A Pattern-Based Approach to Protocol Mediation for Web Services Composition

Xitong Li
Yushun Fan
Stuart Madnick
Quan Z. Sheng

Working Paper CISL# 2008-10

September 2008

Composite Information Systems Laboratory (CISL)
Sloan School of Management, Room E53-320
Massachusetts Institute of Technology
Cambridge, MA 02142

A Pattern-Based Approach to Protocol Mediation for Web Services Composition

Xitong Li [a,*], Yushun Fan [a], Stuart Madnick [b], Quan Z. Sheng [c]

[a] *Department of Automation, Tsinghua University, Beijing 100084, P.R. China*

[b] *MIT Sloan School of Management, 50 Memorial Drive, Cambridge, MA 02142, USA*

[c] *School of Computer Science, University of Adelaide, Adelaide, SA 5005, Australia*

* Corresponding author

E-mail address: lxt04@mails.tsinghua.edu.cn

Abstract

With the increasing popularity of Service Oriented Architecture (SOA), service composition is gaining momentum as the potential silver bullet for application integration. However, services are not always perfectly compatible and therefore can not be directly composed. Service mediation, roughly classified into signature and protocol ones, thus becomes one key working area in SOA. As a challenging problem, protocol mediation is still open and existing approaches only provide partial solutions. In this paper, a systematic approach based on mediator patterns is proposed to generate executable mediators and glue partially compatible services together. The mediation process and its main steps are introduced. By utilizing message mapping, a heuristic technique for identifying protocol mismatches and selecting appropriate mediator patterns is presented. The corresponding BPEL templates of these patterns are also developed. Moreover, a prototype system, namely Service Mediation Toolkit (SMT), is implemented to validate the feasibility and effectiveness of our approach.

Keywords: Service oriented architecture; Web service; Service composition; Protocol mediation; Mediator

1. Introduction

Service Oriented Architecture (SOA) is a newly-emerging software architecture consisting of loosely-coupled services that communicate with each other through open-standard interfaces [1, 2]. With the increasing popularity of SOA, service composition is gaining momentum as the potential silver bullet for the seamless integration of heterogeneous computing resources, rapid deployment of new business applications, and increasing reuse possibilities to a variety of legacy systems [3-5].

Based on our observation, however, there exist various challenges that result in the incompatibilities/mismatches of services composition. Firstly, Web services are usually developed separately and independently. Secondly, services are not unalterable and need evolution. With the variation of business requirements, service evolution and upgrading have to be addressed. Besides that, services must be interacted with client applications. It is impossible to make these services consistent with the large number of client applications. Last but not least, software developers can not always predict the deployment and runtime contexts when they develop Web services. As a result, few Web services are exactly compatible and additional efforts are needed to compose these partially compatible services together. By partial compatibility, we mean the situation that two (or more) services provide complementary functionalities and could be composed together in principal; however, their interfaces

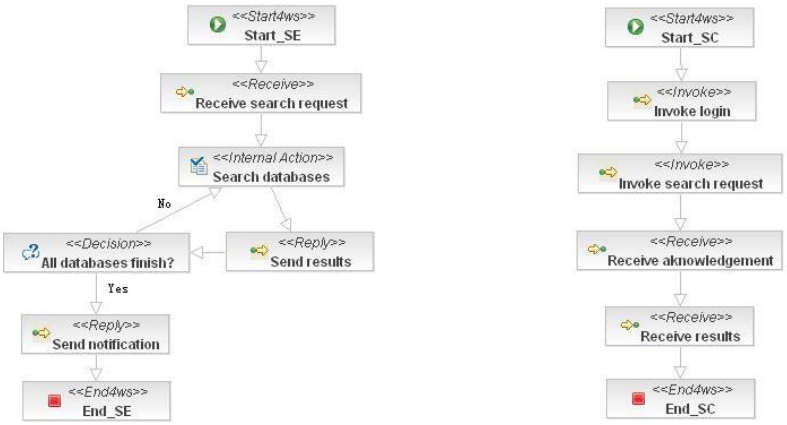
and interaction protocols do not fit each other exactly.

An effective solution to these challenges is *service mediation*, which enables a service requester to connect to a relevant service provider regardless of the heterogeneities between them and works in a transparent way – neither of them needs to be aware of its existence [6]. First proposed in the Enterprise Service Bus (ESB) industry community [7], service mediation is referred to as the act of retrofitting existing services by intercepting, storing, transforming, and (re-)routing messages going into and out of these services [8]. Nowadays, service mediation has become a key working area in the field of SOA and Component-Based Software Engineering (CBSE) [9-11].

Service mediation can be roughly classified into signature and protocol. *Signature mediation* which focuses on message types has received considerable attention [12-14] and many commercial tools have been developed, such as Microsoft BizTalk Mapper[1], Stylus Studio XML Mapping Tools[2] and SAP XI Mapping Editor[3]. In comparison, the problem of *protocol mediation* (also known as process mediation), which aims at reconciling mismatches of message exchanging sequences, is still open. A frequently-used approach to this issue is to develop a mediator/adapter which is a piece of code that sits between the interacting services and reconciles the mismatches [15-17]. However, the mediators developed by existing approaches have no control logics and can not compensate complicated mismatches. Few of these approaches can be used to automatically generate executable codes of the mediators. Additionally, no existing approach provides a comprehensive solution to protocol mediation for Web services composition. Last but not least, to the best of our knowledge, there exists no software tool which assists developers to ease their efforts on mediation tasks, such as identifying protocol mismatches or generating mediation codes. This paper presents the approaches to resolving these problems.

1.1. Motivating Example

We present a motivating example that will be used to demonstrate our research idea and approach throughout the paper, as shown in Fig. 1.



(a) BPEL of search engine service (b) BPEL of search client service

Fig. 1. A motivating example of service composition with protocol mismatches

The example consists of a search client (S_C) and a search engine (S_E). S_C invokes S_E by sending its login information and the search request respectively. After that, S_C waits for the acknowledgement and

[1] http://msdn.microsoft.com/en-us/library/ms943073.aspx
[2] http://www.stylusstudio.com/
[3] http://www.wsw-software.de/en-sap_services-mapping_sap_xi.mapping-sap-xi.html

the results from S_E. On the other hand, after receiving search request, S_E starts to search several distributed databases one by one (by performing its internal searching action). Once S_E finishes a database and obtains some searched items, it sends these items to S_C immediately. When all databases have been searched, S_E sends a completing notification to S_C and the search work is finished.

Among various specification languages of service composition (e.g., BPEL, WS-CDL, WSCI), BPEL obtains the dominance and has been proposed by OASIS as an industry standard which is supported by major software vendors such as IBM, Oracle and SAP. In this paper, we take BPEL as the specification language for describing the protocol of Web services. Fig. 1 shows the BPEL of the search engine and the search client services, i.e., S_E and S_C. It is easy to see that S_E and S_C are partially compatible. They provide complementary functionalities but do not fit each other exactly. Apparently, without reconciling the protocol mismatches between them, S_E and S_C can not interact with each other successfully.

1.2. Contributions

The rationale of our work has been presented in the conference papers [18, 19]. As an extension of our previous work, we aim at developing a systematic approach to (semi-)automatically generating mediators for reconciling all possible protocol mismatches. The main contributions are as follows:

1) We present several basic mediator patterns which are derived from basic protocol mismatches identified in our previous work [20]. With the knowledge of protocol mismatches, the well-defined basic mediator patterns can be configured and composed by service developers. These basic mediator patterns are referred to as a sufficient set of building blocks which can be used to construct advanced mediators and reconcile all possible protocol mismatches.

2) We propose a technique to semi-automatically identify protocol mismatches when two partially compatible services need to be composed together. The technique is based on message mappings which are specified by service developers. By using the technique, basic mediator patterns are semi-automatically selected according to the identified protocol mismatches.

3) We develop BPEL templates for the mediator patterns which can be used to generate executable mediation codes. Each mediator pattern has a corresponding BPEL template and a composite mediator corresponds to a combined BPEL-based mediation code.

4) We propose a systematic engineering approach for service developers to reconcile all possible protocol mismatches. The approach combines our work on identification of protocol mismatches, selection of mediator patterns and code generation of BPEL-based mediation codes. All these mediation tasks can be performed (semi-)automatically.

5) We develop a prototype system, namely Service Mediation Toolkit (SMT), which provides a user-friendly workbench and can assist service developers to ease their efforts on the mediation tasks. As an implementation work, SMT is integrated with IBM WebSphere Integration Developer (IBM WID)[4] and validates the feasibility and effectiveness of our approach.

The rest of the paper is structured as follows. In Section 2, several basic mediator patterns are proposed. The configurability and composability of the mediator patterns are presented in this section as well. The proposed approach to protocol mediation is presented in Section 3. The technique for selecting mediator patterns based on message mapping is also introduced and BPEL templates of the mediator patterns are developed for code generation of executable mediators. And then, the prototype

[4] http://www-306.ibm.com/software/integration/wid/

system, i.e., Service Mediation Toolkit (SMT), is presented in Section 4. In Section 5, related work and the comparisons with ours are given. Finally, the conclusion and future work are drawn up in Section 6.

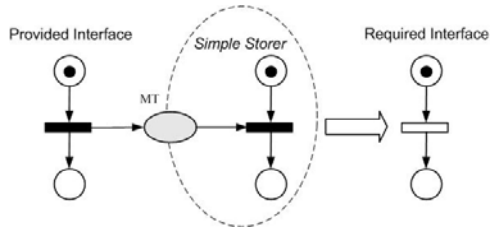
2. Protocol Mediator Patterns

2.1. Basic Mediator Patterns

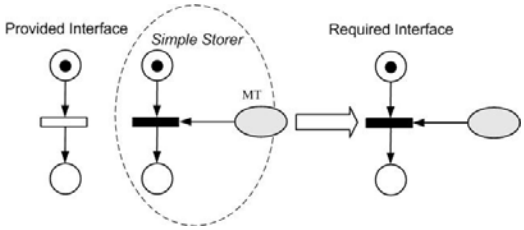
An effective solution to reconciling protocol mismatches is to develop a mediator. By protocol mismatches, we mean the mismatches that occur in the message exchanging sequences between two partially compatible services. In our previous work, we have proposed several basic mismatches that can be referred to as basic constructs of all protocol mismatches [20]. And we have developed six basic mediators for reconciling the basic mismatches. It has been pointed out that these basic mediators can be referred to as basic patterns which assist service developers to modularly construct more powerful mediators and reconcile all possible protocol mismatches [18]. Hence, the set of basic mediator patterns is considered to be sufficient. To make this paper self-contained, we present the six basic mediator patterns and corresponding using scenarios in this section. Detailed illustrations of the mediator patterns are presented in [18].

Note that the protocols of both Web services and mediators are depicted based on Colored Petri Nets (CPN) [21]. The benefit of adopting CPN models as an underlying formalism lies in that they provide rich analysis capability to support formal verification of protocol mediation and solid approaches to the transformation between BPEL and CPN models have been developed [22, 23]. Details of CPN models are given in [21]. In the following figures, the round places (i.e., circles) depict the states of control flows of Web services; the gray ellipse places depict the messages of Web services communicated with outside partners. The black transitions (i.e., filled rectangles) depict the operations of Web services that send/receive messages; the white transitions (i.e., empty rectangles) depict those actions without sending/receiving any message. The symbol “MT” stands for a specific message type.

(1) Simple Storer pattern: the mediator with the capability of simply receiving and storing messages. It is used for reconciling mismatches of extra sending messages and missing receiving messages, as shown in Fig. 2.



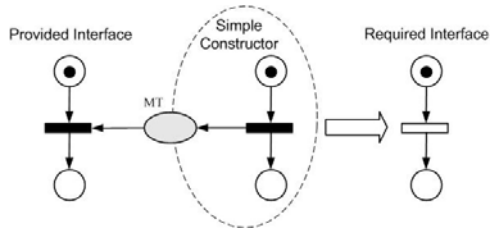
(a) Extra sending message scenario



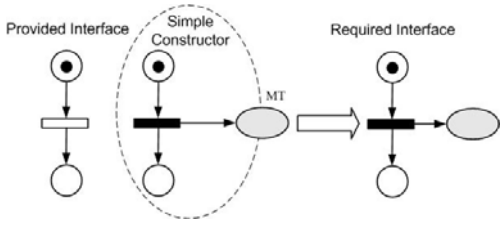
(b) Missing receiving message scenario

Fig. 2. Scenarios of using Simple Storer pattern

(2) Simple Constructor pattern: the mediator with the capability of simply constructing and sending messages. It is used for reconciling mismatches of extra receiving messages and missing sending messages, as shown in Fig. 3.



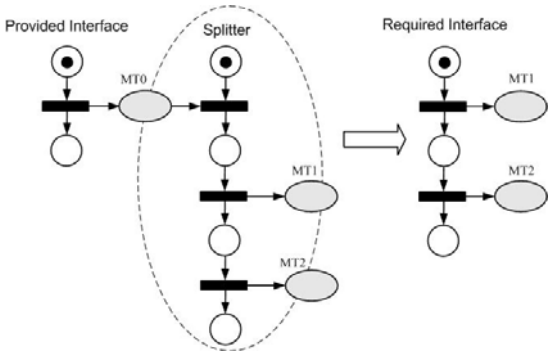
(a) Extra receiving message scenario



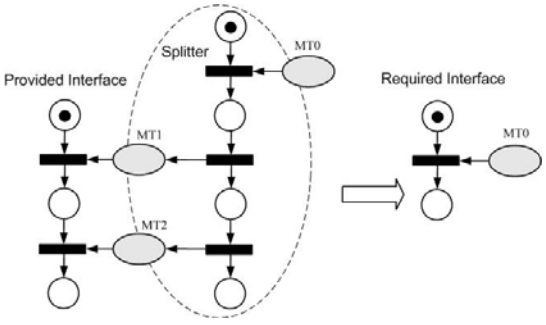
(b) Missing sending message scenario

Fig. 3. Scenarios of using Simple Constructor pattern

(3) Splitter pattern: the mediator with the capability of receiving a single message and splitting it into two or more partial messages. It is used for reconciling mismatches of splitting sending messages and merging receiving messages, as shown in Fig. 4.



(a) Splitting sending message scenario

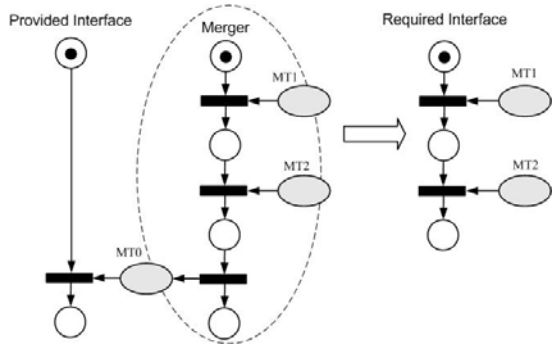


(b) Merging receiving message scenario

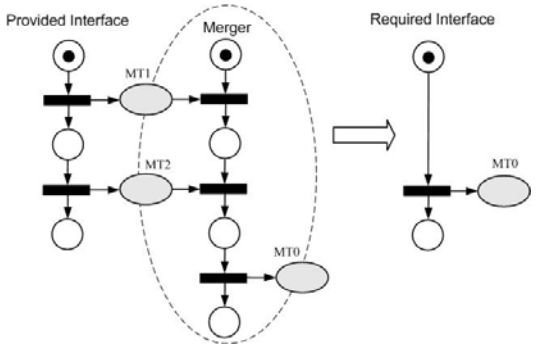
Fig. 4. Scenarios of using Splitter pattern

(4) Merger pattern: the mediator with the capability of receiving two or more partial messages and merging them into a single one. It is used for reconciling mismatches of splitting receiving messages

and merging sending messages, as shown in Fig. 5.



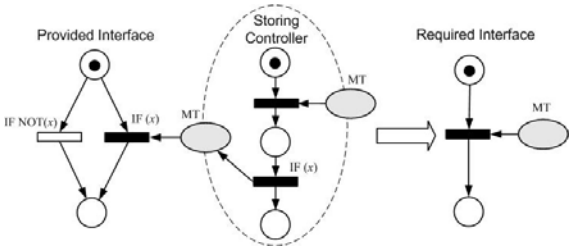
(a) Splitting receiving message scenario



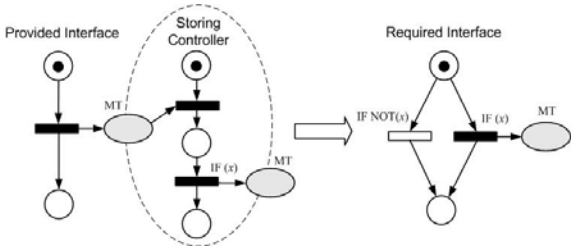
(b) Merging sending message scenario

Fig. 5. Scenarios of using Merger pattern

(5) Storing Controller pattern: the mediator with the capability of storing and conditionally sending some messages in terms of specific logic. It is used for reconciling mismatches of extra condition of receiving messages and missing condition of sending messages, as shown in Fig. 6.



(a) Extra condition of receiving message scenario

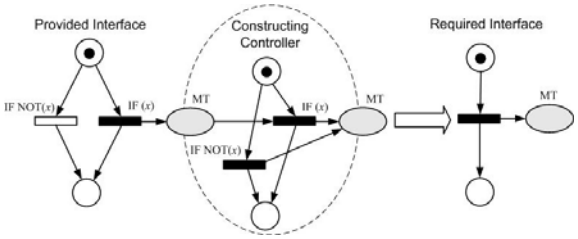


(b) Missing condition of sending message scenario

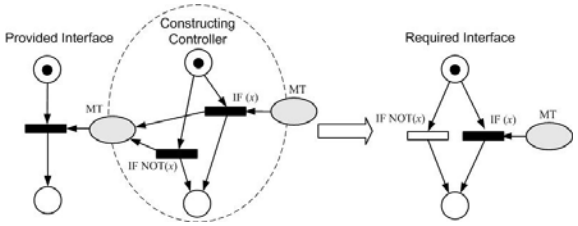
Fig. 6. Scenarios of using Storing Controller pattern

(6) Constructing Controller pattern: the mediator with the capability of conditionally constructing and sending some messages in terms of specific logic. It is used for reconciling mismatches of extra

condition of sending messages and missing condition of receiving messages, as shown in Fig. 7.



(a) Extra condition of sending message scenario



(b) Missing condition of receiving message scenario

Fig. 7. Scenarios of using Constructing Controller pattern

It should be pointed out that both basic and composite mediators presented in [18] are conceptual patterns rather than executable codes. As a further step of our work, the intended benefit of the paper lies in semi-automatic selection of mediator patterns and automatic generation of pseudo-code (i.e., BPEL codes) for protocol mediation.

2.2. Configurability and Composability of Mediator Patterns

As mentioned above, a composite mediator can be constructed by basic mediator patterns and referred to as a new pattern for further use. To facilitate the construction of composite mediators, we investigate the configurability and composability of the mediator patterns.

The specific structures of the Splitter/Merger pattern are variable according to the sequences of the partial messages which may be sequential, parallel or mixed structure. Before generating pseudo-code of the Splitter/Merger pattern, service developers should specify how many partial messages involved and the sequence of these messages. For example, service developers may specify a splitter with three partial messages. After receiving a single message MT0, the splitter may send message MT1 and message MT2 in parallel. And it may send the third partial message MT3 after message MT1 is sent out, as shown in Fig. 8. Once service developers configure the sequence of partial messages, the specific structure of the splitter pattern is identified and automatically concretized by the Service Mediation Toolkit (SMT) (see Section 4).

When reconciling extra or missing condition mismatches, service developers should specify the condition constraints of the Storing Controller and Constructing Controller patterns according to the condition of the provided or required interfaces of services to be composed. The condition constraints are eventually transformed to BPEL elements, such as <*switch*>, <*pick*>, <*while*> or <*repeatUntil*>. For example, there exists a seller service that sends the invoice message after receiving payment from its buyer. However, the buyer service only expects to receive the invoice under the condition that the total payment is greater than 1000 USD. In this case, the Storing Controller pattern can be used to reconcile such mismatch, as shown in Fig. 9. For compatible reconciliation, service developers should specify the internal condition of the Storing Controller pattern. The condition *x* should be specified as

"*Total payment > 1000 USD*".

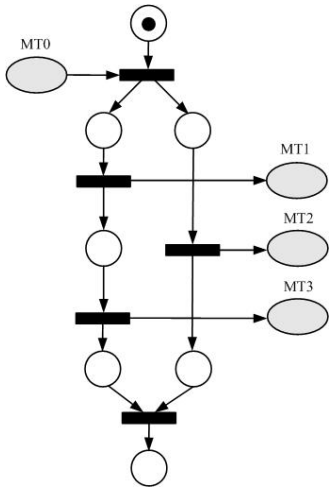


Fig. 8. Splitter pattern with three partial messages

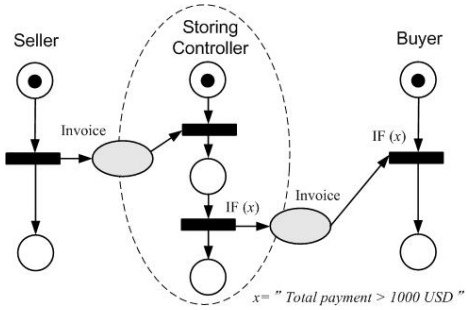


Fig. 9. Storing Controller pattern with specified condition

In real-world situations, protocol mismatches are complicated and should be addressed by advanced mediators with control logics that are composed by the basic mediators. Each mediator presented in this paper has two special places, i.e., the initial place and the end place, as shown in Fig. 8 and Fig. 9. Informally, the composition of two mediators is performed by merging the common parts of the two mediators, and then merging the end place of one mediator with the initial place of the other. To illustrate the composition of mediators, take a mediator with iterative structure, namely *Merging Repeater*, for example, as shown in Fig. 10. It's easy to see that Merging Repeater can iteratively receive messages of the type MT1 until the completing condition *x* occurs. Merging Repeater can be used as a mediator pattern to reconcile protocol mismatches with iterative structure. More details about the configurability and composability of the mediator patterns are given in [18].

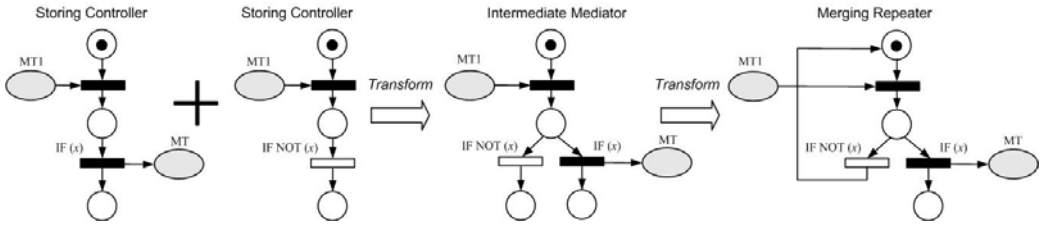


Fig. 10. Merging Repeater pattern composed by two Storing Controller patterns

3. Proposed Approach to Protocol Mediation

3.1. Overview of Mediation Process

As BPEL has become the *de facto* standard for specifying protocols of Web services, we focus on the mediation of BPEL-based services. We take the BPEL files of two partially compatible services as the input. And then, we produce executable mediators as the output for reconciling protocol mismatches and compatibly gluing the two services together if the correct mediator exists. Fig. 11 shows the mediation approach consisting of five steps.

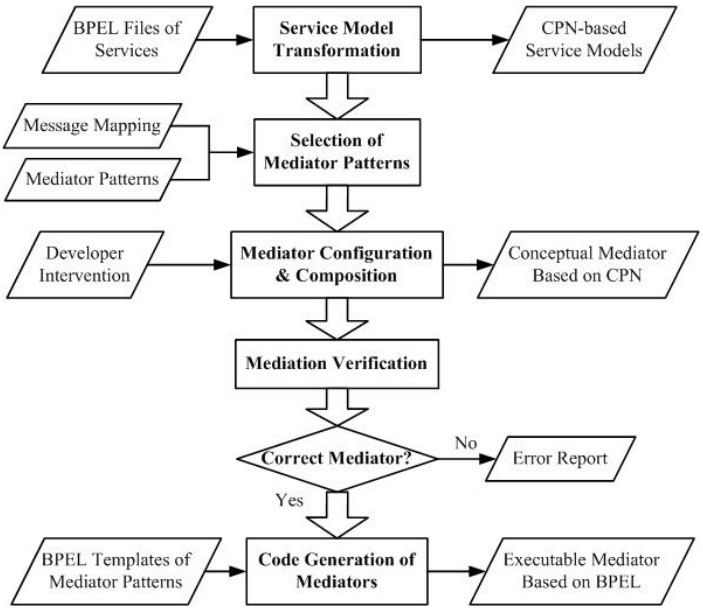


Fig. 11. Overview of the proposed approach

(1) Service model transformation

As the first step, BPEL-based services are transformed to formal models for the purpose of generating and verifying mediators. As mentioned above, the formulism of CPN models can not only depict the internal logic and message exchanging sequences, but also provide rich analysis capability to support solid verification of correctness of protocol mediation. We adopt CPN models to depict the protocols of services and mediators. Techniques for transforming BPEL-based service models to CPN models have been recently proposed in [22, 23].

(2) Selection of mediator patterns

It is very challenging to automatically identify protocol mismatches and select mediator patterns. To this end, we propose a heuristic technique based on message mapping that assists service developers to select appropriate mediator patterns for gluing partially compatible services. The role of message mapping is to define mapping relations for syntactically/semantically equivalent elements of the exchanging messages so that mismatches can be identified. In the WSDL /BPEL specification, message exchanged between Web services are specified as an aggregation of parts and/or elements. In this paper, we assume that the low-level structures (i.e., data types) of the exchanged messages are consistent. Thus the message mappings are specified at the message and part/element level. By performing a selection rule, appropriate mediator patterns are selected automatically. More details about the technique will be presented in Section 3.2.

(3) Mediator configuration and composition

As mentioned in Section 2.2, the structures and control logics of the mediator patterns need to be configured as parameters by service developers according to the identified mismatches. After configuration, the mediator patterns are composed to construct a composite mediator that reconciles all identified protocol mismatches. It is noted that a composite mediator can also be referred to as a complex pattern for further use. Both mediator patterns and composite mediators are depicted as underlying CPN models for the following formal verification.

(4) Mediation verification

The mediator produced in the above steps is only a conceptual model and should be put between the interacting services. The composition model of the two services and the mediator need to be formally verified. Generally, we consider that the mediation fails if any deadlock exists. Otherwise, the mediation is successful. The rationale of the verifying method relies on searching reachable states. Many approaches to formal verification of service composition and mediation are presented in both our previous work [17] and literatures of other researchers [24]. Details of mediation verification are beyond the scope of this paper.

(5) Code generation of mediators

Only successful mediator will be performed in this step. It is the converse of the first procedure, i.e., transforming CPN models to BPEL-based mediators. To facilitate code generation of executable mediators, BPEL templates for the corresponding mediator patterns are developed. With these BPEL templates, the pseudo-code for protocol mediation can be produced automatically.

3.2. Selection of Mediator Patterns

Mediator pattern selection is a very challenging issue in the sense that mismatches between two partially compatible services should be identified first. The appropriate patterns can be selected once the mismatches are identified. To the best of our knowledge, few of the existing approaches are developed for (semi-)automatically identifying protocol mismatches and selecting appropriate patterns. As the first step towards this challenge, we propose a heuristic technique based on message mapping for semi-automatic selection of mediator patterns. By semi-automation, we mean that service developers should specify the message mappings and adjust the selected patterns. It is noted that automatic specification of message mappings is also a challenging problem in the areas of data integration, schema mapping and semantic-related researches [25-27], which is a separate research thread and beyond the scope of this paper.

Message mapping M between two partial compatible services is a finite set of mapping relations, i.e., $M = \{mr_i\}$. Each mapping relation mr_i is expressed in the form of <*source*, *cnst_s*, *target*, *cnst_t*>, where *source* is a part/element of the sending message and *target* is the corresponding part/element of the receiving message. *source*/*target* is expressed in the form of *Service*.*Message*.*Part*. *cnst_s* is the constraint of the operation that sends *source* and *cnst_t* is the constraint of the operation that receives *target*. *cnst_s*/*cnst_t* can be NULL if there is no constraint with the sending/receiving message. In the motivating example (see Section 1.1), the receiving message *SearchRequest* of S_E has two parts: *login* and *request*. Thus the part *login* is a target and expressed as S_E.sreq.login, where sreq stands for the message *SearchRequest*. For the sake of simplicity, the part name is omitted if the message consists of only one part. For example, the sending message *Login* of S_C has only one part *login*. Thus it is a source and expressed as S_C.login. *source*/*target* can be NULL if the sending/receiving message doesn't

exist. The prefix of *source*/*target* is the message name of *source*/*target*, denoted by *prefix*(*source*/*target*), e.g., *prefix*(S_E.sreq.login) = S_E.sreq and *prefix*(S_C.login) = S_C.login.

Every mapping relation of *M* should relate to a certain message. It is not allowed that both the source and the target of a mapping relation are NULL. For every mapping relation, e.g., mr_i, we thus have the following two formulas:

(i) $source(mr_i) \neq \text{NULL}$, if $target(mr_i) = \text{NULL}$;

(ii) $target(mr_i) \neq \text{NULL}$, if $source(mr_i) = \text{NULL}$.

For every two message mappings, e.g., mr_i and mr_j, the constraints imposed on their sources/targets should be the same if their sources/targets belong to the same message. Thus we have the following two formulas:

(iii) $cnst\_s(mr_i) = cnst\_s(mr_j)$, if $prefix(source(mr_i)) = prefix(source(mr_j))$;

(iv) $cnst\_t(mr_i) = cnst\_t(mr_j)$, if $prefix(target(mr_i)) = prefix(target(mr_j))$.

In terms of the above notation, service developers can specify the message mapping relations, as shown in Table 1.

Table 1 Message Mapping Relations

mapping	**source**	**cnst_s**	**target**	**cnst_t**
mr_1	S_C.login	NULL	S_E.sreq.login	NULL
mr_2	S_C.sreq	NULL	S_E.sreq.request	NULL
mr_3	NULL	NULL	S_C.ack	NULL
mr_4	S_E.partialResult	<*while*> condition(x)	S_C.totalResult	NULL
mr_5	S_E.ntf	condition($\overline{x}$)	NULL	NULL

The first mapping relation (i.e., mr_1) indicates that S_C sends a message login and S_E receives the message as the part login of its message sreq. There is no constraint with the two operations. We denote that $source(mr_1) = S_C$.login and $target(mr_1) = S_E$.sreq.login. In the fourth mapping relation (i.e., mr_4), "<*while*> condition(x)" indicates that the message "S_E.partialResult" is sent iteratively under the condition x. In the fifth mapping relation (i.e., mr_5), "condition($\overline{x}$)" indicates that the message "S_E.ntf" is sent when the condition x doesn't hold. We also denote that $cnst\_s(mr_5)$ = condition($\overline{x}$) and $cnst\_t(mr_5)$ = NULL.

Herein, we introduce a heuristic rule for identifying which mediator pattern should be selected, by using the mapping relations. For two mapping relations, i.e., mr_i and mr_j, the selection rule is as follows:

Selection Rule of Mediator Patterns

(1) **if** $(cnst\_s(mr_i) = cnst\_t(mr_i)) \wedge (prefix(source(mr_i)) = prefix(source(mr_j))) \wedge (prefix(target(mr_i)) = prefix(target(mr_j)))$

then there is no need of mediator patterns;

(2) **else if** $(cnst\_s(mr_i) = cnst\_t(mr_i)) \wedge (target(mr_i) = \text{NULL})$

then a Simple Storer pattern is selected;

(3) **else if** $(cnst\_s(mr_i) = cnst\_t(mr_i)) \wedge (source(mr_i) = \text{NULL})$

then a Simple Constructor pattern is selected;

(4) **else if** $(cnst\_s(mr_i) = cnst\_t(mr_i)) \wedge (prefix(source(mr_i)) = prefix(source(mr_j))) \wedge (prefix(target(mr_i)) \neq prefix(target(mr_j)))$

then a Splitter pattern is selected;

(5) **else if** $(cnst\_s(mr_i) = cnst\_t(mr_i)) \wedge (prefix(source(mr_i)) \neq prefix(source(mr_j))) \wedge (prefix(target(mr_i)) = prefix(target(mr_j)))$

then a Merger pattern is selected;

(6) **else if** $(cnst\_s(mr_i) \neq cnst\_t(mr_i)) \wedge ((cnst\_s(mr_i) = \text{NULL} \wedge source(mr_i) = target(mr_i)) \vee (cnst\_s(mr_i) \neq \text{NULL} \wedge target(mr_i) = \text{NULL}))$

then a Storing Controller pattern is selected;

(7) **else if** $(cnst\_s(mr_i) \neq cnst\_t(mr_i)) \wedge ((cnst\_t(mr_i) = \text{NULL} \wedge source(mr_i) = target(mr_i)) \vee (cnst\_t(mr_i) \neq \text{NULL} \wedge source(mr_i) = \text{NULL})$

then a Constructing Controller pattern is selected;

(8) **else** more complicated mismatches and developers' intervention is needed.

The first part of the selection rule, i.e., sub-rule (1), shows that for two mapping relations, i.e., mr_i and mr_j, their sources belong to the same message, i.e., $prefix(source(mr_i)) = prefix(source(mr_j))$, and their targets belong to the same message, i.e., $prefix(target(mr_i)) = prefix(target(mr_j))$, and the constraints imposed on the source and the target of mr_i are the same, i.e., $cnst\_s(mr_i) = cnst\_t(mr_i)$, then we have the constraints imposed on mr_i and mr_j are all the same, according to Formula (iii) and Formula (iv). Hence, the source message of mr_i and mr_j can be directly related to the target message of mr_i and mr_j without need of mediation.

The second part of the selection rule, i.e., sub-rule (2), shows that a message is sent out by one service but the other service doesn't receive it. In this case, a Simple Storer pattern should be selected. Similarly, the third part of the selection rule, i.e., sub-rule (3), shows that a message is expected to be received by one service but the other service doesn't send it. In this case, a Simple Constructor pattern should be selected.

The fourth part of the selection rule, i.e., sub-rule (4), shows that the sources of two mapping rules, i.e., mr_i and mr_j, belong to the same message, but their targets belong to two different messages. In this case, a Splitter pattern should be selected. Similarly, the fifth part of the selection rule, i.e., sub-rule (5), shows that the sources of two mapping relations belong to different messages, but their targets belong to the same message. In this case, a Merger pattern should be selected.

The sixth part of the selection rule, i.e., sub-rule (6), shows that the Storing Controller pattern should be selected in two cases. In the one case, a message is sent without any constraint but it is received with some constraint imposed on it. In the other case, a message is sent with some constraint imposed on it, but it isn't received by any service.

The seventh part of the selection rule, i.e., sub-rule (7), shows that the Constructing Controller pattern should be selected in two cases. In the one case, a message is sent with some constraint imposed on it but it is received without any constraint. In the other case, a message needs to be received under some constraint, but it isn't sent by any service.

The eighth part of the selection rule, i.e., sub-rule (8), shows that there exist more complicated mismatches between the two service. Usually, developers' intervention is needed to compose some mediator patterns for reconciling complicated mismatches.

Let us consider the motivating example, four mediator patterns can be selected to address the mismatches after performing the selection rule. The selected mediator patterns are given as follows:

i) A Merger pattern is used to receive S_C.login and S_C.sreq from S_C, and then it sends S_E.sreq to S_E, where $S_E.\text{sreq} = S_E.\text{sreq.(login, request)}$. This pattern is selected according to mr_1 and mr_2.

ii) A Simple Constructor pattern is used to construct S_C.ack and send it to S_C. This pattern is selected according to mr_3.

iii) A Merging Repeater pattern is used to iteratively receive S_E.partialResult from S_E until all partial databases are finished according to mr_4. The Merging Repeater merges all partial results

together and sends S_C.totalResult to S_C. Since mr_4 corresponds to a complicated mismatch with iterative structure, the Merging Repeater pattern can be selected by service developers. It is composed by two storing Controller patterns and compensates the mismatch (see Section 2.2).

iv) A Storing Controller pattern is used to conditionally store S_E.ntf that is sent by S_E. This pattern is selected according to mr_5.

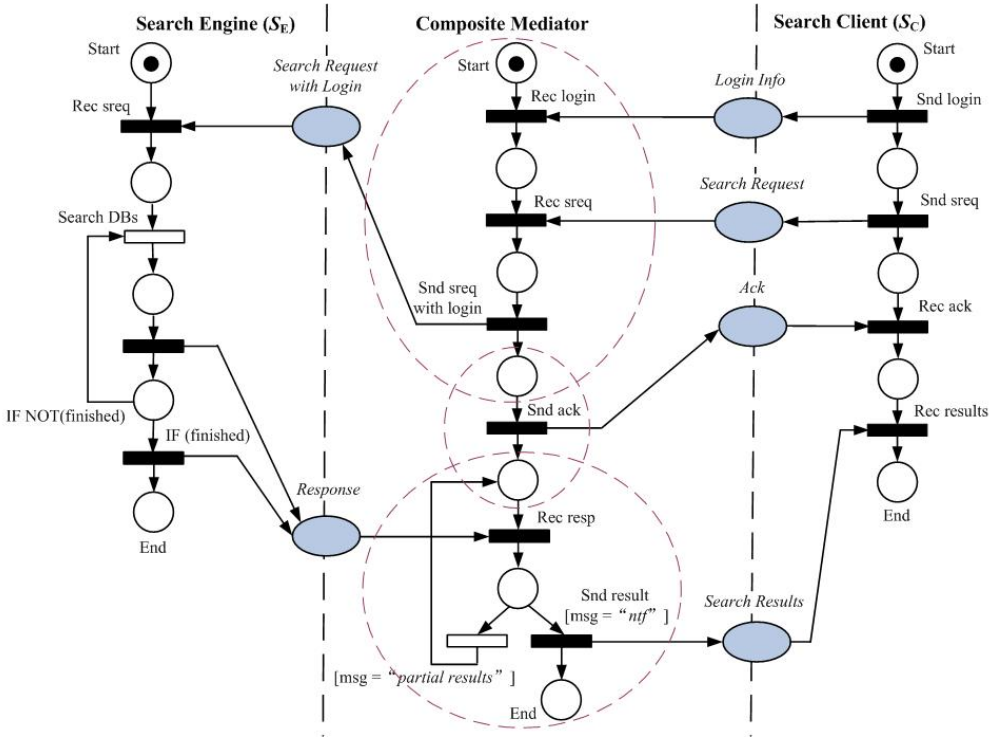


Fig. 12. A composite mediator for the composition of S_E and S_C

As mentioned in Section 2.2 and 3.1, service developers should configure the structures and control logics of the selected mediator patterns and compose them together. In the motivating example, the Merging Repeater pattern can successfully compensate the mismatch with iterative structure and there is no need for another Storing Controller pattern. Thus three mediator patterns are eventually selected for mediation, that is, a merger, a simple constructor and a merging repeater. As shown in Fig. 12, a composite mediator composed by the above three mediator patterns sits between the two interacting services (i.e., S_E and S_C) and reconciles their mismatches. The three mediator patterns are circled with dashed ellipses. Since the protocols of S_E, S_C and mediators are modeled by the CPN formalism, it is easy to verify that S_E and S_C can successfully interact through the composite mediator.

3.3. BPEL Templates of Mediator Patterns

Both basic and composite mediators developed in the above steps are conceptual patterns depicted by the CPN formalism, rather than executable codes. As a further step towards generating executable codes of mediators, corresponding BPEL templates are provided for the mediator patterns. With these BPEL templates, the pseudo-code (i.e., BPEL codes) for protocol mediation can be generated automatically. Each mediator pattern has its corresponding BPEL template. In the following, we will present the BPEL templates of Simple Constructor pattern, Splitter pattern and Storing Controller pattern. Others can be found in the Appendix.

(1) BPEL template of Simple Constructor pattern

Simple Constructor pattern constructs and sends a message. It is used for reconciling mismatches of extra receiving messages and missing sending messages. When creating a message, Simple Constructor pattern invokes a creator service for constructing the message. It should be pointed out that how to construct a message of certain type is a non-trivial task and some evidences can be used to address the issue [28].

```
<sequence>
    <invoke name="creating" partnerLink="creator" portType="..."
            operation="creatMsg" inputVariable="creatingMsg"
            outputVariable="createdMsg">
    </invoke>
    <reply variable=" createdMsg" name="..." partnerLink="..."
           portType="..." operation="...">
    </reply>
</sequence>
```

(2) BPEL template of Splitter pattern

Splitter pattern receives a single message and splits it into two or more partial messages. It is used for reconciling mismatches of splitting sending messages and merging receiving messages. The specific structure of Splitter pattern is adjustable according to the sequence of partial messages which may be sequential, parallel or mixed structure (see Section 2.2). Herein, the BPEL template of the Splitter pattern with two sequential partial messages is given. It is similar to develop more complex Splitter pattern.

```
<sequence>
    <receive variable="splitter_receiver" name="..."
             partnerLink="..." portType="..." operation="...">
    </receive>
    <assign>
        <copy>
            <from part="part1" variable="splitter_receiver" />
            <to part="part" variable="splitter_partialMsg1" />
        </copy>
        <copy>
            <from part="part2" variable="splitter_receiver" />
            <to part="part" variable="splitter_partialMsg2" />
        </copy>
    </assign>
    <reply variable="splitter_partialMsg1" name="..."
           partnerLink="..." portType="..." operation="...">
    </reply>
    <reply variable="splitter_partialMsg2" name="..."
           partnerLink="..." portType="..." operation="...">
    </reply>
</sequence>
```

(3) BPEL template of Storing Controller pattern

Storing Controller pattern receives and stores a message and then conditionally sends the message in terms of specific logic. It is used for reconciling mismatches of extra condition of receiving messages and missing condition of sending messages.

```
<sequence>
      <receive variable="msgName" name="..."
                partnerLink="..." portType-"..." operation="...">
      </receive>
      <switch>
            <case codition="getVariableData(...)">
                  <reply variable="msgName" name="..."
                        partnerLink="..." portType="..." operation="...">
                  </reply>
            </case>
      </switch>
      <otherwise>
            ...
      </otherwise>
</sequence>
```

It should be pointed out that the BPEL templates developed above are pseudo-codes for protocol mediation. To get executable BPEL codes for deployment, service developers needs to do further refinement on the pseudo-codes. For example, service developers should specify the definitions of appropriate variables, operations, partnerLinks, portTypes, etc.

4. Prototype Implementation

4.1. Architecture of Prototype System

The systematic approach presented in this paper has been implemented in the Service Mediation Tookit (SMT), as shown in Fig. 13. SMT has been implemented on top of IBM Websphere Integration Developer which is an eclipse-based IDE for development of composite applications based on Service Component Architecture (SCA). Main components of the toolkit are introduced in the following.

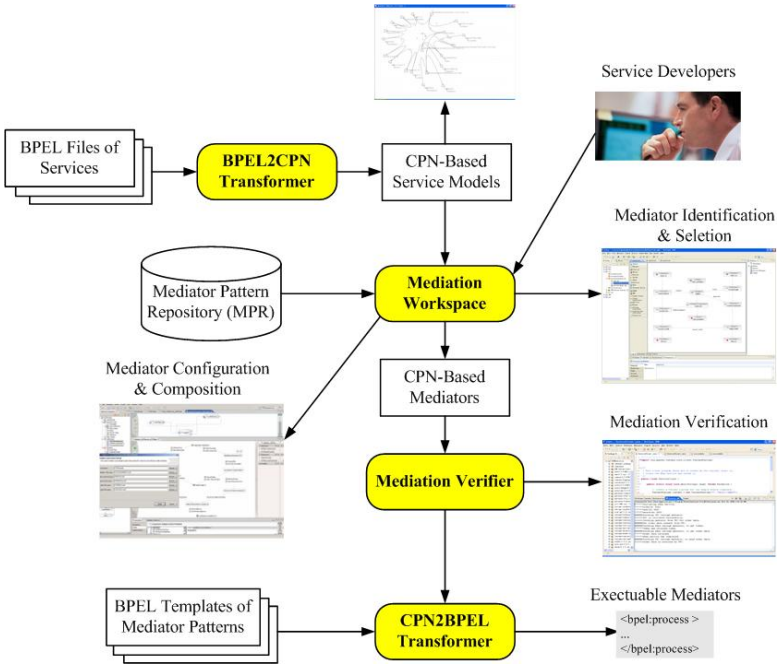


Fig. 13. Architecture of Service Mediation Toolkit (SMT)

(1) BPEL2CPN Transformer

Web services to be composed together are specified in BPEL files and wrapped as SCA components. BPEL-based services are transformed to CPN-based service models through a separate tool, namely BPEL2CPN Transformer. Recently, existing tools have provided similar functionalities, e.g., BPEL2PNML[5].

(2) Mediation Workspace

The Mediation Workspace is the core component of our mediation toolkit and provides a user-friendly workbench for developers to manipulate services and mediators. Although mediator patterns are depicted by using CPN models as an underlying formalism, the protocols of services and mediators are graphically represented by means of an intuitional notation, like Business Process Modeling Notation (BPMN). The Mediation Workspace provides a GUI to illustrate the protocols of the two services to be composed. Service developers specify the message mapping relations between the two partially compatible services and provide the mapping relations to Mediation Workspace as the input. We have pre-established several basic mediator patterns that are stored in a certain repository, i.e., Mediator Pattern Repository (MPR). The basic mediator patterns are well-defined and can be used as building blocks to construct complex mediators. Composite mediators can also be stored as patterns in MPR for further use. MPR provides the functionality of flexible extension for mediator patterns. By means of the selection rule (see Section 3.2), appropriate mediator patterns are identified and selected from MPR. Service developers configure the selected patterns if needed. After that, the selected mediator patterns are composed to produce a composite mediator. The composite mediator is also depicted as the intuitional notation with underlying CPN models, which is automatically constructed in the Mediation Workspace.

(3) Mediation Verifier

The service mediator produced in the above steps may not successfully compensate all protocol mismatches and deadlocks may exist. To make sure the mediation successful, services and the produced mediator are composed together to be a composite CPN model. Mediation Verifier checks whether any deadlock may occur. Formal approaches/algorithms for protocol mediation developed in our previous work [17] are implemented by Mediation Verifier.

(4) CPN2BPEL Transformer

Only successful mediator will be performed on the CPN2BPEL. BPEL templates of mediator patterns (see Section 3.3) are utilized for generating mediation codes. Note that the BPEL-based mediator obtained as the output of CPN2BPEL is only pseudo-code of BPEL files. Service developers should refine the pseudo-code and generate executable codes.

4.2. Implementation of Mediation Workspace

As a core component of SMT, Mediation Workspace is a separate tool and provides a GUI workbench for service developers to manipulate services and mediators. We have implemented Mediation Workspace based on an open-source project, i.e., jBPM jPDL Process Designer[6]. jPDL is a process language that is built on top of a flexible and extensible framework for process languages. jPDL Process Designer is an eclipse plugin application. Thus Mediation Workspace is also developed as an eclipse plugin that is easy to be integrated with other eclipse-based applications.

---

[5] http://www.bpm.fit.qut.edu.au/projects/babel/tools/

[6] http://docs.jboss.org/jbpm/v3/userguide/index.html

Fig. 14 shows a screenshot of Mediation Workspace for the motivating example. Basic mediator patterns and the Merging Repeater pattern have been developed and placed on the left toolbar. To glue S_E and S_C together, a mediation project (i.e., Search Service Mediation) is created. By using the message mapping relations as specified in Table 1 (see Section 3.2), a composite mediator, consisting of a merger pattern, a simple constructor pattern and a merging repeater pattern, is constructed and put between the two services. The composite mediator reconciles the protocol mismatches between them. The BPEL file of the composite mediator is generated in the mediation project.

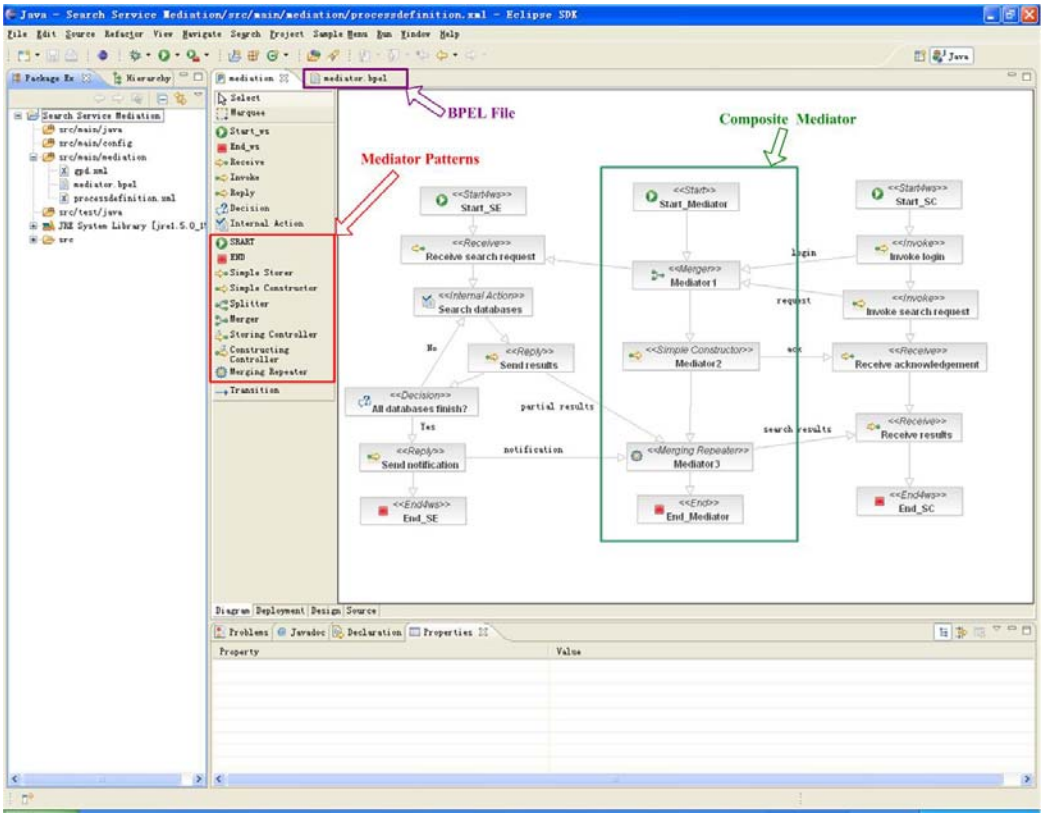


Fig. 14. Screenshot of the Mediation Workspace

5. Related Work

A large number of research works have been developed for service mediation with the purpose of addressing various kinds of composition mismatches [10]. Signature mediation has received considerable attention [12, 13] and many commercial tools have been developed, e.g., Microsoft BizTalk Mapper, Stylus Studio XML Mapping Tools and SAP XI Mapping Editor. However, protocol mediation is still a challenging issue and needs further research [15, 28].

Several formal approaches have been developed to conquer this challenge, such as Automata [15], Process Algebra [29] and Petri nets [17, 24], etc. A methodology is developed in [30] for the automated generation of adapters (i.e., mediators) that is capable to solve protocol mismatches among BPEL processes. The work presented in [28] identifies a few ordering mismatches and provides a semi-automated support to reconcile these mismatches. A technique based on schema matchmaking is used to handle the issue of message mapping. The existing approaches, however, provide only partial solutions and few of them can sufficiently address all possible protocol mismatches. Particularly,

mediators developed by these approaches have no control logics and therefore can not reconcile complicated protocol mismatches, such as mismatches of extra condition, missing condition, or iterative structure [20].

It has been recognized that patterns can be used to reconcile composition mismatches and address protocol mediation [10, 16, 31]. In [16], Benatallah et al. identify five mismatch patterns and provide templates of BPEL code for service developers to build appropriate mediators, but these patterns are not sufficient. Although two more protocol mismatches derived from repetition structures, namely *Collapse* and *Burst*, are introduced in [8], no approach is proposed to address the two types of mismatches. Similarly, Pokraev and Reichert summarize several typical protocol mismatches and propose appropriate mediation patterns to compensate them [31]. These patterns are still insufficient and not delicately designed for further manipulation, such as composition, verification or code generation. The taxonomy of composition mismatches is proposed in [10] and a selection of patterns is presented to eliminate these mismatches. The taxonomy, however, does not sufficiently address protocol mismatches. Moreover, the problem of generating mediation codes is not discussed.

Another significant work is presented in [32]. The authors discuss the possible mismatches of service composition and propose a general approach that aims to assist service developers for resolving these mismatches. The focus of their work is on identifying mismatches at the syntactic and/or semantic level, e.g., signatures and data types, rather than protocol mismatches. And the approach can not be used to generate executable mediation codes automatically.

Inspired by [16, 32], our approach is significantly different from the existing works in the following aspects. Firstly, the mediator patterns presented in this paper are derived from our comprehensive identification of protocol mismatches and can be used to sufficiently address those mismatches [20]. The configurability and composability of mediators are first investigated. Secondly, a formal modeling method (i.e., CPN models) is adopted as an underlying formalism for depicting the protocols of both services and mediators. The CPN-based formalism can not only depict the internal logics and message exchanging sequences, but also support solid verification of protocol mediation. Thirdly, the approach presented herein can be used to automatically generate executable codes of the mediators. Lastly, to the best of our knowledge, there exists neither comprehensive solution nor appropriate software tool to support protocol mediation. Our approach can be referred to as a systematic solution and the implemented prototype system, i.e., Service Mediation Toolkit (SMT), can assist service developers to ease their efforts on mediation tasks, such as selecting mediator patterns and generating mediation codes.

6. Conclusion

Service mediation is one of the most essential components of Enterprise Service Bus (ESB) and thus becomes one key working area in SOA. In this paper, we have proposed a systematic approach to protocol mediation for Web services composition. The approach involves several basic mediator patterns and their composition as well. The major advantage of the pattern-based approach lies in that it can be used to successfully reconcile all possible protocol mismatches in an engineering way, especially such mismatches with complicated control logics. A technique based on message mapping is developed for identifying protocol mismatches and selecting appropriate mediator patterns. We have developed BPEL templates of the mediator patterns which are used to generate executable codes of mediators. Furthermore, a prototype system, namely Service Mediation Toolkit (SMT), has been

implemented to validate the feasibility and effectiveness of our approach.

Our future work will focus on the following two aspects. On the one hand, message mapping relations are specified by service developers in the current approach. In some complicated situations, service developers' intervention is needed to select the mediator patterns. The challenge is that current Web services standards (e.g. WSDL, BPEL, etc.) lack of semantic specifications. The next step is to utilize existing techniques developed by semantic Web initiatives for promoting the automation of specifying message mappings and selecting mediator patterns. On the other hand, we plan to improve our prototype system (i.e., SMT) for addressing more general and real-world cases.

Acknowledgement

This work was supported by National Natural Science Foundation of China (No. 60674080 and No. 60704027), National High-Tech R&D (863) Plan of China (No. 2006AA04Z151 and No. 2007AA04Z150) and National Basic Research Development (973) Program of China (No. 2006CB705407).

References

[1] M. P. Papazoglou, and W. J. van den Heuvel, "Service oriented architectures: approaches, technologies and research issues," *The International Journal on Very Large Data Bases*, vol. 16, no. 3, pp. 389-415, 2007.

[2] M. P. Papazoglou, P. Traverso, S. Dustdar *et al.*, "Service-Oriented Computing: State of the Art and Research Challenges," *IEEE Computer*, vol. 40, no. 11, pp. 38-45, 2007.

[3] Q. Yu, X. Liu, A. Bouguettaya *et al.*, "Deploying and managing Web services: issues, solutions, and directions," *The International Journal on Very Large Data Bases*, vol. 17, no. 3, pp. 537-572, 2008.

[4] S.-M. Huang, Y.-T. Chu, S.-H. Li *et al.*, "Enhancing conflict detecting mechanism for Web Services composition: A business process flow model transformation approach," *Information and Software Technology*, vol. 50, no. 11, pp. 1069-1087, 2008.

[5] Z. Maamar, "On coordinating personalized composite web services," *Information and Software Technology*, vol. 48, no. 7, pp. 540-548, 2006.

[6] C. Wu, and E. Chang, "An Analysis of Web Services Mediation Architecture and Pattern in Synapse," *Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops-Volume 01*, pp. 1001-1006, 2007.

[7] M. T. Schmidt, B. Hutchison, P. Lambros *et al.*, "The Enterprise Service Bus: Making service-oriented architecture real," *IBM Systems Journal*, vol. 44, no. 4, pp. 781-797, 2005.

[8] M. Dumas, M. Spork, and K. Wang, "Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation," *Proc. of the 4th Intl. Conf. on Business Process Management (BPM 2006)*, pp. 65-80, 2006.

[9] C. Canal, P. Poizat, and G. Salaun, "Adaptation of Component Behaviour using Synchronous Vectors," Technical Report ITI-05-10, 2005.

[10] S. Becker, A. Brogi, I. Gorton *et al.*, "Towards an Engineering Approach to Component Adaptation," *Architecting Systems with Trustworthy Components*, vol. 3938, pp. 193–215, 2006.

[11] I. G. Kim, D. H. Bae, and J. E. Hong, "A component composition model providing dynamic,

flexible, and hierarchical composition of components for supporting software evolution," *Journal of Systems and Software*, vol. 80, no. 11, pp. 1797-1816, 2007.

[12] A. M. Zaremski, and J. M. Wing, "Signature matching: a tool for using software libraries," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 4, no. 2, pp. 146-170, 1995.

[13] S. R. Ponnekanti, and A. Fox, "Interoperability among independently evolving web services," *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pp. 331-351, 2004.

[14] M. Szomszor, T. R. Payne, and L. Moreau, "Automated Syntactic Medation for Web Service Integration," *Proceedings of the International Conference on Web Services, Chicago, IL*, 2006.

[15] D. M. Yellin, and R. E. Strom, "Protocol specifications and component adaptors," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 19, no. 2, pp. 292-333, 1997.

[16] B. Benatallah, F. Casati, D. Grigori *et al.*, "Developing Adapters for Web Services Integration," *Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE)*, 2005.

[17] W. Tan, Y. S. Fan, and M. C. Zhou, "A petri net-based method for compatibility analysis and composition of Web services in business process execution language," *IEEE Transactions on Automation Science and Engineering (in Press)*, 2008.

[18] X. Li, Y. Fan, J. Wang *et al.*, "A Pattern-Based Approach to Development of Service Mediators for Protocol Mediation," *Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)-Volume 00*, pp. 137-146, 2008.

[19] F. Jiang, Y. Fan, and X. Zhang, "Rule-Based Automatic Generation of Mediator Patterns for Service Composition Mismatches," *Grid and Pervasive Computing Workshops, 2008. GPC Workshops' 08. The 3rd International Conference on*, pp. 3-8, 2008.

[20] X. Li, Y. Fan, and F. Jiang, "A Classification of Service Composition Mismatches to Support Service Mediation," *Proc. of the 6th Intl. Conf. on Grid and Cooperative Computing (GCC 2007)*, pp. 315-321, 2007.

[21] K. Jensen, "Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Vol. 1, Basic Concepts," *Monographs in Theoretical Computer Science. Springer-Verlag*, 1997.

[22] C. Ouyang, E. Verbeek, W. M. P. van der Aalst *et al.*, "Formal semantics and analysis of control flow in WS-BPEL," *Science of Computer Programming*, vol. 67, no. 2-3, pp. 162-198, 2007.

[23] W. M. P. van der Aalst, and K. Bisgaard Lassen, "Translating unstructured workflow processes to readable BPEL: Theory and implementation," *Information and Software Technology*, vol. 50, no. 3, pp. 131-159, 2008.

[24] A. Martens, S. Moser, A. Gerhardt *et al.*, "Analyzing Compatibility of BPEL Processes," *Telecommunications, 2006. AICT-ICIW'06. International Conference on Internet and Web Applications and Services/Advanced International Conference on*, pp. 147-147, 2006.

[25] E. Rahm, and P. A. Bernstein, "A survey of approaches to automatic schema matching," *The International Journal on Very Large Data Bases*, vol. 10, no. 4, pp. 334-350, 2001.

[26] P. Shvaiko, and J. Euzenat, "A survey of schema-based matching approaches," *Journal on Data Semantics*, vol. 4, pp. 146-171, 2005.

[27] B. Spencer, and S. Liu, "Inferring Data Transformation Rules to Integrate Semantic Web

Services," *International Semantic Web Conference*, pp. 456–470, 2004.

[28] H. R. M. Nezhad, B. Benatallah, A. Martens *et al.*, "Semi-automated adaptation of service interactions," *Proceedings of the 16th international conference on World Wide Web*, pp. 993-1002, 2007.

[29] A. Bracciali, A. Brogi, and C. Canal, "A formal approach to component adaptation," *The Journal of Systems & Software*, vol. 74, no. 1, pp. 45-54, 2005.

[30] A. Brogi, and R. Popescu, "Automated Generation of BPEL Adapters," *Service-Oriented Computing-ICSOC 2006, 4th International Conference, Chicago, IL, USA, December 4-7, 2006, Proceedings*, vol. 4294, pp. 27–39, 2006.

[31] S. Pokraev, and M. Reichert, "Mediation Patterns for Message Exchange Protocols " *Open INTEROP-Workshop on Enterprise Modeling and Ontologies for Interoperability (EMOI06) in Proc. CAiSE*, vol. 6, pp. 659-663, 2006.

[32] M. Younas, K. M. Chao, and C. Laing, "Composition of mismatched web services in distributed service oriented design activities," *Advanced Engineering Informatics*, vol. 19, no. 2, pp. 143-153, 2005.

Appendix

The BPEL templates of Simple Storer pattern, Merger pattern, Constructing Controller pattern and Merging Repeater pattern are presented in the appendix as follows.

(1) BPEL template of Simple Storer pattern

Simple Storer pattern receives and stores a message. It is used for reconciling mismatches of extra sending messages and missing receiving messages.

```
<sequence>
    <receive variable="msgName" name="..."
             partnerLink="..." portType="..." operation="...">
    </receive>
</sequence>
```

(2) BPEL template of Merger pattern

Merger pattern receives two or more partial messages and merges them into a single one. It is used for reconciling mismatches of splitting receiving messages and merging sending messages. Similar to Splitter pattern, the specific structure of Merger pattern is adjustable according to the sequence of merged messages which may be sequential, parallel or mixed structure. Herein, the BPEL template of the Merger pattern with two sequential partial messages is given. It is similar to develop more complex Merger pattern.

```
<sequence>
    <receive variable="merger_receiver1" name="..."
            partnerLink="..." portType="..." operation="...">
    </receive>
    <receive variable="merger_receiver2" name="..."
            partnerLink="..." portType="..." operation="...">
    </receive>
    <assign>
        <copy>
            <from part="part" variable="merger_receiver1" />
            <to part="part1" variable="merger_sender" />
        </copy>
        <copy>
            <from part="part" variable="merger_receiver2" />
            <to part="part2" variable="merger_sender" />
        </copy>
    </assign>
    <reply variable="merger_sender" name="..."
          partnerLink="..." portType="..." operation="...">
    </reply>
</sequence>
```

(3) BPEL template of Constructing Controller pattern

Constructing Controller pattern conditionally constructs and sends a message in terms of specific logic. It is used for reconciling mismatches of extra condition of sending messages and missing condition of receiving messages.

```
<sequence>
    <switch>
        <case codition="getVariableData(...)">
            <receive variable="msgName" name="..."
                    partnerLink="..." portType-"..." operation="...">
            </receive>
            <reply variable="msgName" name="..."
                   partnerLink="..." portType-"..." operation="...">
            </reply>
        </case>
        <case condition="getVariableData(...)">
            <invoke name="creating" partnerLink="creator"
                    portType="..." operation="creatMsg"
                    inputVariable="creatingMsg" outputVariable="createdMsg">
            </invoke>
            <reply variable="createdMsg" name="..."
                  partnerLink="..." portType="..." operation="...">
            </reply>
        </case>
    </switch>
    <otherwise>
        ...
    </otherwise>
</sequence>
```

(4) BPEL template of Merging Repeater pattern

Merging Repeater pattern receives messages iteratively and merges the received messages

together under certain condition. When the condition doesn't hold, it sends the whole merged message to its partner. Merging Repeater pattern can be used for reconciling protocol mismatches with iterative structure.

```
<sequence>
      <while name="…">
            <condition expressionLanguage="...">
                  conditionExpression
            </condition>
            <receive variable="message1" name="..."
                        partnerLink="..." portType="..." operation="...">
            </receive>
            <assign>
                  <copy>
                        <from part="part" variable="message1" />
                        <to part="part{$count}" variable="message2" />
                  </copy>
                  <copy>
                        <from>($count + 1)</from>
                        <to variable="count"/>
                  </copy>
            </assign>
      </while>
      <reply variable="message2" name="..."
               partnerLink="..." portType="..." operation="...">
      </reply>
</sequence>
```