

# Space Based Process Mediator

Zhangbing Zhou and Sami Bhiri

Digital Enterprise Research Institute, National University of Ireland at Galway, Ireland  
{firstname.lastname}@deri.org

## Abstract

*Web service interactions lie in the core of Service Oriented Architecture. Due to the inherent autonomy, heterogeneity and continuous evolution of Web services, mediators are often needed to support service interactions to overcome possible mismatches existing among Web service based business processes. This paper introduces a space based process mediator which considers both control-flow and data-flow, presents possible mismatch patterns, and discusses how they can be automatically mediated. Our process mediator can address not only all mismatch patterns prescribed by existing process mediators, but also new mismatch patterns related to data-flow. In addition, our process mediator provides a uniform mechanism to perform runtime mediation without the need of a design-time work, and greatly facilitates Web service interactions.*

## 1. Introduction

A main promise of Web service architecture<sup>1</sup> is to enable and support seamless service interactions, which can be described as a flow of messages [11], of diverse business processes (BPs) encapsulated as Web services [24]. Standards, such as SOAP<sup>2</sup>, WSDL<sup>3</sup>, and BPEL<sup>4</sup>, make service interactions easier. However, because of the inherent *autonomy, heterogeneity and continuous evolution* of Web services, messages are often different in *format and granularity*, and BPs are often diverse in *activity sequence and their message format*. In such a context, it is difficult, if not impossible, to find two BPs that are completely compatible [4], and their interactions are often carried out with the help of data and/or process mediators [10] called mediated service interactions [26].

In recent years, much research has been conducted for

<sup>1</sup><http://www.w3.org/TR/ws-arch/>.

<sup>2</sup><http://www.w3.org/TR/2000/NOTE-SOAP-20000508/>.

<sup>3</sup><http://www.w3.org/TR/wsdl>.

<sup>4</sup><http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf>. BPEL is now the *de facto* standard to model Web service based business processes.

process mediation, e.g. [2] [3] [5] [7] [8] [12] [13] [18] [19] [21] [25] [27]. However, they mainly focus on control-flow but largely ignore data-flow. This ignorance suggests that they are limited to identify and resolve mismatches related to data-flow, but simply regard these kinds of mismatches as unresolvable [10]. In addition, most work needs a design-time mediation, i.e. specific adapters are developed for specific mismatch patterns [3] [7] respectively. They lack a uniform mechanism to possibly resolve all kinds of resolvable mismatches [10] for execution purposes.

In this paper, we propose a space based process mediator which considers both control-flow and data-flow. We firstly introduce the architecture of our process mediator, present mechanism and possible mismatch patterns, and then discuss how these mismatch patterns can be automatically mediated in the space based process mediation architecture.

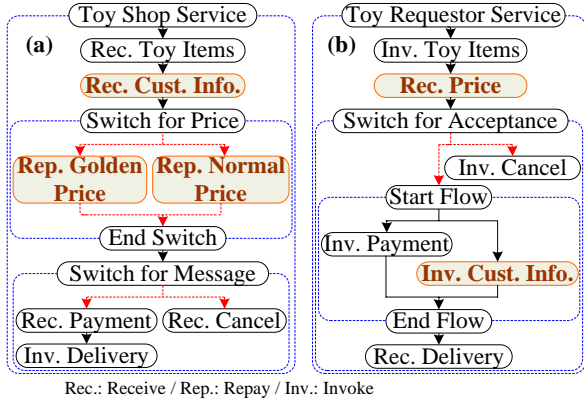
Our approach does not deal with the heterogeneity at a data level since it is out of the focus of this paper and also much research has been conducted at this aspect [17].

To the best of our knowledge, our work is the first study of process mediation which considers both control-flow and data-flow, and can support not only all mismatch patterns prescribed by existing process mediation approaches [3] [7], but also new mismatch patterns related to data-flow. In addition, our work provides a uniform mechanism to deal with all resolvable mismatches at runtime, and thus a design-time mediation is not needed.

## A Motivating Example

Fig. 1-a and 1-b illustrate two BPEL processes for a Toy Shop (*TS*) and a Toy Requestor (*TR*) services which would like to interact to achieve a goal: *buying toys*. A definition for our business process model is presented in Section 2.

*TS* may apply a discount on the “*Price*” depending on the customer profile. That is why it requests “*Cust. Info.*” before sending the corresponding “*Price*”. On the contrary, due to the privacy concern, *TR* prefers to provide “*Cust. Info.*” only if he/she is convinced by the “*Price*” and is committed to buy. This example shows an interaction mismatch where *TS* is requesting “*Cust. Info.*” before sending the “*Price*” while *TR* is expecting the “*Price*” before deciding upon continuing and sending “*Cust. Info.*” or not.



**Figure 1. BPEL business processes for a toy shop (a) and a toy requestor (b)**

*TS* and *TR* are incompatible according to current approaches for compatibility analysis, e.g. [4] [15], because there is no a sequence of messages that can lead them from their *Start* nodes to one of their final nodes. In addition, the mismatch between *TS* and *TR* is regarded as unresolvable according to current approaches for process mediation [3] [7] and thus no interaction is assumed as possible.

To address this problem, this paper proposes a space based process mediator which is able to resolve such kind of mismatches. Shortcomings of current approaches for process mediation follow from their equivalence between sequencing constraints and control dependency relations. Indeed, as shown in [23], different kinds of dependency relations may exist between BPEL process activities which are obfuscated by a BPEL process description. Current process mediation approaches consider each sequencing constraint within a BPEL process as a control dependency which is not always the case. For instance, the sequencing constraint between “*Rec. Cust. Info.*” and “*Rep. Normal Price*” is due to a data dependency rather than to a control dependency. [16] discussed how data dependency relations can be explicitly extracted from BPEL processes.

To overcome current limitations, our approach considers data dependencies in addition to control dependencies to go around interaction mismatches considered unresolvable so far. Back to our example, “*Cust. Info.*” is used to decide if a discount price applies or not. However, a “*Normal Price*” always applies by default. On the contrary, “*Cust. Info.*” is mandatory for “*Delivery*”. Knowing such information, a process mediator can send a *mock-up* “*Cust. Info.*” message to *TS* which will then execute the following activity and provide “*Normal Price*”. The process mediator will update “*Cust. Info.*” for *TS*, in order to achieve “*Inv. Delivery*”, whenever “*Cust. Info.*” is provided by *TR*.

*This paper is structured as follows.* Section 2 presents

a definition of our business process model and introduces control and data dependencies, which are used in our process mediator. Section 3 presents our space based process mediator including architecture and mechanism. In addition, using our motivating example, we explain how mismatches are resolved at runtime. Section 4 presents mismatch patterns and discusses how they fit with the architecture. Section 5, 6 and 7 present the prototype, discuss the related work, and draw a conclusion respectively.

## 2 Business Process, Control and Data Dependencies

### 2.1 Business Process

Below we give a definition for a BP where messages and guard functions are the first-class citizens.

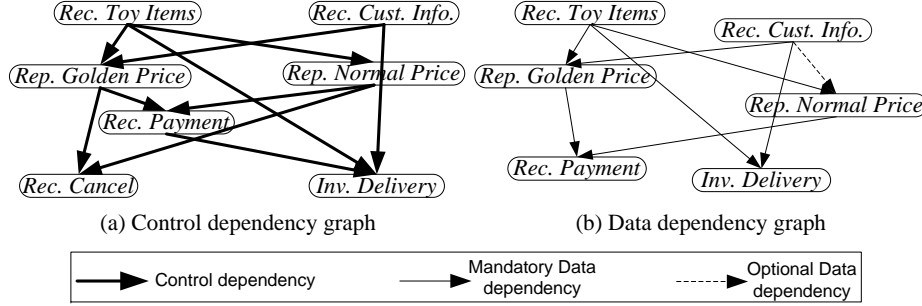
**Definition 1 (Business Process).** A business process  $p$  is a tuple  $p: (MSG, ACT, CNT, GRD, ARC)$ , where  $MSG = \{msg\}$  is a finite set of messages,  $ACT = \{act\}$  is a finite set of activities which send or receive these messages,  $CNT = \{Start, Failure, End, While, Switch, Flow\}$  are control elements,  $GRD = \{grd\}$  is a finite set of guard functions related to these control elements, and  $ARC = \{arc\}$  is a finite set of arcs, which connect activities and control elements to specify sequencing constraints [23] among activities.

Both activities and control elements are the nodes in a business process. An activity sends or receives a message, a message contains a data, and a business process is modeled as a structured workflow [14]. We define the function  $data(act)$  that retrieves the data related to the message which is used by the activity  $act$ .

An activity has a semantic description that defines its precondition, effect, input and output. In fact, an activity has two kinds of purposes. Firstly, it can consume and produce messages. The message consumption and production are described by the input and the output of the activity. Secondly, the execution of an activity can produce a change on the state of the process instance. This state transition is described by the precondition and effect of the activity [1].

A node has four possible states during its life cycle namely *Initial*, *Enabled*, *Completed* and *Failed*. Initially when an instance is created all its nodes are in the state *Initial*. A node becomes *Enabled* if its precondition is satisfied and its input is available. It moves to the state *Completed* once completes successfully. Otherwise, (that means when it fails) it moves to the state *Failed*. It is possible that a node change its state from *Completed* to *Enabled* if it is in a *While* block.

We define the concept *local configuration* of a running instance at a certain time as the set of states of its nodes instances at that time. This concept allows capturing the state of a process instance at a certain time.



**Figure 2. Control and data dependency graphs for *Toy Shop Process***

**Definition 2 (Local Configuration).** Let  $p$  a business process. The local configuration,  $lc$ , of an instance of  $p$  at a given time  $t$  is  $lc = \{state_i \mid state_i \text{ is the state of the node (instance) } i \text{ at } t\}$ .

When instantiated a process instance has an initial local configuration where all its nodes are in the state “Initial”. A process instance reaches a final local configuration when one of its final nodes (“Failure” or “End”) reaches the state “Completed”.

## 2.2 Control and Data Dependencies

As discussed in [23], different kinds of dependencies, such as *control*, *data*, *service* and *cooperation* dependencies, exist between activities of a business process. These dependencies are often obfuscated by a business process model that defines all possible activities execution sequencing. A sequencing constraint in a process model may result from one or several kinds of dependencies [23]. In our approach, we consider two kinds of dependencies namely control<sup>5</sup> and data dependencies.

Data dependencies are related to the data required and produced by the activities. An activity  $B$  is data dependent on an activity  $A$  if  $A$  produces some data required by  $B$ .

Control dependencies are rather related to the preconditions and effects of the activities. An activity  $B$  is control dependent on an activity  $A$  if the completeness of  $A$  (marked by its effect) is a necessary condition for the execution of  $B$  (guarded by its precondition).

For instance, “*Inv. Delivery*” is control dependent on “*Rec. Payment*” and data dependent on “*Rec. Cust. Info.*” (see Fig. 2).

[23] claims that the different kinds of dependencies between activities can be extracted from design documents. [16] presents an approach to extract data dependencies from BPEL process models. In our approach we extract data and control dependencies from the semantic description (in

<sup>5</sup>Note that the control dependency we are considering is different from the one defined in [23]

terms of precondition, effect, input and output) of the process activities.

Hence, we distinguish control and data dependency graphs of a business process that specify a finite set of *asymmetric*, *irreflexive* and *transitive* relations among activities. Fig. 2 illustrates the control and data dependency graphs of *TS*. We define the function  $depCN(act1, act2)$  that checks if  $act1$  is control dependent on  $act2$  or not.

We record data dependencies as mandatory or optional.  $B$  is optional data dependent on  $A$  if the data required by  $B$  from  $A$  is optional for  $B$  and  $B$  can be executed without it. For instance “*Rep. Normal Price*” is optional data dependent on “*Rec. Cust. Info.*”. However “*Inv. Delivery*” is mandatory data dependent on “*Rec. Cust. Info.*”. If there is a data dependency path between  $A$  and  $B$  where there is one optional data dependency among this path then  $B$  is considered optional data dependent on  $A$ . We define the function  $depDT(act1, act2)$  that returns  $M$  (for *Mandatory*) if  $act2$  is mandatory data dependent on  $act1$ ,  $O$  (for *Optional*) if  $act2$  is optional data dependent on  $act1$ , or  $N$  (for *None*) if  $act2$  is not data dependent on  $act1$ .

## 3 Space Based Process Mediator

We firstly present the architecture of our space based process mediator, and introduce utility functions that are used by the process mediator. Then we present mediation mechanisms to explain how our process mediator works. Finally, using an interaction between *TS* and *TR* as discussed in the motivating example, we show how mismatches are resolved by our process mediator at runtime.

### 3.1 Architecture Overview

To support service interactions in the dynamic and heterogeneous environment, we propose a space based process mediator (SPM). SPM does not deal with compatibility analysis which is out of the scope of this paper. It rather ensures successful interactions among a set of (fully or at least

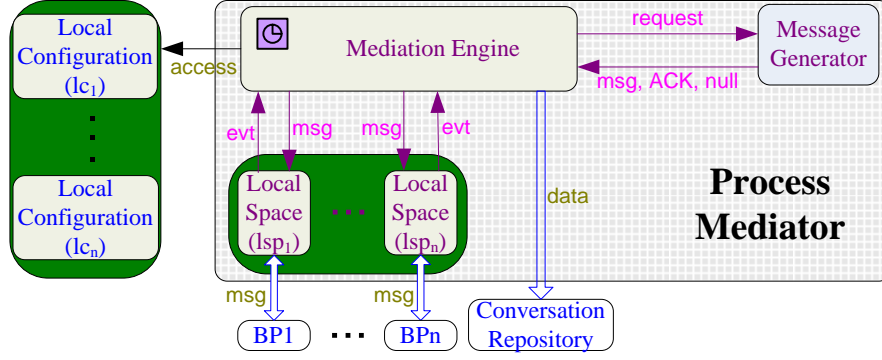


Figure 3. Space based process mediator architecture

partially) compatible BPs [4] [26]. In particular, it provides mechanisms to automatically mediate potential mismatches among them.

Fig. 3 depicts the overall architecture of our SPM. Each BP has a local space which saves its incoming and outgoing messages. BPs do not communicate with each other in a direct manner. When a BP deposits a to-be-sent message into its local space, an event is generated and the Mediation Engine is notified. Based on the current configuration of the interaction, and using a set of communication and mediation rules, the Mediation Engine either dispatches the corresponding message or drop it out as explained below. The Mediation Engine can itself generate a sanity check event when no messages have been exchanged during a certain period of time.

The messages and their sequences exchanged so far are recorded in the conversation repository, which are used for recovery purposes if the interaction fails, and BP consistency analysis later on.

### 3.2 Utility Functions

Before detailing how the SPM operates, we need to introduce some utility functions used during BP interaction. Recall that SPM have knowledge about BP specifications and local configurations of their running instances that describe their states during an interaction. Having the process model of a BP  $p$  and the local configuration  $lc$  of a running instance, the SPM can check (1) if this instance is expecting a given message  $msg$ , (2) if it is sending a message, (3) if it is expecting an acknowledgement for a sent message  $msg$ , or (4) if it is expecting a business data and if this data is mandatory or optional for the following activities. Thereafter, we consider the following functions.

- $isInterest(p, lc, msg)$  is a boolean function that checks if an instance of  $p$  having  $lc$  as current state is expecting  $msg$  (see Algorithm 1). Based on the process model and the current local configuration, this function

compares  $msg$  to the messages expected by currently or eventually Enabled receiving activities.

---

#### Algorithm 1: $isInterest(p, lc, msg)$

---

**Input** : -  $p$ : the process model,  
-  $lc$ : the current local configuration of the running instance,  
-  $msg$ : the message to check if the running instance of  $p$  is expecting or not.  
**Output**: -  $true$  or  $false$

```

1 begin
2   for each receiving activity  $act \in ACT$  which has a
   state "Initial" or "Enabled" do
3     if  $msg = data(act)$  then
4       return true
5   return false
6 end

```

---

- $isSendAction(p, lc)$  is a boolean function that checks if one of the currently *Enabled* activities in the running instance of  $p$  (having  $lc$ ) is sending a message.
- $expACK4ReceivedMsg(p, lc)$  checks if one of the currently *Enabled* activities of the running instance of  $p$  (having  $lc$ ) is expecting an acknowledgement *ACK* for a sent message  $msg$ . If yes,  $msg$  is returned. Otherwise,  $null$  is returned.
- $chkOptionalDataDependent(p, lc)$  returns a message expected by the running instance of  $p$  which is optional for one of the immediately following activities. This function checks for each of its currently *Enabled* receiving activities  $act_{enb}$  (line 2) if there is an optional data dependency between  $act_{enb}$  and one of its following activities  $act_{fol}$  (line 3). An optional data dependency exists between  $act_{enb}$  and  $act_{fol}$  if there is neither mandatory data dependency nor a control depen-

---

**Algorithm 2:** *chkOptionalDataDependent(p, lc)*

---

**Input** : -  $p$ : the process model,  
-  $lc$ : the current local configuration of the running instance.  
**Output**: - an optional message or *null*

```
1 begin
2   for each receiving activity  $act_{enb} \in ACT$ 
   currently "Enabled" do
3     for each activity  $act_{fol}$  immediately following
        $act_{enb}$  do
4       if  $depDT(data(act_{enb}), data(act_{fol})) \neq M$ 
         and  $depCN(act_{enb}, act_{fol}) = N$  then
5         return  $data(act_{enb})$ 
6   return null
7 end
```

---

dependency between them (line 4). If an optional dependency exists between  $act_{enb}$  and  $act_{fol}$ , the message expected by  $act_{enb}$  is returned (line 5). Otherwise, *null* is returned (line 6).

The time complexity of this algorithm is  $O(m)$ , where  $m$  is the number of the nodes in  $p$ , because the algorithm visits all nodes in  $p$  once in the worst case.

### 3.3 Mediation Mechanism

Here we come to explain the mediation mechanisms applied by the Mediation Engine whenever an event occurs. There are two kinds of events that the Mediation Engine needs to deal with: (1) *a message is produced by a BP*, and (2) *a request for the sanity check*, which indicates that no messages have been exchanged among BPs for a period of time, which is longer than a given threshold.

**Event: a message is produced by a BP.** Algorithm 3 shows how the Mediation Engine reacts to the fact that a BP  $p_k$  has generated a message  $msg$ . The Mediation Engine writes  $msg$  into the local space of each BP  $p_j$  which is expecting  $msg$  (line 2-4).  $msg$  is dropped out if no BP is expecting it (line 5). It is possible that one message is consumed by multiple BPs. The time complexity of this algorithm is  $O(n \times m)$ , where  $n$  is the number of BPs, and  $m$  is the maximum number of nodes in BPs, because the algorithm visits all nodes in each BP once at most.

**Event: a request for the sanity check.** A sanity check event is generated by a timer when no messages have been exchanged for a certain period of time. Algorithm 4 shows how the Mediation Engine reacts to such kind of events. There are four reasons for not exchanging messages for a certain period of time.

---

**Algorithm 3:** *messageConsumption*

---

**in** : -  $msg$ : a message produced by  $p_k$   
**effect**: -  $msg$  is written into the local spaces of interested BPs, or is dropped off

```
1 begin
2   foreach  $j \in \{1, \dots, k-1, k+1, \dots, n\}$  do
3     if  $isInterest(p_j, lc_{cur}^j, msg) = true$  then
4       write  $msg$  in the local space of  $p_j$ 
5   drop off  $msg$ 
6 end
```

---

1. The first case holds when the interaction has completed successfully. An interaction completes successfully if all involved BPs terminates successfully. The Mediation Engine simply counts the number of BPs which has reached one of its final local configurations (line 2-7) and checks if the result is equal to  $n$  (the number of all involved BP) (line 8-10).
2. The second case corresponds to a healthy situation where some BPs are waiting for messages, while some others are still working internally (line 12) and then will generate new messages (line 11-14).
3. The third case corresponds to a blockage where all non-completed BPs are expecting messages. We distinguish between two kinds of messages that a BP may expect: an acknowledgment (*ACK*) for a sent message or a business data message. Note that the Mediation Engine has knowledge about the acknowledgment and the business data structure that BPs may exchange.
  - (a) For each BP which expects an acknowledgment for a sent message (line 17), the Mediation Engine will write the relevant acknowledgment (line 19), which is generated by the Message Generator (line 18), into the local space of  $p$  (line 16-20).
  - (b) For each BP which expects a business data (line 22), the Mediation Engine will write a *mock-up* message, which is generated by the Message Generator (line 23), into the local space of  $p$  (line 24) if such data is neither control dependent nor mandatory data dependent to the following activities (line 21-25). However, this *mock-up* message will be updated by Mediation Engine once a concrete message is produced by the partner.
4. The fourth case corresponds to the failure of the interaction whenever an unresolvable mismatch is encountered (line 26), i.e. the mismatch cannot be resolved using our process mediator. The Mediation Engine is

---

**Algorithm 4:** sanityCheck

---

**in** : - *request*: a sanity checking request  
**effect**: - a *Completion*, *Heathy*, or *failure* message for the Mediation Engine (*ME*), or an *ACK*, *mockUp* message for business processes

```
1 begin
2   cnt ← 0
3   foreach  $j \in \{1, \dots, n\}$  do
4     if  $lc_{cur}^j$  is not a final local configuration then
5       break;
6     else
7       cnt ← cnt + 1
8   if cnt = n then
9     inform ME the completion of this mediated
10    service interaction
11    exit the procedure
12  foreach  $j \in \{1, \dots, n\}$  do
13    if  $isSendAction(p_j, lc_{cur}^j) = true$  then
14      send a Heathy message to ME, which
15      suggests that the interaction is in-progress
16      and still healthy
17      exit the procedure
18  isResolved ← false
19  foreach  $j \in \{1, \dots, n\}$  do
20    if ( $msg \leftarrow expACK4ReceivedMsg(p_j,$ 
21     $lc_{cur}^j)$ )  $\neq null$  then
22      ask MG to generate an acknowledgment,
23       $ackMsg$ , for  $msg$ 
24      write  $ackMsg$  into the local space of  $p_j$ 
25      isResolved ← true
26  foreach  $j \in \{1, \dots, n\}$  do
27    if ( $msg \leftarrow chkControlDataDependent(p_j,$ 
28     $lc_{cur}^j)$ )  $\neq null$  then
29      ask MG to generate a mockUpMsg
30      message for  $msg$ 
31      write mockUpMsg into the local space of
32       $p_j$ 
33      isResolved ← true
34  if  $isResolved = false$  then
35    inform ME the failure of this mediated service
36    interaction
37 end
```

---

notified by a *failure* message, which will inform all BPs of this failure situation (line 27).

The time complexity of this algorithm is  $O(n \times m)$ , where  $n$  is the number of BPs involved in a given interac-

tion, and  $m$  is the maximum number of nodes in these BPs, because the algorithm needs to visit all nodes in each BP once in the worst case.

### 3.4 A Mediated Service Interaction Example between *TS* and *TR*

To illustrate how our SPM works, Fig. 4 depicts an interaction between *TS* and *TR* as discussed in the motivating example. For simplicity, the nodes which do not contribute to this interaction are not presented, and only the messages related to *Enabled* activities are presented in “*Received*” and “*Consumed*” columns, which represent a part of messages saved in the local spaces. “*Received*” means that messages have been received but have not been “*Consumed*”.

- **Step 1.** *TR* produces a message: “*Toy Items*”, which is forwarded by the *Mediation Engine* to *TS*’s local space using the algorithm *messageConsumption* because  $isInterest(TS, lc_{cur}^{TS}, \text{“Toy Items”}) = true$ .
- **Step 2.** *TS* consumes “*Toy Items*” from its local space.
- **Step 3.** *TS* is expecting “*Cust. Info.*” for possibly applying a discount on the price, and *TR* is expecting to consume “*Price*”.

Since no messages are exchanged later on, an *Event*: a request for the sanity check is triggered by the timer and the algorithm *sanityCheck* is executed. According to the data and control dependency graphs (see Fig. 2) there is only an optional data dependency between “*Rec. Cust. Info.*” and the “*Rep. Normal Price*”. That is why a mock-up “*Cust. Info.*” message is generated and sent to the local space of *TS*.

- **Step 4.** *TS* consumes this “*Mock-up Cust. Info.*” from its local space.
- **Step 5.** *TS* produces a message: “*Normal Price*”, which is saved into the local space of *TR*.
- **Step 6.** *TR* consumes “*Normal Price*” from its local space.
- **Step 7.** *TR* produces two messages: “*Payment*” and “*Cust. Info.*”, which are saved into the local space of *TS* like **Step 1**.
- **Step 8.** The process mediator notices that a concrete “*Cust. Info.*” is available, and thus the process mediator replaces “*Mock-up Cust. Info.*” by this concrete “*Cust. Info.*” in *TS*’s local space.
- **Step 9.** *TS* consumes “*Payment*” from its local space.
- **Step 10.** *TS* produces a message: “*Delivery*”, which is saved into the local space of *TR*.

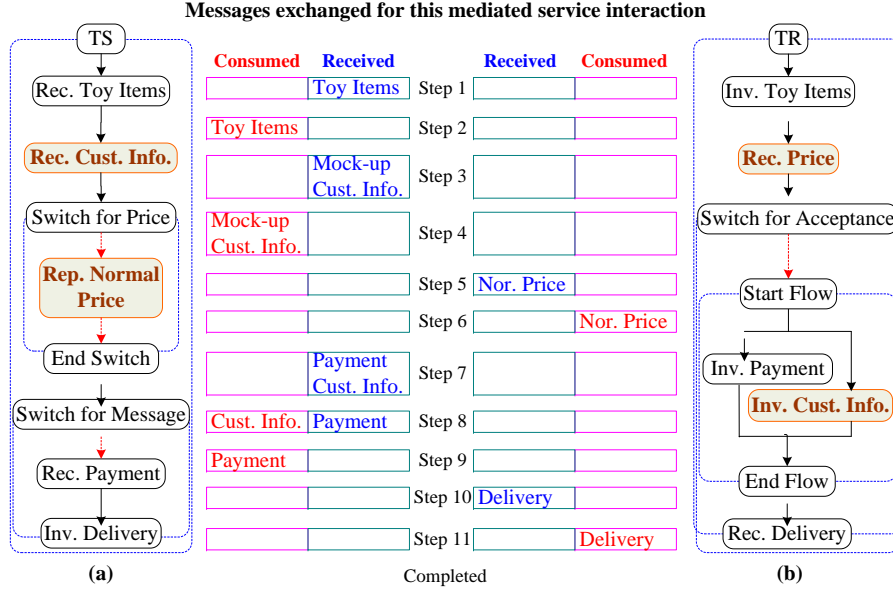


Figure 4. A successful mediated service interaction between *TS* and *TR*

- **Step 11.** *TR* consumes “*Delivery*” from its local space.
- **Completed.** Since no messages are exchanged later on, the algorithm *sanityCheck* is triggered by an *Event: a request for the sanity check*, which gets known that both *TS* and *TR* are in one of their final local configurations. This means that this mediated service interaction is successfully completed. Hence *TS* and *TR* are notified by the *Mediation Engine* with a *Completion* message.

#### 4 Mismatch Patterns and Potential Solutions

This section presents a set of process mismatch patterns and shows how they can be resolved by our SPM. As expressed above, our SPM focuses on process mediation and does not deal with data mediation. However, it can easily be extended to resolve two kinds of data mismatch patterns *Message Split Mismatch* and *Message Merge Mismatch* [27] prescribed by [3] [7]. Our SPM can support not only all process mismatch patterns prescribed by [3] [7] which are related to control-flow, but also new process mismatch patterns related to data-flow.

**Extra Message Mismatch.** This kind of mismatches occurs when a BP produces a message *msg* which is not required by the other BPs, i.e.  $isInterest(otherBP, lc_{cur}, msg) = false$ . According to Algorithm 3, *msg* is dropped out immediately and silently. As discussed in [3] [7], this kind is always resolvable.

**Missing Message Mismatch.** This kind of mismatches occurs when an activity *act* in a BP is expecting to con-

sume a message *msg* which is not produced by another BP. According to SPM mechanisms, this kind of mismatches is resolvable in the following two cases:

The first case is that *msg* is an *ACK* for a sent message and thus the *Message Generator* can generate such an *ACK*.

The second case holds when one of the following activities of *act* is neither control dependent nor mandatorily data dependent on *msg*. This is the case for instance for “*Cust. Info.*” in Fig. 1-a which is optional for determining the price since a “*Normal Price*” is always applied by default. Thus a *mock-up* message is generated by the *Message Generator* and is sent to *act*.

Apart from these two cases, this kind of mismatches is considered unresolvable because the *Message Generator* has no knowledge, and also cannot delegate any BP, to generate any concrete message.

[3] [7] can partially resolve this kind of mismatches by generating an *ACK* for a sent message. However, because they do not consider data dependencies, they regard the missing of a business data as unresolvable.

**Message Order Mismatch: All Sending or All Receiving.** Fig. 5-a illustrates this kind of mismatches. It occurs when a BP is producing a set of messages according to a given order while other BPs expect to consume them in a different order. This mismatch may also occur, as depicted by Fig. 5-b, when a *Flow* block produces a set of messages concurrently, but other BPs consume them following a given sequence. A mismatch may happen if the “actual” message sending order by the *Flow* block does not match the receiving order prescribed by the sequence.

The time decoupling between message production and

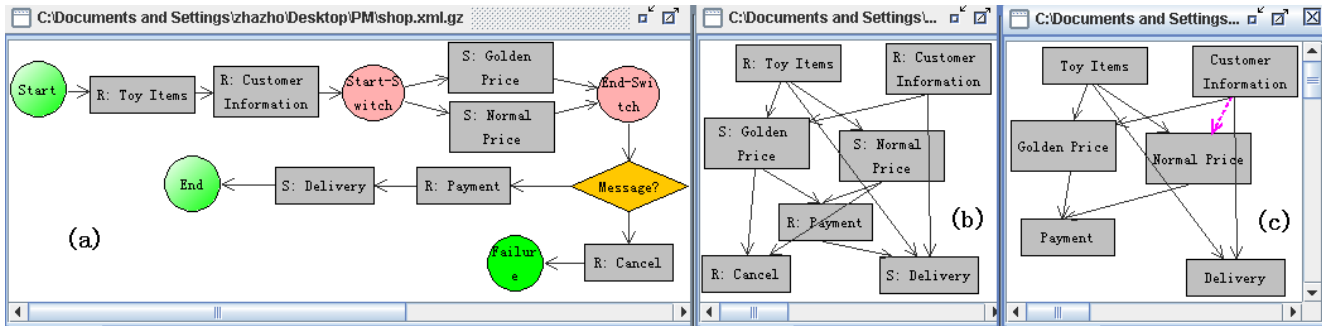


Figure 6. TS: business process, control and data dependency graphs

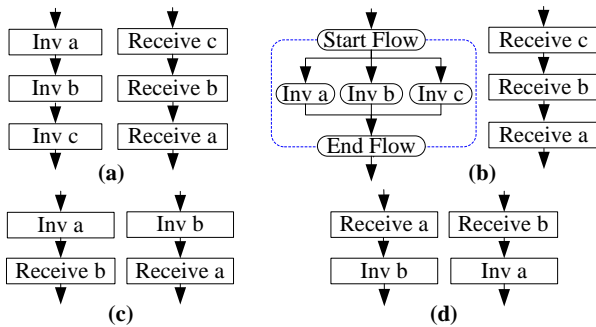


Figure 5. Message order mismatches

message consumption enables to easily resolve this mismatch. Our SPM will dispatch produced messages whenever they are generated, and a requestor BP will consume them once they are available in its local space.

[3] [7] supports the case depicted in Fig. 5-a, but regard the case in Fig. 5-b as unresolvable, because the *Flow* block and *Sequence* are different control-flow structures.

**Message Order Mismatch: Firstly Sending and Then Receiving.** This kind of mismatches is depicted by Fig. 5-c. It occurs when all BPs produce and then consume their respective messages. Like the previous mismatch and as discussed in [3] [7], the time decoupling of message production and message consumption enables to easily resolve this mismatch.

**Message Order Mismatch: Firstly Receiving and Then Sending.** Fig. 5-d depicts this kind of mismatches. It occurs when each BP is expecting a message that will be generated by another BP (itself expecting first a message before generating the corresponding message). Current methods [3] [7], which take into account only control dependencies, consider this kind of mismatch as unresolvable.

However, like the *Missing Message Pattern*, this kind of mismatches is resolvable if (1) one or many BPs are expecting *ACK* for sent messages, or (2) one or many BPs are expecting messages which are not mandatory for the following activities. In such cases, the *Message Generator*

is asked to produce the required acknowledgments or the *mock-up* messages which will be sent to the relevant BPs.

## 5 Prototype Implementation

Our prototype is implemented based on *JGraphPad*<sup>6</sup>. BP, control and data dependency graphs for TS are presented in Fig. 6-a, 6-b and 6-c respectively, which are coded in terms of *GXL*<sup>7</sup> format. Fig. 7 shows our *MEDI* tool to simulate an interaction between TS and TR and to demonstrate that, using our space based process mediator, the interaction can be successfully mediated as discussed in the motivating example. For simulation purposes, the content of a message includes a string which represents the name of the message, rather than the real business data.



Figure 7. A successful mediated Web service interaction between TS and TR

## 6 Related Work

We discuss the related work from the following two aspects: *process mediation* and *service interaction*.

<sup>6</sup><http://www.jgraph.com/jgraphpad.html>.

<sup>7</sup><http://www.gupro.de/GXL/index.html>.

**Process mediation.** There are two kinds of process mediation initiatives: *syntactic* methods enabling the interactions of BPs which are modeled by different workflow protocols or languages, such as EDIFACT<sup>8</sup>, RosettaNet<sup>9</sup>, XPD<sup>10</sup>, ebXML<sup>11</sup> or BPEL etc, and *semantic* methods which focus on process sequence mismatches [10] although BPs are modeled using the same language. Our SPM is actually a *semantic* method for process mediation.

**Syntactic process mediation.** A description driven adaptation was proposed in [21] where: (1) common domain-specific and protocol-independent communication acts are identified, (2) abstract protocols describe the sequencing of communication acts, (3) concrete protocols elaborate interaction behaviors required by the client to initiate and perceive communication acts, and thus (4) service providers can be substituted by each other albeit they are modeled using different protocols. In [8], the authors proposed a mediation architecture, which includes *Mediation Interface*, *Mediation Service* and *Mediation Repository*, for service interactions of BPs modeled by different workflow languages.

**Semantic process mediation.** In [3] [18], Benatallah et al presented an adapter-based approach to semi-automatically resolve business protocol mismatches. *Mismatch patterns* were used to formalize the mismatches and thus, potentially, to provide a uniform mechanism to address these mismatches. In addition, they proposed to identify actual mismatches semi-automatically (may need the help of service providers). WSMX<sup>12</sup> proposed five basic mismatch patterns for process mediation [7], integrated process mediation as a component of WSMX, and specified the interaction mode with other components [12]. Furthermore, mismatch patterns were classified into five classes, and thus, potentially, these mismatches could be resolved in a uniform way [27]. However, they need a design-time mediation by developing specific adapters to resolve specific mismatch patterns respectively, and these mismatch patterns are related to control-flow (actually sequencing constraints) only. They claim to support *Message Order Mismatch*. This is a wrong claim because *Message Order Mismatch* is actually partial-resolvable as discussed in Section 4. In addition, they do not, although potentially can, provide a uniform mechanism to resolve all kinds of mismatches at runtime.

In [19], the authors proposed to automatically mediate two OWL-S process models [1], and the process mediation was supposed to find an appropriate mapping for the *possible execution paths* between the provider and the requestor. However, the authors focused on the *structural difference* between process models and can only resolve some

of the mismatch patterns described by [3] [7]. A similar approach was proposed in [5], where the authors proposed to automatically generate BPEL adapters to solve behavioral mismatches. BPEL processes are firstly translated into YAWL<sup>13</sup> workflows, and an adapter, which is also a YAWL workflow, is generated based on *Service Execution Trees*, thereafter the lock analysis is applied to check whether the adapter is a *full* or *partial* adapter.

A Virtual Provider (VP), or a mediator, was proposed in [2] which is defined as a high-level behavioral interface based on ASM<sup>14</sup>. In fact, VP can be categorized as a both *syntactic* and *semantic* method. A process mediator, which acts as an integration model for supporting process equivalence, was proposed in [25] to check if BPs are flatly bi-similar, and to facilitate BP integration and collaboration. A requester-based mediation framework was presented in [13] which proposed to improve the flexibility and reliability of Web service interactions. The *Invocation Engine* works like a proxy in the middle to solve *element* and *choreography* mismatches. In [9], using *deletion* and *resolution*, the authors proposed a mediator based approach to resolve the incompatibility among behavioral part of Web service interfaces and thus to facilitate service interactions.

To summarize, current methods mainly focus on control-flow but largely ignore data-flow. This ignorance suggests that they are limited to deal with the issues raised by our motivating example.

**Service interaction.** In [20], the authors proposed a *Public-to-Private* approach for inter-organizational workflow interoperations based on *inheritance*. This is a top-down approach and may not fit with Web service domain. In [22], a bottom-up approach was presented to establish a consistent, multi-lateral collaboration in a decentralized way. However, due to privacy and security concerns, it may not fit with Web service domain. A view-based approach [6] was proposed to support dynamic inter-organizational workflow cooperation including three steps: *advertisement*, *interconnection*, and *cooperation*. In [25], the authors proposed to support process integration for extended enterprise, i.e. integration of intra- and inter-enterprise. However, current approaches mainly focus on control-flow, and aim to support direct service interactions only in which no mismatches are allowed. They are limited to support mediated service interactions.

## 7 Conclusion and Future Work

We have identified that process mediation is critical for Web service interactions. However, current process mediators are limited because they mainly focus on control-flow

<sup>8</sup><http://repository.edifice.org/migs/index.htm>.

<sup>9</sup><http://www.rosettanet.org/cms/sites/RosettaNet/Standards/index.html>.

<sup>10</sup><http://www.wfmc.org/standards/xpdl.htm>.

<sup>11</sup><http://www.ebxml.org/>.

<sup>12</sup><http://www.wsmx.org/>.

<sup>13</sup><http://www.yawl-system.com/>.

<sup>14</sup><http://www.eecs.umich.edu/gasm/>.

but largely ignore data-flow. They improperly regard mismatches, which actually can be resolved with the help of data dependencies, as unresolvable. In addition, they often need a design-time mediation, i.e. specific adapters are developed to resolve specific mismatch patterns respectively. To address these problems, we have introduced a space based process mediator which (1) considers both control-flow and data-flow, (2) can settle not only all mismatch patterns prescribed by existing process mediation approaches, but also new mismatch patterns related to data-flow, and (3) can resolve all mismatches in a uniform mechanism at run-time, and does not need a design-time work.

Our process mediator presented in this paper is one important component of our broader *MEDI* tool, partially implemented, which aims to support mediated service interactions. Our future work includes (1) compatibility and similarity analysis, (2) conversation set generation supported by BPs, and (3) to make our *MEDI* tool as a platform to support mediated service interactions.

## References

- [1] OWL-S: Semantic markup for web services. W3C, 2006.
- [2] M. Altenhofen, E. Borger, and J. Lemcke. An abstract model for process mediation. Proceedings of the 7th International Conference on Formal Engineering Methods (ICFEM'05), 2005.
- [3] B. Benatallah, F. Casati, D. Grigori, H. R. M. Nezhad, and F. Toumani. Developing adapters for web services integration. Proceedings of International Conference. of Advanced Information System Engineering (CAiSE'05), 2005.
- [4] B. Benatallah, F. Casati, and F. Toumani. Representing, analysing and managing web service protocols. *Data and Knowledge Engineering*, 58(3):327–357, 2006.
- [5] A. Brogi and R. Popescu. Automated generation of bpel adapters. Proceedings of the 4th International Conference on Service Oriented Computing (ICSOC'06), 2006.
- [6] I. Chebbi, S. Dustdar, and S. Tata. The view-based approach to dynamic inter-organizational workflow cooperation. *Data and Knowledge Engineering*, (2):139–173, 2006.
- [7] E. Cimpian and A. Mocan. Wsmx process mediation based on choreographies. Proceedings of the first International Workshop on Web Service Choreography and Orchestration for Business Process Management at the BPM, 2005.
- [8] R. Farmer, A. Raybone, R. Uddin, M. Odetayo, and K.-M. Chao. Mediation architecture for integration of heterogeneous discipline focused workflow languages. Proceedings in IEEE International Conference on e-Business Engineering (ICEBE'07), 2007.
- [9] M.-C. Fauvet and A. Ait-Bachir. An automaton-based approach for web service mediation. Proceedings of the 13th ISPE International Conference on Concurrent Engineering (ISPE CE'06), 2006.
- [10] D. Fensel and C. Bussler. The web service modeling framework wsmf. *Electronic Commerce Research and Applications*, pages 113–137, 2002.
- [11] X. Fu. *Formal Specification and Verification of Asynchronously Communicating Web Services*. PhD thesis, University of California at Santa Barbara, 2004.
- [12] T. Haselwanter, P. Kotinurmi, M. Moran, T. Vitvar, and M. Zaremba. Wsmx: A semantic service oriented middleware for b2b integration. Proceedings of the 4th International Conference on Service Oriented Computing (ICSOC'06), 2006.
- [13] B. Lin, N. Gu, and Q. Li. A requester-based mediation framework for dynamic invocation of web services. Proceedings of IEEE International Conference on Services Computing (SCC'06), 2006.
- [14] R. Liu and A. Kumar. An analysis and taxonomy of unstructured workflows. Proceedings of International Conference on Business Process Management (BPM'05), 2005.
- [15] A. Martens. Analyzing web service based business processes. Proceedings of International Conference on Fundamental Approaches to Software Engineering (FASE'05), Part of the 2005 European Joint Conferences on Theory and Practice of Software (ETAPS'05), 2005.
- [16] S. Moser, A. Martens, K. Gorchach, W. Amme, and A. Godlinski. Advanced verification of distributed ws-bpel business processes incorporating cssa-based data flow analysis. Proceedings in IEEE International Conference on Services Computing (SCC'07), 2007.
- [17] M. Nagarajan, K. Verma, A. P. Sheth, J. Miller, and J. Lathem. Semantic interoperability of web services - challenges and experiences. Proceedings of the 4th IEEE International Conference on Web Services (ICWS'06), 2006.
- [18] H. R. M. Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-automated adaptation of service interactions. Proceedings of the 16th international conference on World Wide Web Conference (WWW'07), 2007.
- [19] R. Vaculin and K. Sycara. Towards automatic mediation of owl-s process models. Proceedings in IEEE International Conference on Web Services (ICWS'07), 2007.
- [20] W. M. P. van der Aalst and M. Weske. The p2p approach to interorganizational workflows. Proceedings of The 13th International Conference on Advanced Information Systems Engineering (CAiSE'01), 2001.
- [21] S. K. Williams, S. A. Battle, and J. E. Cuadrado. Protocol mediation for adaptation in semantic web services. Technical report, HP Technical Report. HPL-2005-78, 2005.
- [22] A. Wombacher. *Decentralized establishment of consistent, multi-lateral collaborations*. PhD thesis, Faculty of Informatics, Technical University Darmstadt, 2005.
- [23] Q. Wu, C. Pul, A. Sahai, and R. Barga. Categorization and optimization of synchronization dependencies in business processes. Proceedings in IEEE 23rd International Conference on Data Engineering (ICDE'07), 2007.
- [24] Q. Yu, X. Liu, A. Bouguettaya, and B. Medjahed. Deploying and managing web services: issues, solutions, and directions. *The VLDB Journal*, 17(3):537–572, 2008.
- [25] J. Zdravkovic. *Process Integration for the Extended Enterprise*. PhD thesis, Royal Institute of Technology, 2006.
- [26] Z. Zhou. A scenario-view based approach for supporting mediated web service interaction. Proceedings in CAiSE Doctoral Consortium (CAiSE-DC'08), 2008.
- [27] Z. Zhou, B. Sapkota, E. Cimpian, D. Foxvog, L. Vasiliua, M. Hauswirth, and P. Yu. Process mediation based on triple space computing. Proceedings in the 10th Asia Pacific Web Conference (APWeb'08), 2008.