Rule-based Automatic Generation of Mediator Patterns for Service Composition Mismatches

Feng Jiang, Yushun Fan, Xun Zhang
Department of Automation, Tsinghua University, Beijing, 100084, P.R. China
jiangf00@mails.thu.edu.cn, fanyus@tsinghua.edu.cn, xunzh@tsinghua.edu.cn

Abstract

To perform service composition, mismatches are challenging obstacles due to the decentralization and independent development of services. Mediation, as a promising solution, attracts most attentions. And pattern based mediation proposed a modularly constructive thoughtway: Basic mediator patterns were created and sufficient for advanced mediators against all possible mismatches. The pattern structure is illustrated in this paper. And construction rules for each pattern are presented. Executable codes such as BPEL codes can be automatically generated from these rules. As a systematic engineering solution, its feasibility is validated through a case study in the end.

1. Introduction

Composition is one of the most important research areas in service oriented environment. Services, due to the decentralization and independent development, are not perfectly compatible with one another. Thus mismatches which may cause structure or temporal faults of workflows [1] come to be unavoidable and common problems. Mediation, which retrofits existing services by intercepting, storing, transforming, and (re-)routing messages going into and out of these services, is considered as one of the most effective solutions, and has become the status of a definite working area in the fields of SOA [2].

In comparison to various mediation researches, which may either focus on message types transforming [3] or aggress towards a partial solutions for message exchanging sequences [4], our work aims at developing a systematic engineering solution to semi-automatically generate service mediators for resolving all kinds of composition mismatches. This is achieved based on a comprehensive classification of these mismatches defined in our previous work [5], where Service composition mismatches were categorized into functional and nonfunctional domains, both of which were further divided into semantic and syntactic levels. Since either nonfunctional-domain related or semantic-level related mismatches can seldom be conquered by normal mediation method, only syntactic functional mismatches consisting of signature and protocol ones are mainly concerned in our work. And syntactic protocol mismatches are further specified into six basic patterns: extra and missing messages, merging and splitting messages, extra and missing conditions.

We have created six basic mediator patterns and presented a mechanism about the composition of patterns to modularly construct advanced mediators in [7]. However, these basic patterns are conceptual ones. In this paper we illustrate the structure of these patterns and present the construction and configuration rules for automatic generation of executable mediators. And the systematic engineering solution is studied throughout.

The rest of this paper is structured as follows: Section 2 introduces seven mediator patterns which are proved to be sufficient. Section 3 illustrates the structure of mediator pattern and construction and configuration rules for each pattern. The feasibility is validated through a case study in section 4. Related work and comparison are presented in Section 5. And finally, conclusions are drawn up in Section 6 with our acknowledgements followed.

2. Mediator patterns

Mediator patterns are originated from the classification of mismatches described in article [5], Based on this specification, six conceptual mediator patterns are developed:

(1) Storer pattern: receiving and storing messages. It is used for mismatches of extra message.

(2) Constructor pattern: constructing and sending messages. It is used for mismatches of missing message.
(3) **Merger pattern**: receiving two or more partial messages and merging them into one single message. It is used for mismatches of merging messages.

(4) **Splitter pattern**: receiving a single message and splitting it to two or more partial messages. It is used for mismatches of splitting messages.

(5) **Storing Controller pattern**: storing and conditionally sending messages of certain type in terms of specific logic. This pattern can be used for mismatches of extra conditions for a receive activity.

(6) **Constructing Controller pattern**: conditionally constructing and sending messages of certain type in terms of specific logic. This pattern can be used for mismatches of missing conditions for a receive activity.

Thus basic protocol mismatches are resolved by the abovementioned basic mediator patterns. And as a fact that complicated mismatches are composited by basic ones, they can be compensated by composed mediator patterns. However, in order to addressing all possible mismatches, one more mediator pattern, the **signature transformer pattern**, is crucially required, whose construction will be discussed in the follow context.

### 3. Construction and configuration rule

Abovementioned basic mediators are conceptual ones which can not be directly used for service mediation unless they are transferred into executable ones. As the intended benefit of this work is to help developers produce executable mediators through a slight engineering way, rule-enhanced patterns for automatic construction of executable mediator codes are preferred. And it is important to guarantee the configurability of these patterns during constructing. As BPEL4WS is the main language for service composition, we also accept BPEL4WS as our business process modeling and mediator constructing languages in this paper.

#### 3.1. Mediator structure

The purpose to introduce mediators when composing services together is to make both of the requestor and provider interfaces fit each other smoothly. Thus every mediator should have at least one input and/or output interface(s) connecting to the requestor and/or the provider interface(s) defined in a BPEL specification. And inside a mediator, transform rules are required to deal with these input and output interfaces. The structure can be illustrated as Figure 1.

The $1$ to $n$ input interfaces are used to receive messages sent from a service requestor while the $l$ to $m$ output interfaces are used to send messages to a provider. Rule engine is to determine which transform rule defined in the rule base should be activated.

![Figure 1. Structure of mediator pattern](image-url)

For sake of convenience, IM is short for Incoming Message from a send activity, OM for Outgoing Message to a receive activity, SM for requestor Sending Message and RM for provider Required Message. Obviously, IM=SM and OM=RM.

#### 3.2. Construction and configuration rule

1. **Rule engine**

   Construction: The rule engine acts as brain of a mediator, determining which transform rule should be activated to construct a mediator. This is achieved according to the mediator pattern label which is preset manually by developer. The engine rule is as follows:

   ```plaintext
   If (patternlabel=signature transformer) Then
       invoke rulename="SignatureTransformerRule";
   elseif (patternlabel=storer) Then
       invoke rulename="StorerRule";
   ..... elseif (patternlabel=constructing controller) Then
       invoke rulename="ConstructingControllerRule";
   End If;
   ``

   Configuration: Variable patternlabel is configured to match mismatches when designing a workflow process. If the composition mismatch is judged as a signature mismatch, patternlabel is set to be signature transformer and a signature transformer mediator is created according to the rule presented below.

2. **Signature transformer rule**

   Message carried in a send/receive activity of a service interface can be presented as:

   ```plaintext
   Message:= Messagename{part 1; part 2; ...; part n}
   where part:= {partname, parttype, partunit(optional)}
   ```

   It is similar to [8] where message is named element, but much more precise to WSDL and BPEL scheme.

   Rule: Signature transformer mediator is generated when a sending message is not compatible with the message expected. This happens in several ways: the sequence of different parts, the name, the part type and the part unit. Note that missing or extra part is
recognized as merging or splitting mismatches and is not considered here. Rule for generating Signature transformer mediator can be illustrated as follows (OP is short for outgoing message part and RP for required message part):

```
Begin
Receive IM;
Get message name;
While (there is a part i)
  {Get message part i;
   Get part name, part type, part unit separately;
   Delete part i from message;
   i+1;}
End while;
Create OM;
Assign IM to OM;
If (message name mismatches)
  Assign RM name to OM;
If (part name mismatches)
  Assign RP name to OP;
If (part type/unit mismatches)
  Assign RP type/unit to OP according specific conversion definitions;
If (sequence between part i and j has to be changed)
  Assign IP j to OP i and IP i to OP j;
If (part i and j has to be merged)
  Assign IP i + IP j to OP i;
If (part i has to be spited)
  Assign IP i to OP i and OP j;
End If;
Send OM;
End.
```

Construction: IM is divided into parts, and a part falls into name, type and unit. OM is created and initialized with IM. According to mismatch patterns, sub-rules are invoked.

Configuration: The if-condition needs to be configured by service composition designers.

With the signature transformer mediator, the assumption that message itself is suitable for composition can be naturally made and taken for the six rules presented below.

(3) Storer rule

Rule: Storer mediator with the function of receiving and storing message is used when a send activity meets no receive activities. Generating rules are as follows:

```
Begin
Receive IM;
Create messagestorercer;
Assign IM to messagestorercer;
End.
```

Construction: Variable messagestorercer is created to store the incoming message whatever it is.

Configuration: There is no need to make any configurations to a storer mediator.

(4) Constructor rule

Rule: Constructor mediator with the ability of constructing and sending messages is used when a receive activity meets no send activity. Generating rules are as follows:

```
Begin
Create OM;
Assign RM to OM;
Set default instance of OM;
Repeat until (process is terminated)
  {Pick receive activity status;
   onAlarm(receive activity is ready)
     {Send OM to the receive activity;}
  }
End.
```

Construction: OM with a default instance is created according to the definition of RM. As soon as the receive activity is ready for execution, the default OM instance is immediately sent.

Configuration: The default OM instance needs to be configured according to scenarios where the receive activity is running.

(5) Splitter rule

Rule: Splitter mediator is used when message from one send activity splits into more than one receive activities. Generating rules are as follows:

```
Begin
Receive IM;
While (there is a part i)
  {Get message part i;
   Delete part i from IM;
   i+1;}
Assign amount of splitting target messages to k;
While (k)
  {Create OMk;
   Assign amount of target parts in OMK to l;
   While (l)
     {Assign j according to RM;
      Assign IP j to OP l;
      l-1;}
   End While;
   Send OMk;
   k-1;}
End While;
End.
```

Construction: IM is divided into parts. And OMK consisting of IM parts is created according to RM.

Configuration: The local variable k is configured to the amount of target messages. IP j that composes OMK is also configurable according to RM.
(6) Merger rule
Rule: Merger mediator operates when messages from more than one send activities merge into one receive activity. Generating rules are as follows:

```
Begin
   Assign amount of merged messages to k;
   Create OM;
   While (k)
      {Receive IMk;
       Assign IMk to OPk;
       k-1;}
   End While;
   Send OM;
End.
```

Construction: OM is created for merging incoming messages which are assigned as parts of OM.
Configuration: The local variable $k$ is configurable to the amount of merged messages.

(7) Storing controller rule
Rule: Storing controller mediator is used where a sending message is unexpected if its condition is not satisfied. Generating rules are as follows:

```
Begin
   Receive IM;
   Create messagestorer;
   Assign IM to messagestorer;
   Pick if-conditions of RM;
   onAlarm (condition is satisfied)
      {Create OM;
       Assign messagestorer to OM;
       Send OM;}
End.
```

Construction: Variable messagestorer is created to store the incoming message, similar to the storer rule. The if-condition of RM is picked by the mediator. And when it is satisfied, OM is created and sent out to the receive activity. Otherwise, IM is blocked.
Configuration: The if-condition of RM is left for developers to pre-deploy manually.

(8) Constructing controller rule
Rule: Constructing controller mediator is used where a sending message is under expecting although its condition is not satisfied. Generating rules are as follows:

```
Begin
   Create OM;
   Assign RM to OM;
   Set default instance of OM;
   Pick if-conditions of SM;
   onAlarm (condition is not satisfied)
      {Send OM to the receive activity ;}
End.
```

Construction: OM with a default instance is created according to the definition of RM, similar to the constructor rule. The if-condition of SM is picked by the mediator. And when it is not satisfied, OM is sent out to the receive activity.
Configuration: The default instance of OM and if-condition of SM needs to be configured.

4. Feasibility validation

These rules are used for generating mediators. It is easy to transform them into executable ones although they are pseudo-codes. It is proved to be sufficient to conquer all mismatches [7]. To prove the feasibility, a motivating example where mismatches are resolved by a composite mediator which is constructed by mediator patterns is studied.

4.1. Motivating scenario

The motivating scenario comes from where we use an Automatic Teller Machine (ATM). As a customer application client, ATM is developed and maintained by a third provider and has to be composited with specific bank e-service (BS).

ATM sends card number to BS first and then simultaneously sends password and target amount of money. After that ATM waits for permission: If permitted, money is disgorged and a confirmation is sent to BS, with waiting for credence; otherwise, ATM process is terminated. On the other side, BS receives card number and password and makes a validation: If password approved, BS receives target amount of money, and sends permission to ATM. When confirmation is arrived, BS process terminates; otherwise, password is not approved and BS process terminates with an error sent to ATM.

```
Begin
   Create OM;
   Assign RM to OM;
   Set default instance of OM;
   Pick if-conditions of SM;
   onAlarm (condition is not satisfied)
      {Send OM to the receive activity ;}
End.
```

For the sake of clear representation, we abstract the
BPEL definitions of the two services as Figure 2.

There are complex mismatches in this scenario:

i) BS expects an incoming message containing both card number and password, while ATM sends them separately. Merging mismatch is identified here.

ii) Password sent by ATM is in conjunction with the target amount of money, while BS expects the amount independently. Splitting mismatch is identified here.

iii) BS accepts amount only when password is approved, while ATM sends it in any case. Thus missing controller mismatch is identified here.

iv) BS sends an error message to ATM if password not approved, while ATM expects a message named permission. Signature mismatch is identified here.

v) ATM expects credence from BS before process is terminated while BS does not send any credence. Message missing mismatch is identified here.

In the next section, we will use mediator patterns to construct a composited mediator to conquer all these mismatches. And executable codes will be generated automatically.

4.2. Problem solution

There are five mismatches in the motivating scenario and five mismatch patterns can be respectively used to address these mismatches:

i) A splitter pattern is used to receive message password&amount (password; amount) and split it into two messages password and amount.

ii) A merger pattern is used to receive two messages, one is from the SendCardNO activity and the other from splitter mediator. They are merged as one message cardno&pw (cardno; password) which is sent to BS.

iii) A storer controller pattern is used to store the message amount from splitter and to pick the password validation result which is set as the if-condition of BS ReceiveAmount activity. If password not approved, the message amount is stored only. Otherwise, it is sent to BS.

iv) A signature transformer pattern is used to change the message error sent from BS to permission and send it to ATM.

v) A constructor pattern is used to construct a message credence=yes and send to ATM as soon as the ReceiveCredence activity is ready for execution.

These five mediator patterns construct together as a holistic mediator deployed between the two services. See Figure 3(circled in dashed ellipse). Verification can be made here [9] to ensure the correctness.

According to mediator construction rules proposed in section 3.2, executable BPEL codes are generated for this holistic mediator, shown as follow (only snippet due to space limitation of this paper):

![BPEL code snippet]

5. Related work and comparison
There is significant number of researches working on service mediation. Automata [10], Process Algebra [11] and Petri nets [12] are the most common ways for compatibility analysis. While for executable mediator generation, [13] introduces a meta-data based way and [14] makes it based on AID which abstracts interfaces into certain definitions. Similar work takes in [8] and [15]: Interface is divided into choreography and element parts in [8], and mediators generate from special algorithm, while, in [15], from specifications consisting of participating scheme models and mapping rules. It is not a slight way and mediators are not configurable. A newly improvement is pattern-based mediation. A selection of patterns is proposed in [4] based on insufficient taxonomy of composition mismatches. Construction of these patterns, however, is not mentioned. [16] identifies five mismatch patterns and provides templates for generating mediators, however, these patterns are not sufficient either.

Thus the distinct contribution of our work lies in a comprehensive classification of mismatches, an introduction of mediator patterns which are proved to be sufficient, and a set of rules for the automatic generation of configurable mediators.

6. Conclusions

We adopt pattern based tutorial to develop a systematic engineering solution to (semi-)automatically generate mediators for service composition mismatches. Based on a comprehensive mismatches classification, seven basic mediator patterns are introduced. Pattern structure and construction rules are illustrated in this paper. Executable mediators are automatically generated from these rules, and can be configured according identified mismatches. As patterns can be used to modularly construct advanced mediators, they are sufficient for addressing all mismatches. The feasibility can be seen through the real-life scenario happened between Automatic Teller Machine (ATM) service and bank e-service.

Acknowledgments

This work on service mediation was granted by China National Natural Science Foundation (No. 60674080) and National High-Tech R&D (863) Plan (No. 2006AA04Z151 and 2006AA04Z166). Also, it was supported by joint project “Mediator-aided Service Composition” cosponsored by IBM CRL and National CIIMS Engineering Center, Tsinghua University.

References