5	1
active endpoints	loanApprovalProcess	5	1
	marketplace	6	2
OMII BPEL	conditionalworkflow	8	3
	echoworkflow	3	1
Oracle SOA DEMO	BPELProcess1	3	1
	DHLShipment	1	1
	SelectManufacturer	3	1
	SOAOrderBooking	*39 / 15*	*16 / 2*

Table I
SAMPLE BPEL SERVICE PROTOCOLS OF PRACTICAL RELEVANCE FOR SHOWING THE NUMBER OF ACTIVITIES AND PROTOCOL SCENARIOS IN THESE SERVICE PROTOCOLS

Besides *SOAOrderBooking*, other samples are small in the number of both activities and protocol scenarios. We have taken a close look at *SOAOrderBooking*. Within these thirty-nine activities, two are empty activities and another twenty-two are assign activities. Since empty and assign activities represent the internal logic, and hence, there are only fifteen activities relating to the interaction with the partner. Because of the embedded *Switch* blocks, *SOAOrderBooking* has sixteen protocol scenarios. However, most branches of these *Switch* blocks specify different internal logic (through empty or assign activity). Hence, several protocol scenarios are the same to the partner although they are different in their internal processing logic. The context of this paper is the interaction within service protocols, and the difference of internal logic is outside of our interest. Hence, there are only two protocol scenarios which are different considering the possible interaction with a partner service protocol.

Besides, Figure 7 in [13] presents twelve realistic protocols which are used by Fu et al for examining the applicability of synchronizability and realizability analyses. They are consistent with our survey that a service protocol is normally not big in size (i.e., the protocol biggest in size has only twelve states and fifteen transitions).

The survey above is aligned with claims made in [14], [15] about the size of service protocols of practical relevance. Consequently, we can conclude that the technique presented in this paper is computationally not complex with regard to service protocols of practical relevance.

IV. Modeling Mediated Service Interactions

We first formalize a protocol scenario and an adapter using GFSA and generate an adaptation logic. Thereafter, we formalize mediated service interactions and their conversations that are ready for the formal verification purpose.

We formalize the protocol scenario and the adapter using GFSA which are suitable for formal verification purposes. On the other hand, their examples are represented in BPEL processes. Two major reasons support this choice. First, compared with BPEL, GFSA is relatively complex and it is not easy to capture concurrent structures. Moreover, the input format of the WSAT is a BPEL specification, which is automatically translated into a GFSA by the WSAT.

A. Modeling a Protocol Scenario

Definition 1 (Protocol Scenario). *A protocol scenario is a tuple sce = (M, Σ, S, s, f, δ). $M = M^{in} \cup M^{out}$ is a finite set of message classes where M^{in} is for incoming and M^{out} is for outgoing message classes. Σ is a finite set of messages in respect to M. S is a finite set of states where s is the initial state and f is the final state. A protocol scenario has one finite state. δ is a finite set of transitions. A transition $\tau \in \delta$ can be one of the following three types:*

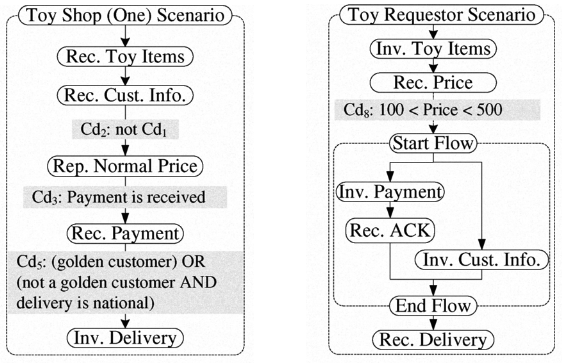
1) *a send transition (s_1, ($!\alpha$, (g | true)), s_2) where s_1, $s_2 \in S$ such that, s_1 is the source state and s_2 is the destination state. $\alpha \in M^{out}$, and g is a transition guard that determines whether this transition can be enabled according to exchanged message instances [2]. If no guard is specified, the guard of default is true. A send transition sends α to an adapter.*
2) *a receive transition (s_1, ($?\beta$, (g | true)), s_2). $\beta \in M^{in}$. A receive transition blocks to wait β from an adapter.*
3) *a local transition (s_1, (ϵ, true), s_2) to realize the local (also known as internal) logic.*

A transition evolves the state of sce from s_1 to s_2.

! and ? denote send and receive actions respectively.

A state of a protocol scenario is a unique configuration of its properties including exchanged message instances, states of transition guards, and states of transitions.

A transition guard can be one of the following three states: *fulfilled*, *violated*, or *undefined*. It starts at *undefined*, and evolves to *fulfilled* or *violated* relying on the evaluation of message instances. A transition is in one of the following four states: *initial*, *enabled*, *completed*, or *failed*. It starts at *initial*, becomes *enabled* if related transition guards are *fulfilled* and all immediately preceding transitions are in state *completed*, and moves into *completed* if it is executed successfully or into *failed* otherwise.



sce_{TS1}: a protocol scenario of TS1 sce_{REQ}: a protocol scenario of REQ

Figure 3. Protocol scenarios for TS_1 and REQ service protocols

The generation of protocol scenarios for a service protocol is intuitive. Generally, a protocol scenario is inherited from the service protocol, while keeping only one exclusive branch for each *Switch* block. Figure 3 illustrates two protocol scenarios in terms of BPEL processes, denoted sce_{TS1} and sce_{REQ}, for TS_1 and REQ respectively. sce_{TS1} and sce_{REQ} can lead TS_1 and REQ to their expected interaction: *Toy Items* to be delivered. TS_1 has four protocol scenarios, and REQ has two.

The control and data dependency graphs of a protocol scenario are directly extracted from those of the service protocol through keeping these dependency relations whose source and destination activities are both in this protocol scenario. This step is straightforward, and thus, we do not give an example to show it.

B. Modeling an Adapter

Definition 2 (Adapter). *An adapter is a tuple adt = (M, Σ, S, s, f, δ). M, Σ, S, s, f are the same as those in Definition 1. δ is a finite set of transitions, where a transition $\tau \in \delta$ can be one of the following three types:*

1) *a receive transition (s_1, ($?\beta$, true), s_2). $\beta \in M^{in}$. The guard is true by default, which indicates that an adapter receives all messages produced by protocol scenarios without conditions.*
2) *a send transition (s_1, ($!\alpha$, g), s_2). $\alpha \in M^{out}$. An adapter sends α to a protocol scenario in case of:*

Case 1: an adapter sends a message α, which is produced by a protocol scenario, to a peer protocol scenario if this peer protocol scenario may consume α. The transition guard g refers to the function isInterest() [1] *that identifies if there exists a transition in a certain protocol scenario, with a state* initial, enabled, *or* completed, *that receives α.*
Case 2: an adapter generates an ACK *and sends this* ACK *to a protocol scenario. This happens when a deadlock encounters such that, all protocol scenarios expect messages, but some protocol scenarios expect* ACK*. The transition guard g is the function expACK4ReceivedMsg() which identifies if one of the currently enabled transitions is expecting an* ACK*.*
Case 3: an adapter generates a mock-up *message and sends it to a protocol scenario. This happens when a deadlock encounters such that, all protocol scenarios expect messages, but no protocol scenario expects an* ACK*. The transition guard g is the function chkOptionalDataDependent() which identifies if a certain protocol scenario is expecting a message, and this message is optional to one of the immediately succeeding transitions.*

3) *a local transition (s_1, (ϵ, true), s_2) which generates an* ACK *or a* mock-up *message or performs the local (also known as internal) logic.*

A transition evolves the state of adt from s_1 to s_2.

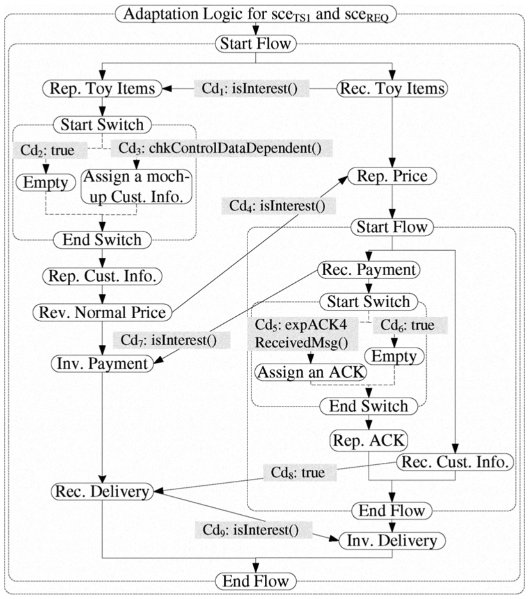


Figure 4. The adaptation logic for two protocol scenarios sce_{TS1} and sce_{REQ} according to the mechanisms of SPM

[1]Note that for space consideration, we do not present the functions *isInterest*() and *chkOptionalDataDependent*() in this paper, whereas they are detailed in our technical report [16]. The function *expACK4ReceivedMsg*() is intuitive and hence is not detailed in this paper.

An example adaptation logic for sce_{TS1} and sce_{REQ} is depicted in Figure 4 (in terms of a BPEL process).

Conditions in BPEL processes are encoded through XPath expressions. The function *expACK4ReceivedMsg()* can be expressed using XPath. However, another two functions *isInterest()* and *chkControlDataDependent()* cannot since *isInterest()* relies on the state of protocol scenarios, and *chkControlDataDependent()* depends on control and data dependency relations. They are only the placeholders in a BPEL process (or a GFSA) for an adapter specification, and are not to be checked at a static verification phase.

On the other hand, the functions *isInterest()* and *chkControlDataDependent()* have been taken into consideration when the adaptation logic is generated (for details we refer to the next section). Hence, the fact that these two functions are not to be checked at a static verification phase does not impact the verification result.

The state of an adapter is defined similarly to that of a protocol scenario.

C. Generating an Adaptation Logic

The SPM [2] is general since its adaptation mechanism is independent of any specific interaction. However, an adaptation logic is specific which is coupled with interacting protocol scenarios. The function *genAdaptationLogic*() presented in Algorithm 1 generates such an adaptation logic, where protocol scenarios are encoded in BPEL processes.

Note that we use BPEL processes as input in Algorithm 1, instead of GFSAs, since activity sequencing is simple and clear in BPEL processes through control structures, but is relatively complex and vague in GFSAs [2]. This facilitates the generation of an adaptation logic. In addition, the generated adaptation logic encoded in a BPEL process can be automatically translated into a GFSA by WSAT.

The procedure for generating the adaptation logic is detailed in Algorithm 1. Generally, an adaptation logic is composed of a *Flow* block. A concurrent branch is a revised protocol scenario (see an example in Figure 4 for two protocol scenarios sce_{TS1} and sce_{REQ}). We next explain how a branch is derived from a protocol scenario.

First, the polarity of activities in a protocol scenario (denoted sce_i) is reversed (line 5). The modified protocol scenario (denoted sce_i^{adt}) is integrated into the adaptation logic as a concurrent branch (line 6). Conditions in sce_i are not passed to sce_i^{adt} since they do not need to be re-checked in the adaptation logic.

Receive activities in the adaptation logic are not to be further proceeded since they are to receive messages sent

[2]Note that this procedure for generating the adaptation logic is necessary for the SPM because the adaptation mechanisms of the SPM are general, and hence, an adapter is not needed to be generated at the design time. On the other hand, for other adaptation techniques such as [1], [4], [7], they need to construct a concrete adapter at the design time. Consequently, this adaptation logic generation may not be necessary for them.

Algorithm 1 GENADAPTATIONLOGIC($\{sce\}$)

```
In:  {sce} is a set of protocol scenarios encoded in BPEL processes. Without the loss of generality, we model sce as a structured workflow
Out: adt is an adaptation logic with respect to {sce} encoded in a BPEL process

1:  $adt \longleftarrow$ initialize $adt$ by adding a Flow block $flow_{adt}$ with an empty Flow body (i.e., no branches)
2:  n $\longleftarrow$ $|\{sce\}|$ to get the number of protocol scenarios that are numbered from 1 to n
3:  $T_M \longleftarrow$ generate a set of tuples through the function $isInterest()$, and each tuple is expressed in terms of $(msg, i, j)$ which specifies that a
    message $msg$ is to be produced by $sce_i$ and is to be consumed by $sce_j$. Without the loss of generality, we assume that any $msg$ is to be produced
    by only one protocol scenario, but may be consumed by multiple protocol scenarios
4:  for i = 1 to n do
5:      $sce_i^{adt} \longleftarrow$ reverse the polarity of activities in $sce_i$, i.e., change the send to the receive and vice versa
6:      $flow_{adt} \longleftarrow$ insert $sce_i^{adt}$ as a concurrent branch
7:  end for
8:  for i = 1 to n do
9:      foreach a send activity $act$ in $sce_i^{adt}$ that sends a message $msg$ do
10:         $nd \longleftarrow$ a node in $sce_i^{adt}$ such that there is an arc in $sce_i^{adt}$ which leads $nd$ to $act$. Note that since we model $sce_i$ as a structured
            workflow, there is only one node in $sce_i^{adt}$ that links to $act$
11:         if $msg$ is an $ACK$ then
12:             $act_{agn} \longleftarrow$ generate an Assign activity for producing such an $ACK$
13:             $act_{empty} \longleftarrow$ generate an Empty activity
14:             $switch \longleftarrow$ generate a Switch block, and make $act_{agn}$ as one branch with a condition expACK4ReceivedMsg(), and make $act_{empty}$
                as another branch with a condition true. The $act_{empty}$ branch acts as the default (i.e., <else>) branch of $switch$
15:             $sce_i^{adt} \longleftarrow$ remove the arc in $sce_i^{adt}$ for $nd$ to $act$, and insert $switch$ between $nd$ and $act$
16:             continue
17:         end if
18:         $act_k \longleftarrow$ get the receive activity in $sce_k^{adt}$ that receives $msg$, i.e., a tuple $(msg, k, i)$ exists in $T_M$. Without the loss of generality, we
            assume that only one activity (like $act_k$) in a protocol scenario (like $sce_k$) is to receive a certain message (like $msg$)
19:         if $nd$ is not null and there exists a send or receive activity $act_{fol}$ in $sce_i^{adt}$ such that, $act_{fol}$ immediately follows $act$ (i.e., there is an
            arc in $sce_i^{adt}$ leading from $act$ to $act_{fol}$), and $act_{fol}$ is neither control nor mandatorily data dependent on $act$ (based on the dependency
            relations between the activities of $sce_i$ specified by its control and data dependency graphs) then
20:             $act_{agn} \longleftarrow$ generate an Assign activity for producing a mock-up msg
21:             if $act_k$ is null then
22:                 $sce_i^{adt} \longleftarrow$ remove the arc in $sce_i^{adt}$ for $nd$ to $act$, add an arc for $nd$ to $act_{agn}$, and add a link to specify a control
                    dependency relation for $act_{agn}$ to $act$, and set $chkControlDataDependent()$ as the transition condition on this link
23:             else
24:                 $act_{empty} \longleftarrow$ generate an Empty activity
25:                 $switch \longleftarrow$ generate a Switch block, and make $act_{agn}$ as one branch with a condition chkControlDataDependent(), and
                    make $act_{empty}$ as another branch with a condition true. The $act_{empty}$ branch acts as the default (i.e., <else>) branch of
                    $switch$
26:                 $sce_i^{adt} \longleftarrow$ remove the arc in $sce_i^{adt}$ for $nd$ to $act$, and insert $switch$ between $nd$ and $act$
27:                 if $k \neq i$ then
28:                     foreach an activity $act_{mdp}$ in $sce_i^{adt}$ that is mandatorily data dependent on $act$ do
29:                         $sce_k^{adt}$, $sce_i^{adt} \longleftarrow$ generate a link to specify a control dependency relation for $act_k$ to $act_{mdp}$, and set true
                            as the transition condition on this link
30:                     end for
31:                 end if
32:             end if
33:             continue
34:         end if
35:         if $act_k$ is not null and $k \neq i$ then
36:             $sce_k^{adt}$, $sce_i^{adt} \longleftarrow$ generate a link to specify a control dependency relation for $act_k$ to $act$, and set isInterest() as the transition
                condition on this link
37:         end if
38:     end for
39: end for
```

by protocol scenarios. Since *receive* activities in sce_i^{adt} are tightly coupled with *send* activities in sce_i, a message sent by sce_i is ready to be received by sce_i^{adt} immediately.

On the other hand, the situation for *send* activities is different. *Send* activities in sce_i^{adt} are coupled with *receive* activities in sce_i. This means that a message sent by sce_i^{adt} is ready to be received by sce_i, but this message exchange requires that the message to be sent has been received by the adaptation logic. However, this assumption may not be satisfiable in some cases. As such, the SPM needs to mediate missing messages or deadlocks. Consequently, line 10-37 handles each *send* activity act in sce_i^{adt} as follows:

- If a message to be sent is an *ACK* for business data that is to be received previously, an *Assign* activity act_{agn} generates such an *ACK* (line 11-17).
  On the other hand, this *ACK* may have been received already, and hence, an *Empty* activity act_{empty} is used for representing this situation. act_{agn} and act_{empty} are assembled into a *Switch* block (denoted $switch$) with conditions $expACK4ReceivedMsg()$ and $true$

respectively. The block $switch$ is integrated into sce_i^{adt} immediately before act.

- If a message msg to be sent by act is concrete business data, the activity act_k in a protocol scenario sce_k^{adt} that receives msg is identified depending on T_M (line 18). T_M is a set of tuples generated through the function $isInterest()$, and each tuple is expressed in terms of (msg, k, i), which specifies that: (1) msg is to be produced by sce_k and (2) msg is to be consumed by sce_i. It is possible that msg is produced by sce_i ($k = i$) or does not exist since no protocol scenario produces it (no tuple in T_M for msg).
  There are two cases for handling act that sends msg as concrete business data, depending on whether act is not the first activity in sce_i^{adt} (guarded by the condition that nd is not null), and whether there exists a send or receive activity act_{fol} in sce_i^{adt} under the condition that: (1) act_{fol} immediately follows act, and (2) act_{fol} is neither control nor mandatorily data dependent on act (line 19).
  Case 1: If act is not the first activity in sce_i^{adt} and act_{fol} exists, a *mock-up* msg is generated by an $Assign$ activity act_{agn} for handling the lack of a msg (line 20). There are two situations for handling this *mock-up* msg as follows:
  - If a msg is not to be produced by any protocol scenario (act_k is null), act_{agn} is inserted into sce_i^{adt} immediately before act, and a link with a transition condition *chkControlDataDependent*() is generated for act_{agn} to act (line 21-22);
  - Or act_k exists, which indicates that such a msg may have been received when act is ready to send a msg. An *Empty* activity act_{empty} is used to represent this situation. act_{agn} and act_{empty} are assembled into a *Switch* block (denoted $switch$) with conditions *chkControlDataDependent*() and $true$ respectively. The block $switch$ is integrated into sce_i^{adt} immediately before act (line 23-26).

  Note that act can be *enabled* using this *mock-up* msg, but any activity (denoted act_{mdp}) in sce_i^{adt} which is mandatorily data dependent on act cannot be *enabled* without msg. If act_k is not in sce_i^{adt}, a link is generated to specify a control dependency relation for act_k to act_{mdp} (line 27-31).
  Case 2: Otherwise, if act_{fol} does not exist, no *mock-up* msg can be generated. If act_k is not in sce_i^{adt}, a link with a transition condition *isInterest*() is generated for specifying a control dependency relation between act_k and act (line 35-37).

The time complexity of Algorithm 1 is $O(n * m^2)$, where *n* is the number of protocol scenarios, and *m* is the maximum number of send and receive activities in a protocol scenario. The reason is that the time complexity of handling a *Send* activity (line 10-37) is $O(m)$, since line 19, as well as line 28-30, needs $O(m)$. Hence in the worst case, there are $(n * m)$ activities to be handled and the time complexity of each activity handling is $O(m)$.

D. Mediated Service Interaction and Conversation

After defining a protocol scenario and an adapter and generating the adaptation logic for interacting protocol scenarios, in this section, we formalize the mediated service interaction in Definition 4 leveraging on its schema presented by Definition 3. Thereafter, the conversation is specified in Definition 5 based on the notion of a global configuration. Leveraging on these formalization, we are ready to formally verify a mediated service interaction.

1) Mediated Service Interaction:

Definition 3 (Mediated Service Interaction Schema). *A mediated service interaction schema is a tuple (P, M, Σ). P is a set $\{sce_1, \ldots, sce_n, adt\}$ where sce_i ($i \in [1, n]$) is a protocol scenario and adt is an adapter.*

The alphabet M is a finite set of message classes.

A protocol scenario is $sce_i = (M_i^{in}, M_i^{out})$ such that $M_i^{in} \cap M_i^{out} = \emptyset$ (where $\emptyset$ means an empty set), and $M_i = M_i^{in} \cup M_i^{out}$ is the alphabet of sce_i.

An adapter is $adt = (M_{adt}^{in}, M_{adt}^{out})$. $M_{adt}^{in} = \bigcup_{i\in[1,n]} M_i^{out}$, and $M_{adt}^{out} = M_{gen} \cup \bigcup_{i\in[1,n]} M_i^{in}$ where M_{gen} is a finite set of generated ACK *or* mock-up *messages.*

For any two protocol scenarios $sce_i = (M_i^{in}, M_i^{out})$ and $sce_j = (M_j^{in}, M_j^{out})$, $M_i^{out} \cap M_j^{out} = \emptyset$, but $M_i^{in} \cap M_j^{in} \neq \emptyset$ is allowed. The alphabet $M = M_{gen} \cup \bigcup_{i\in[1,n]}(M_i^{in} \cup M_i^{out})$, since $\bigcup_{i\in[1,n]} M_i^{in} \neq \bigcup_{i\in[1,n]} M_i^{out}$ is allowed.

Σ is is a finite set of message with respect to M.

Different from the GFSA composition schema defined in [2], Definition 3 allows the set of send message classes and receive message classes not to be the same, and one send message to be received by multiple peer protocol scenarios.

Definition 4 (Mediated Service Interaction). *A mediated service interaction is a tuple $medI = \langle(P, M, \Sigma), sce_1, \ldots, sce_n, adt\rangle$ where (P, M, Σ) is the corresponding mediated service interaction schema. $|P| = (n + 1)$. sce_i ($i \in$ [1, n]) is the implementation of the ith protocol scenario, and adt is the adaptation logic of an adapter with respect to these protocol scenarios $sce_1, \ldots, sce_n$.*

2) Conversation: We next present the notion of a conversation with respect to a certain $medI = \langle(P, M, \Sigma)$, $sce_1, \ldots, sce_n, adt\rangle$. A global configuration γ of $medI$ is a $(n + 1)$-tuple in terms of the form $(s^1, \ldots, s^n, s^{adt})$. s^i denotes the state of the i^{th} protocol scenario sce_i, and s^{adt} denotes the state of the adapter adt. $\gamma_0 = (s_0^1, \ldots, s_0^n, s_0^{adt})$ is the initial global configuration, where s_0^i and s_0^{adt} are the initial states of sce_i and adt respectively. $\gamma_f = (s_f^1, \ldots, s_f^n, s_f^{adt})$ is the final global configuration, where s_f^i and s_f^{adt} are the final states of sce_i and adt respectively. Note that $medI$

is synchronizable (see Theorem 1), so that unbounded input queues for storing incoming messages are unnecessary.

For global configurations $\gamma_i = (s_i^1, \ldots, s_i^n, s_i^{adt})$ and $\gamma_j = (s_j^1, \ldots, s_j^n, s_j^{adt})$, we specify that, γ_i derives γ_j (denoted $\gamma_i \longrightarrow \gamma_j$), if one of the following six conditions holds:

- C1: sce_k sends a message m, i.e., $\gamma_i \xrightarrow{!m} \gamma_j$, such that: 1) $(s_i^k, !m, s_j^k) \in \delta^k$, where δ^k is the transition set of sce_k, and 2) $s_i^{adt} = s_j^{adt}$, and $\forall\ l \neq k$: $s_i^l = s_j^l$
- C2: sce_k receives a message m, i.e., $\gamma_i \xrightarrow{?m} \gamma_j$, such that: 1) $(s_i^k, ?m, s_j^k) \in \delta^k$, and 2) $s_i^{adt} = s_j^{adt}$, and $\forall\ l \neq k$: $s_i^l = s_j^l$
- C3: sce_k executes a ϵ-transition, i.e., $\gamma_i \xrightarrow{\epsilon} \gamma_j$, such that: 1) $(s_i^k, \epsilon, s_j^k) \in \delta^k$, and 2) $s_i^{adt} = s_j^{adt}$, and $\forall\ l \neq k$: $s_i^l = s_j^l$
- C4: adt sends a message m, i.e., $\gamma_i \xrightarrow{!m} \gamma_j$, such that: 1) $(s_i^{adt}, !m, s_j^{adt}) \in \delta^{adt}$, where δ^{adt} is the transition set of adt, and 2) $\forall\ l$: $s_i^l = s_j^l$
- C5: adt receives a message m, i.e., $\gamma_i \xrightarrow{?m} \gamma_j$, such that: 1) $(s_i^{adt}, ?m, s_j^{adt}) \in \delta^{adt}$, and 2) $\forall\ l$: $s_i^l = s_j^l$
- C6: adt executes a ϵ-transition, i.e., $\gamma_i \xrightarrow{\epsilon} \gamma_j$, such that: 1) $(s_i^{adt}, \epsilon, s_j^{adt}) \in \delta^{adt}$, and 2) $\forall\ l$: $s_i^l = s_j^l$

Based on the above derivation operations, Definition 5 defines the conversation of a mediated service interaction.

Definition 5 (Conversation). *A conversation of a mediated service interaction is a sequence of global configurations: $\gamma = \gamma_0 \rightarrow \gamma_1 \rightarrow \cdots \rightarrow \gamma_k \rightarrow \gamma_f$, where γ_0 and γ_f are the initial and final global configurations respectively. For any global configuration γ_i ($i \in [0, k]$), γ_i derives $\gamma_{(i+1)}$, i.e., $\gamma_i \xrightarrow{!m|?m|\epsilon} \gamma_{(i+1)}$. Note that $\gamma_{(k+1)}$ corresponds to γ_f.*

Different from the conversation defined in [2] that considers send sequences only, Definition 5 considers receive and ϵ-transitions as well. The reason is that the context of [2] is a direct service interaction, as such, each sent message is to be received by only one peer service protocol, and ϵ-transitions have no impact to the interaction. However, in a mediated service interaction, a sent message is possibly received by multiple peer protocol scenarios, and new *ACK* and *mock-up* messages are possibly being generated by ϵ-transitions in an adapter for reconciling deadlocks. These generated messages are consistent with business requirements.

V. Verifying Mediated Service Interactions

This section specifies the properties of the conversations in terms of LTL Formulae which can be verified through SPIN. A basic principle of SPIN, as well as model checking tools in general, is to trace through all relevant states with respect to a certain property, where the order of these states indicates all possible execution paths. If the answer is *yes*, then the property is satisfied. Otherwise, a counterexample is given which is helpful for debugging purposes.

There are two kinds of general properties to be considered for the verification: *reachability* and *liveness*. A *reachability* property states the fact that a particular situation can *sometimes* be reachable, whereas a *liveness* property prescribes that a situation can ultimately occur under certain conditions. Hence, a *liveness* property formulates a much stronger condition than a *reachability* property, since it requires that a particular situation is *always* reachable. An example of *liveness* is the *termination* property of a mediated service interaction as reflected by Formula 1:

$$\mathbf{G}(\gamma_0 \longrightarrow \mathbf{F}\gamma_f) \tag{1}$$

where $\mathbf{G}$ and $\mathbf{F}$ are temporal operators which mean *globally* and *eventually* respectively.

The satisfaction of Formula 1 means that protocol scenarios can be mediated, under the condition that, the conjunction of all transition guards in protocol scenarios (called a *must-be-held* condition in the following) is satisfiable according to exchanged message instances. A *must-be-held* condition includes *Switch* conditions that lead service protocols to follow these protocol scenarios, and other conditions that guard transitions in these protocol scenarios.

Indeed, a runtime behavior of a service protocol depends on the evaluation of its *Switch* and other conditions, whilst their evaluation relies on exchanged message instances. Considering two protocol scenarios sce_{TS1} and sce_{REQ}, suppose that the interaction between sce_{TS1} and sce_{REQ} is an expected interaction to the client, and suppose this interaction satisfies Formula 1 under a *must-be-held* condition $Cond = Cd_2 \bigwedge Cd_3 \bigwedge Cd_5 \bigwedge Cd_8$. Then TS_1 and REQ are adaptable with a prerequisite that exchanged message instances can lead TS_1 and REQ to follow sce_{TS1} and sce_{REQ} (i.e., their *Switch* conditions in $Cond$ are satisfied), and they can satisfy other conditions in $Cond$.

Formula 1 is usually being violated if a mediated service interaction is modeled at a service protocol level. The reason is that, service protocols usually can interact in some, but not all, situations [1]. In order to perform a verification at a service protocol level, Formula 1 is to be changed to a *reachability* property as shown in Formula 2:

$$\mathbf{F}(\gamma_0^{sp} \longrightarrow \mathbf{F}\gamma_f^{sp}) \tag{2}$$

where γ_0^{sp} is the initial global configuration, and γ_f^{sp} is one of the final global configurations, of a mediated service interaction that is modeled at a service protocol level. γ_0^{sp} and γ_f^{sp} can be specified in a similar manner like γ_0 and γ_f.

The satisfaction of Formula 2 indicates that service protocols can be mediated in some cases. However, this result does not provide much valuable information, since it does not give an indication on which set of protocol scenarios in these service protocols can be mediated with which conditions, as well as which set cannot. Often a service protocol has multiple protocol scenarios. The satisfiability of

Formula 2 is inadequate to inform the client about whether or not an expected interaction can possibly succeed.

On the other hand, the satisfiability of Formula 1 indicates whether a particular set of protocol scenarios can be mediated under a certain *must-be-held* condition. Given the verification result for all expected interactions (i.e., for all expected combinatorial protocol scenarios), the client is informed about an overall knowledge of expected interactions, i.e., whether or not an expected interaction can interact properly under which condition. This knowledge is an important criterion to the client to identify if an expected service interaction is appropriate, and hence, to select the most suitable service provider among functionally equivalent candidates according to her requirements.

VI. Related Work and Conclusion

A significant work is presented in [2], which studies asynchronous BPEL processes interactions following bottom-up and top-down manners (see Section II-B). Based on which, our work is conducted, since we observe that a mediated service interaction is synchronizable. However, this work conducts a verification at a service protocol level which usually cannot provide a precise evaluation (i.e., not client-expected interactions oriented, and not providing the conditions that determine when the interactions are possible) as mentioned in Section V. This approach allows performing synchronizability analysis, which is to verify whether a synchronous composition may be applied for further analysis without loosing behaviors. The synchronous communication model covers only a part of service composition scenarios of practical relevance [3]. This observation leads the work [3], which develops a parametric model for describing service compositions, and captures a hierarchy of communication models ranging from synchronous communication models to asynchronous communication models with complex buffer structures. Consequently, a least general but adequate model can be built for a certain composition scenario.

Besides, in [17], a service protocol interaction is described in terms of BPMN, which is then being verified through a process-algebraic approach. In [18], service protocols are modeled by means of petri nets, and hence, the properties such as *usability*, *compatibility*, and *similarity* can be checked using existing tools.

In summarisation, this paper has two major contributions:

1) We observe that a mediated service interaction is synchronizable. This characteristic is important, and is also a fundamental prerequisite of our approach, because non-synchronizable service compositions cause the undecidability of LTL verification [2], and
2) Our technique provides a precise evaluation about expected (instead of all) interactions (i.e., which expected service interaction is adaptable with which condition, and which is not).

The result of our work is an important criterion to the client for identifying and thus selecting the most suitable service provider from functionally equivalent candidates according to her specific business requirements.

Acknowledgment

This work has been funded by Science Foundation Ireland under Grant No. SFI/08/CE/I1380 (DERI Lion-2).

References

[1] W. Tan, Y. Fan, and M. Zhou, "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*, vol. 6, no. 1, pp. 94–106, 2009.

[2] X. Fu, "Formal specification and verification of asynchronously communicating web services," Ph.D. dissertation, University of California at Santa Barbara, 2004.

[3] R. Kazhamiakin, M. Pistore, and L. Santuari, "Analysis of communication models in web service compositions," in *Proceedings of WWW*, 2006.

[4] H. R. M. Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati, "Semi-automated adaptation of service interactions," in *Proceedings of WWW*, 2007.

[5] Z. Zhou, S. Bhiri, W. Gaaloul, and M. Hauswirth, "Developing process mediator for supporting mediated web service interactions," in *Proceedings of ECOWS*, 2008.

[6] M. Dumas, B. Benatallah, and H. R. M. Nezhad, "Web service protocols: Compatibility and adaptation," *IEEE Data Engineering Bulletin*, vol. 31, no. 3, pp. 40–44, 2008.

[7] D. M. Yellin and R. E. Strom, "Protocol specifications and component adaptors," *ACM Trans. on Programming Languages and Systems* , vol. 19, no. 2, pp. 292–333, 1997.

[8] Z. Zhou, S. Bhiri, and M. Hauswirth, "Control and data dependencies in business processes based on semantic business activities," in *Proceedings of iiWAS*, 2008.

[9] J. Nitzsche, T. van Lessen, D. Karastoyanova, and F. Leyman, "$bpel^{light}$," in *Proceedings of BPM*, 2007.

[10] Q. Wu, C. Pu, A. Sahai, and R. Barga, "Categorization and optimization of synchronization dependencies in business processes," in *Proceedings of ICDE*, 2007.

[11] B. Benatallah, F. Casati, D. Grigori, H. R. M. Nezhad, and F. Toumani, "Developing adapters for web services integration," in *Proceedings of CAiSE*, 2005.

[12] H. R. M. Nezhad, R. Saint-Paul, B. Benatallah, and F. Casati, "Deriving protocol models from imperfect service conversation logs," *IEEE Trans. on Knowledge and Data Engineering*, vol. 20, no. 12, pp. 1683–1698, 2008.

[13] X. Fu, T. Bultan, and J. Su, "Synchronizability of conversations among web services," *IEEE Trans. on Software Engineering*, vol. 31, no. 12, pp. 1042–1055, 2005.

[14] M. D. Backer, M. Snoeck, G. Monsieur, W. Lemahieu, and G. Dedene, "A scenario-based verification technique to assess the compatibility of collaborative business processes," *Data and Knowledge Engineering*, vol. 68, no. 6, pp. 531–551, 2009.

[15] W. M. P. van der Aalst, T. Weijters, and L. Maruster, "Workflow mining: Discovering process models from event logs," *IEEE Trans. on Knowledge and Data Engineering.*, vol. 16, no. 9, pp. 1128–1142, 2004.

[16] Z. Zhou and S. Bhiri, "Analyzing mediated service interactions," Available at http://www.deri.ie/fileadmin/documents/DERI-TR-2009-02-16.pdf, 2009.

[17] P. Y. Wong and J. Gibbons, "Verifying business process compatibility," in *Proceedings of QSIC*, 2008.

[18] A. Martens, "Analyzing web service based business processes," in *Proceedings of FASE*, 2005.