Information and Software Technology 52 (2010) 304–323

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

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, PR 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

ARTICLE INFO

Article history:
Received 6 September 2008
Received in revised form 2 November 2009
Accepted 4 November 2009
Available online 11 November 2009

Keywords:
Service oriented architecture
Web service
Service composition
Protocol mediation
BPEL

ABSTRACT

Context: 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 cannot be directly composed. Service mediation, roughly classified into signature and protocol ones, thus becomes one key working area in SOA.
Objective: As a challenging problem, protocol mediation is still open and existing approaches only provide partial solutions. Further investigation on a systematic approach is needed.
Methods: In this paper, an 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.
Results: A prototype system, namely Service Mediation Toolkit (SMT), has been implemented to validate the feasibility and effectiveness of the proposed approach.
Conclusion: The approach along with the prototype system facilitate the existing practice of protocol mediation for Web services composition.

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]. In real-world situations, however, independently-developed services are not always exactly compatible and cannot be straightly composed together.

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 [6,7]. First proposed in the Enterprise Service Bus (ESB) industry community [8], 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 [9]. Nowadays, service mediation has become a key working area in the field of SOA and Component-Based Software Engineering (CBSE) [10–12].

Service mediation can be roughly classified into *signature* and *protocol*. Signature mediation, which focuses on message types, has received considerable attention [13,14] 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 problem is to develop a mediator/adapter which is a piece of code that sits between the interacting services and reconciles the mismatches [15,16]. However, the mediators developed by existing approaches have no control logics and cannot resolve complex mismatches. Few of these approaches can generate executable code of the mediators. Additionally, no existing approach provides a comprehensive solution to protocol mediation for Web services composition. Last but not least, there is a lack of user-friendly GUI tools that assist developers to alleviate their efforts on mediation

* Corresponding author.
E-mail address: lxt04@mails.tsinghua.edu.cn (X. Li).

[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.

0950-5849/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2009.11.002

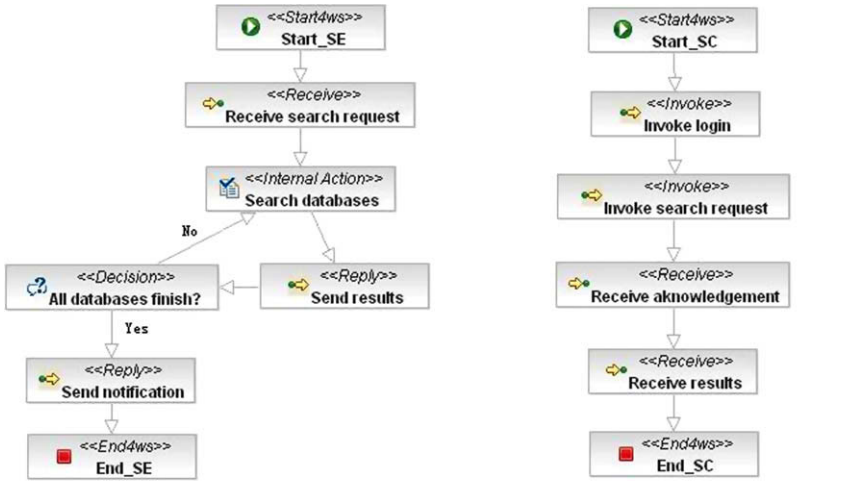


Fig. 1. A motivating example of service composition with protocol mismatches.

tasks, such as identifying protocol mismatches or generating mediation code. The work of this paper aims at addressing these issues.

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.

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. Then, S_C waits for the acknowledgement and 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 items, it sends these items to S_C immediately. In case 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. In this paper, we take BPEL as the specification language for describing service protocols. Fig. 1 shows the BPEL protocols of 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, S_E and S_C cannot interact with each other successfully.

1.2. Contributions

The rationale of our work has been presented in several conference papers [17–20]. As an extension of our previous work, we aim at developing a systematic approach to semi-automatically generating mediators for reconciling protocol mismatches. The main contributions are as follows:

(1) We present several basic mediator patterns which are derived from basic protocol mismatches. With the knowledge of protocol mismatches, the basic mediator patterns can be configured and composed by developers to construct advanced mediators and reconcile all possible protocol mismatches. Thus, the basic mediator patterns are referred to as a sufficient set of building blocks.
(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 need to be specified by developers. By using the technique, basic mediator patterns are selected according to the identified protocol mismatches.
(3) We develop BPEL templates for the mediator patterns which can be used to generate executable mediation code. 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 protocol mismatches. The approach combines our work on identification of protocol mismatches, selection of mediator patterns and code generation of BPEL-based mediation.
(5) We develop a prototype system, namely *Service Mediation Toolkit* (*SMT*), which provides a user-friendly workbench and assist service developers to alleviate their mediation tasks. We use SMT to deal with several case studies in order to validate the feasibility and effectiveness of our approach.

The rest of the paper is structured as follows. In Section 2, several categories of basic protocol mismatches, referred to as basic mismatch patterns, and their composability are presented. In Section 3, basic mediator patterns are proposed to resolve the basic protocol mismatches. The configurability and composability of the mediator patterns are described in this section as well. The proposed solution to protocol mediation is presented in Section 4. 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. In Section 5, the prototype system SMT and the validation with a real-world case study are described. Related work is discussed in Section 6. Finally, Section 7 presents the conclusion and the future work.

2. Protocol mismatch patterns

2.1. Basic mismatch patterns

Protocol mismatches refer to such mismatches that occur between the message exchanging sequences of the interacting services. The existing work has identified several categories of protocol mismatches [21]. However, few paper claims its identification is complete in any sense.

To a certain extent, the dependencies of message exchanging sequences between services are similar to their control flows. We thus use modeling modules for control flows to depict the dependencies of the message exchanging sequences. In the field of workflows, four basic workflow patterns have been presented in [22], i.e., *sequence*, *parallel*, *exclusive choice* and *iteration*. Advanced workflow constructs are supposed to be composed using basic workflow patterns. Derived from basic workflow patterns and message exchanging sequences, we propose six basic mismatch patterns so that a comprehensive identification of protocol mismatches can be achieved. In this paper, mismatch patterns refer to those mismatch categories that can be reused to identify protocol mismatches between Web services. We have illustrated that the six basic mismatch patterns can be considered as basic constructs of all possible protocol mismatches [17]. For example, protocol mismatches in the motivating example can be composed by the basic mismatch patterns.

Herein, we represent protocol mismatches from the unsubstitutability perspective, which is based on the scenario that the required interface cannot be exactly substituted by the provided interface. For the sake of illustration, NULL ACTIVITY is used to represent such an activity that sends/receives no messages. Basic mismatch patterns are presented as follows.

2.1.1. Mismatches of extra messages

Description: The provided interface has some extra messages that the required interface does not expect to send/receive.

Illustration: After receiving purchase order, the provided interface sends receipt while the required interface expects no such message, as shown in Fig. 2a.

2.1.2. Mismatches of missing messages

Description: The provided interface does not have some messages that the required interface expects to send/receive.

Illustration: After receiving purchase order, the provided interface does not send receipt while the required interface expects to send receipt, as shown in Fig. 2b.

2.1.3. Mismatches of splitting messages

Description: The provided interface has some messages that the required interface expects to split to send/receive.

Illustration: After receiving purchase order, the provided interface sends the information of product availability and quote in a single message, while the required interface sends product availability ahead of product quote, as shown in Fig. 2c.

2.1.4. Mismatches of merging messages

Description: The provided interface has some messages that the required interface expects to merge to send/receive.

Illustration: After receiving purchase order, the provided interface sends the information of product availability ahead of product quote, while the required interface expects to send product availability and quote in a single message, as shown in Fig. 2d.

2.1.5. Mismatches of extra conditions

Description: The provided interface has some extra conditions imposed on the control flow of its protocol while the required interface expects no conditions constraining its control flow.

Illustration: After receiving payment of the purchase order, the provided interface sends an invoice if the total sum is greater than 1000 USD. However, the required interface expects to directly send an invoice after receiving payment, as shown in Fig. 2e.

2.1.6. Mismatches of missing conditions

Description: The provided interface has no conditions imposed on the control flow of its protocol while the required interface expects to have some conditions constraining its control flow.

Illustration: After receiving payment of the purchase order, the provided interface directly sends an invoice, while the required interface expects to do it under the condition that the total sum is greater than 1000 USD, as shown in Fig. 2f.

2.2. Composability of mismatch patterns

Complex workflow patterns are supposed to be constructed by the four basic workflow patterns (i.e., *sequence*, *parallel*, *exclusive choice* and *iteration*) [22] and the basic mismatch patterns are derived from these workflow patterns. Correspondingly, complex protocol mismatch patterns can be also composed from the basic mismatch patterns. From this perspective, we consider the set of basic mismatch patterns is comprehensive. In this section, we take two complex mismatch patterns, namely *ordering of messages mismatch* and *missing exclusive choice mismatch*, as examples which are derived from the basic workflow patterns *sequence* and *exclusive choice*, respectively. Further, protocol mismatches, which are called *Collapse* and *Burst* in [9] and derived from the workflow pattern *iteration*, can be also composed in a similar way.

The ordering of messages mismatch refers to the situation that the provided interface has two messages in a different order from that of the required interface. As shown in Fig. 3a, the provided interface successively sends product availability and product quote, but the required interface expects to send product quote followed by product availability. This mismatch pattern can be considered as the composition of extra message pattern and missing message pattern, as illustrated in Fig. 4.

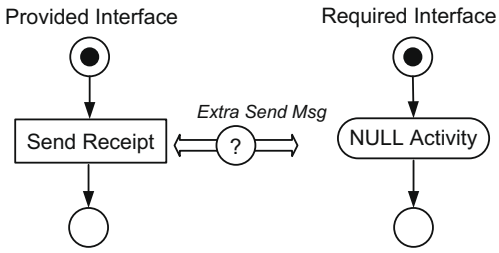
The missing exclusive choice mismatch refers to the situation that the provided interface has no condition imposed on its protocol for sending (or receiving) a message, but the required interface only sends (or receives) the message under certain condition. In the other case the required interface takes other actions. As shown in Fig. 3b, the provided interface sends invoice with no restraint. Differently, the required interface has some choices on its protocol, that is, it sends invoice only if the payment sum is greater or equal to 1000 USD. The required interface sends receipt in case of small payment less than 1000 USD. Fig. 5 illustrates that the missing exclusive choice mismatch can be composed by the missing condition pattern and missing message pattern.

3. Protocol mediator patterns

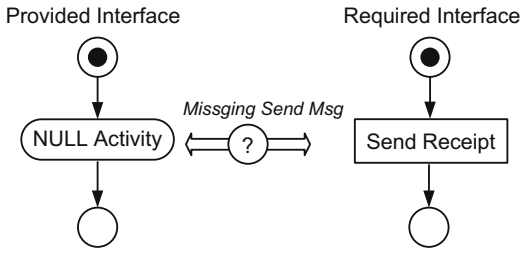
3.1. Basic mediator patterns

An effective solution to reconciling protocol mismatches is to develop mediators. We have developed six mediators to reconcile the basic mismatch patterns. It has been pointed out that the basic mediators can be referred to as basic design patterns which assist service developers to modularly construct more complex mediators and reconcile all possible protocol mismatches [18]. Hence, the set of basic mediator patterns is considered to be comprehensive. A composite mediator can be treated as a new pattern for further reuse. To make this paper self-contained, we present the six basic mediator patterns and corresponding using scenarios in this section.

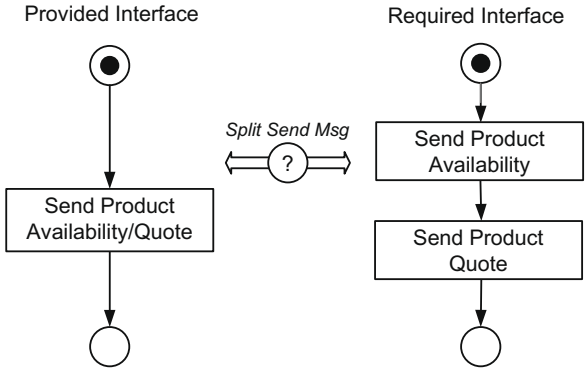
Note that the protocols of both Web services and mediators are depicted based on Colored Petri Nets (CPN) [23]. The benefit of adopting CPN models as an underlying formalism lies in that they provide rich analysis capability to support formal verification of protocol mediation and solid approaches to the transformation between BPEL and CPN models [24,25]. Details of CPN models can be



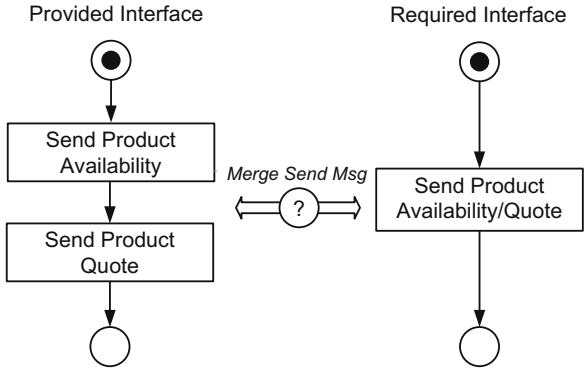
(a) Extra message pattern



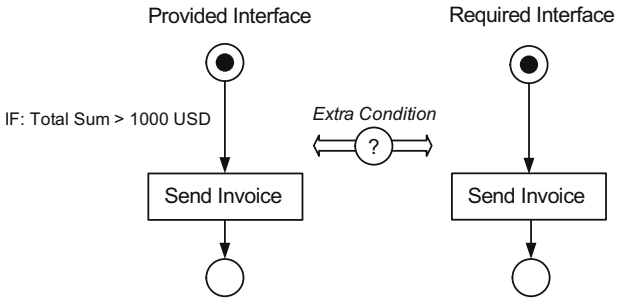
(b) Missing message pattern



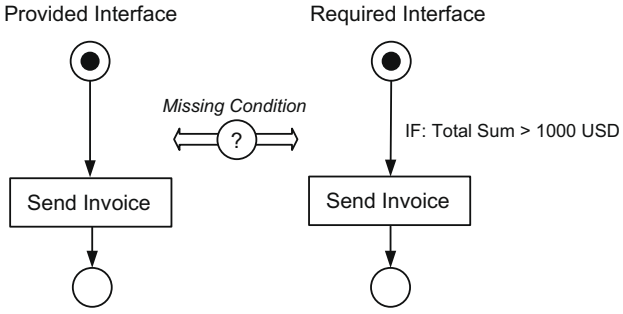
(c) Splitting message pattern



(d) Merging message pattern

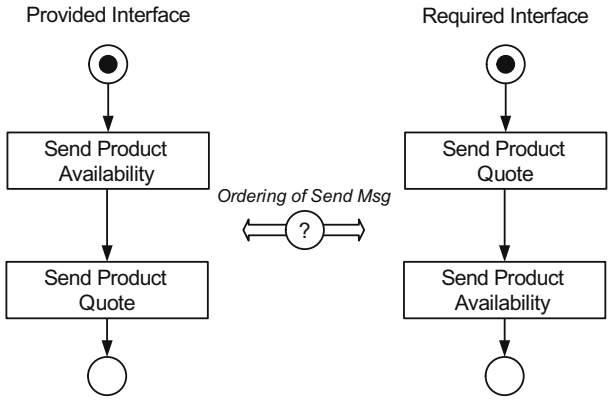


(e) Extra condition pattern

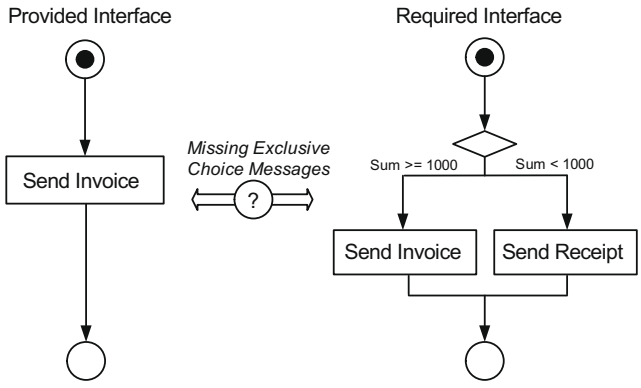


(f) Missing condition pattern

Fig. 2. Basic protocol mismatch patterns.



(a) Ordering of messages mismatch



(b) Missing exclusive choice mismatch

Fig. 3. Two complex protocol mismatches.

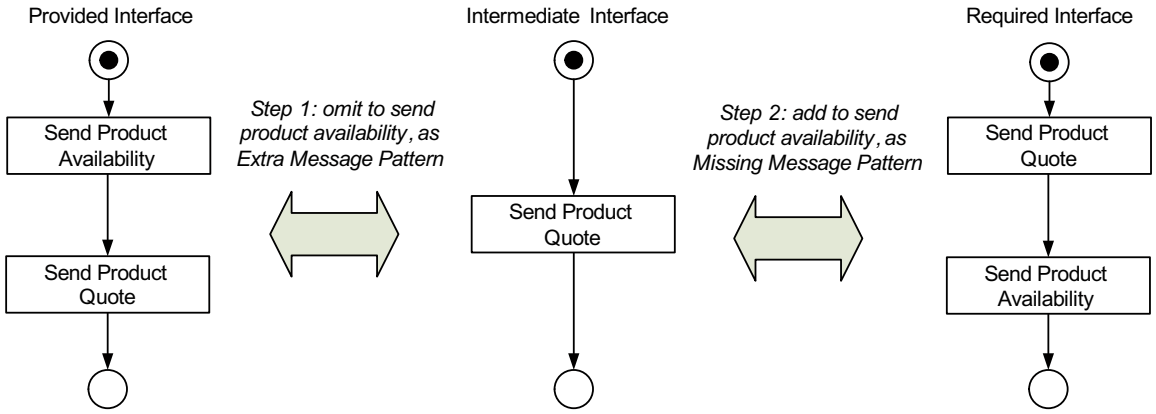


Fig. 4. Composition steps of ordering of messages mismatch.

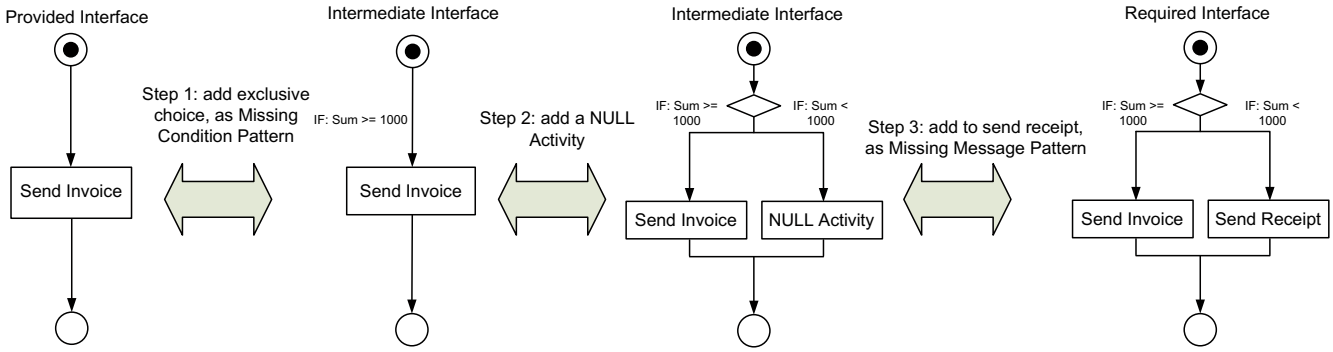


Fig. 5. Composition steps of missing exclusive choice mismatch.

found in [23]. 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.

3.1.1. Simple Storer pattern

Description: A mediator with the capability of simply receiving and storing messages of certain specific type.

Illustration: The Simple Storer pattern can be used to resolve mismatches of extra sending messages and missing receiving messages. The two scenarios of using Simple Storer pattern are illustrated in Fig. 6a and b, respectively. And the structures of Simple Storer pattern are circled with dashed ellipses.

3.1.2. Simple Constructor pattern

Description: A mediator with the capability of simply constructing and sending messages of certain specific type. It should be pointed out that how to construct a message of certain type from a collection of incoming messages is a non-trivial task and some evidences can be used to address the issue [26].

Illustration: The Simple Constructor pattern can be used to resolve mismatches of extra receiving messages and missing sending messages. The two scenarios of using Simple Constructor pattern are illustrated in Fig. 7a and b, respectively.

3.1.3. Splitter pattern

Description: A mediator with the capability of receiving a single message of certain type and splitting it to two or more partial messages. The specific structure of Splitter pattern is variable according to the sequence of partial messages: sequential, parallel or mixed structure. In case of two partial messages, the structure of Splitter pattern can be two types, as shown in Fig. 8a and b, respectively.

Illustration: The Splitter pattern can be used to resolve mismatches of splitting sending messages and merging receiving messages. The two scenarios of using Splitter pattern with sequential structure are illustrated in Fig. 9a and b, respectively.

3.1.4. Merger pattern

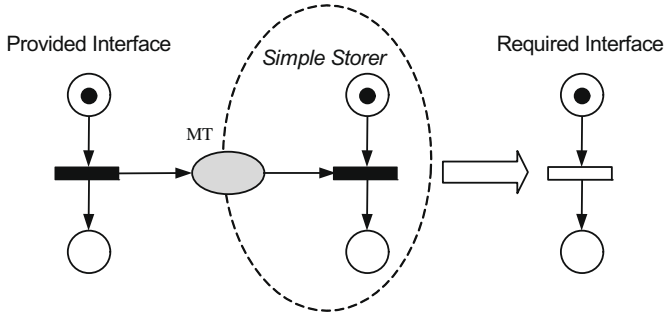
Description: A mediator with the capability of receiving two or more partial messages and merging them to a single one. Similar to Splitter pattern, the specific structure of Merger pattern is variable according to the sequence of merged messages: sequential, parallel or mixed structure. In case of two messages, the structure of Merger pattern can be two types, as shown in Fig. 10a and b, respectively.

Illustration: The Merger pattern can be used to resolve mismatches of splitting receiving messages and merging sending messages. The two scenarios of using Merger pattern with sequential structure are illustrated in Fig. 11a and b, respectively.

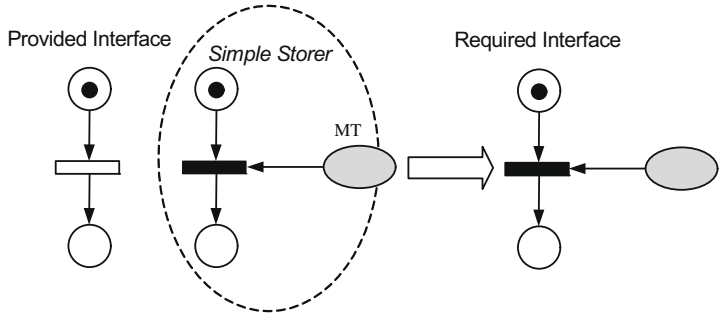
3.1.5. Storing Controller pattern

Description: A mediator with the capability of storing and conditionally sending some messages of certain type in terms of specific logic.

Illustration: The Storing Controller pattern can be used to resolve mismatches of extra condition of receiving messages and missing condition of sending messages. The two scenarios of using Storing Controller pattern are illustrated in Fig. 12a and b, respectively.

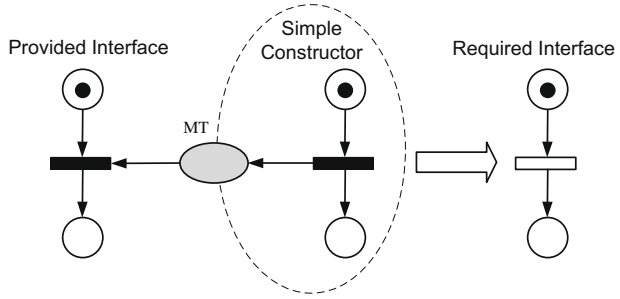


(a) Extra sending message scenario

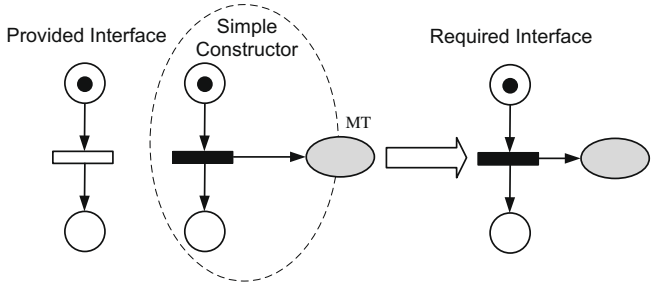


(b) Missing receiving message scenario

Fig. 6. Scenarios of using Simple Storer pattern.



(a) Extra receiving message scenario



(b) Missing sending message scenario

Fig. 7. Scenarios of using Simple Constructor pattern.

3.1.6. Constructing Controller pattern

Description: A mediator with the capability of conditionally constructing and sending some messages of certain type in terms of specific logic.

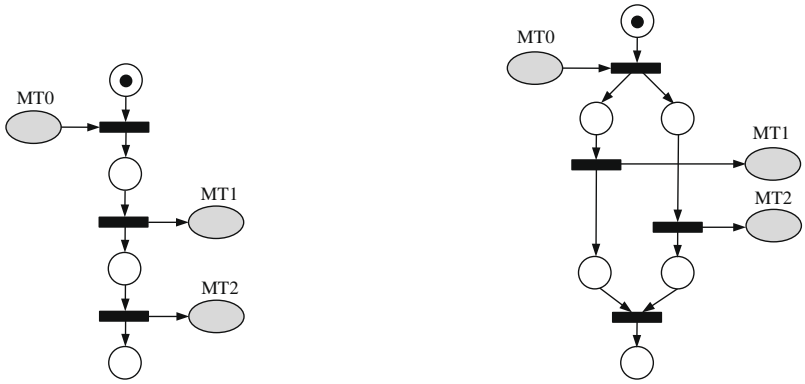
Illustration: The Constructing Controller pattern can be used to resolve mismatches of extra condition of sending messages and missing condition of receiving messages. The two scenarios of using Constructing Controller pattern are illustrated in Fig. 13a and b, respectively.

3.2. Configurability of mediator patterns

As mentioned before, the specific structures of the Splitter and Merger patterns are variable according to the sequences of partial messages. The condition constraints of control logics of the Storing Controller and Constructing Controller patterns are not pre-established. Therefore, we design the mechanism for developers to configure the structures and control logics.

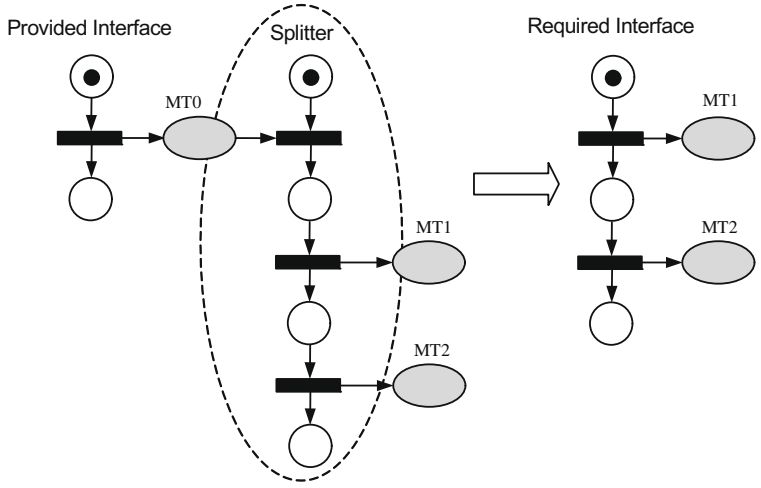
The partial messages of the Splitter/Merger pattern may be sequential, parallel or mixed structure. Before generating pseudo-code of the Splitter/Merger pattern, developers should specify how many partial messages involved and the sequence of these messages. For example, developers can specify a Splitter with three partial messages. After receiving a single message MT 0, the Splitter may send message MT 1 and message MT 2 in parallel. And it may send the third partial message MT 3 after message MT 1 is sent out, as shown in Fig. 14. Once developers configure the sequence of partial messages, the specific structure of the Splitter pattern is identified and automatically concretized.

When reconciling extra or missing condition mismatches, 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

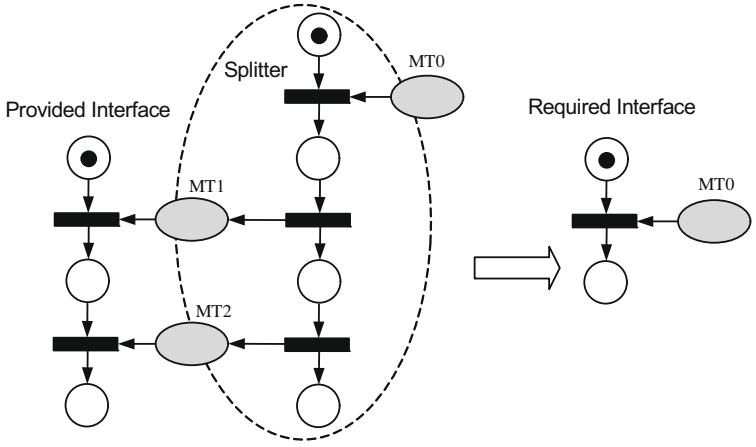


(a) Splitter pattern with sequential structure (b) Splitter pattern with parallel structure

Fig. 8. Two types of Splitter pattern with two partial messages.



(a) Splitting sending message scenario



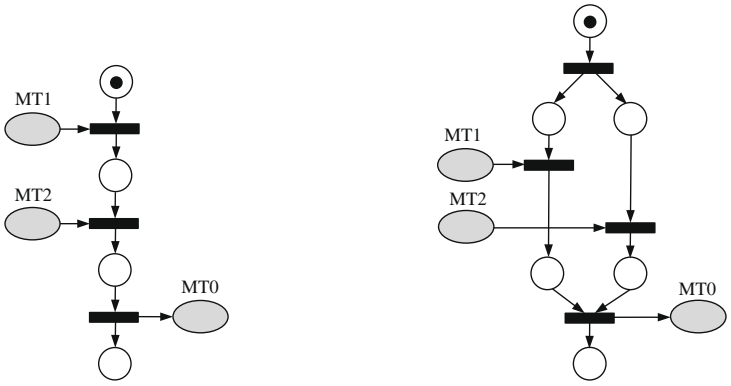
(b) Merging receiving message scenario

Fig. 9. Scenarios of using Splitter pattern.

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. 15. For reconciliation, developers should specify the condition of the Storing Controller pattern as "*Total payment > 1000 USD*".

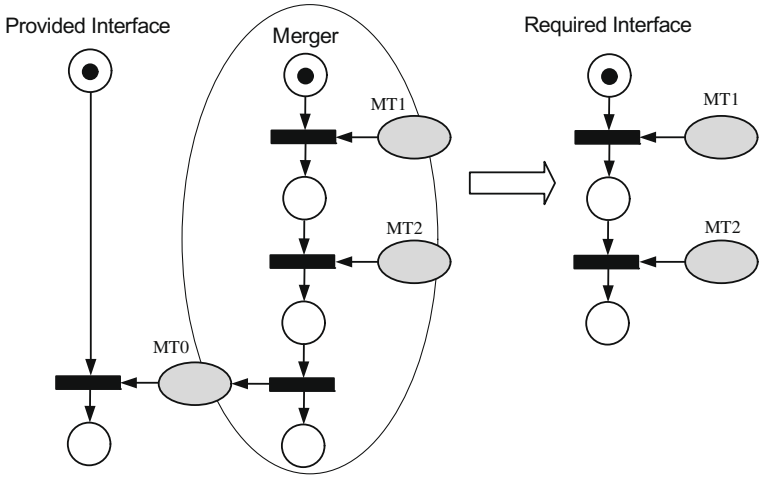
3.3. Composability of mediator patterns

Basic mediator patterns can only reconcile the basic mismatches. More complex protocol mismatches have to be addressed by the composition of basic mediator patterns. Each mediator presented

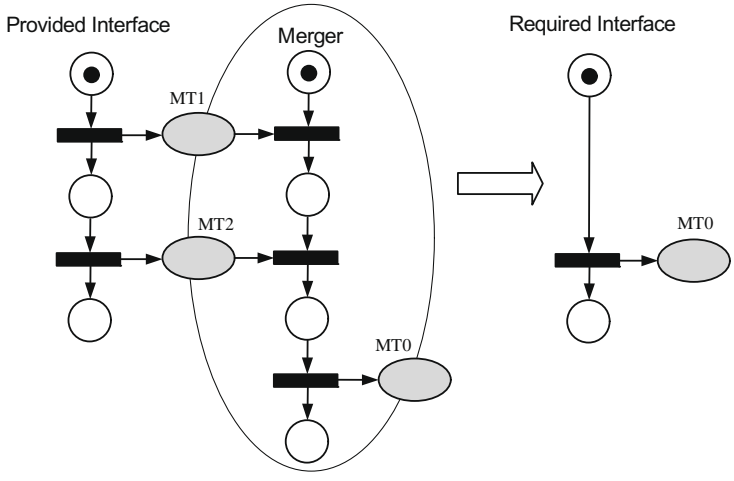


(a) Merger pattern with sequential structure (b) Merger pattern with parallel structure

Fig. 10. Two types of Merger pattern with two merged messages.



(a) Splitting receiving message scenario

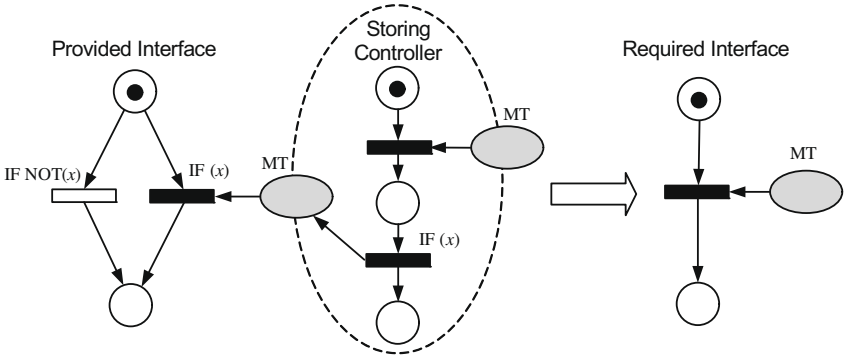


(b) Merging sending message scenario

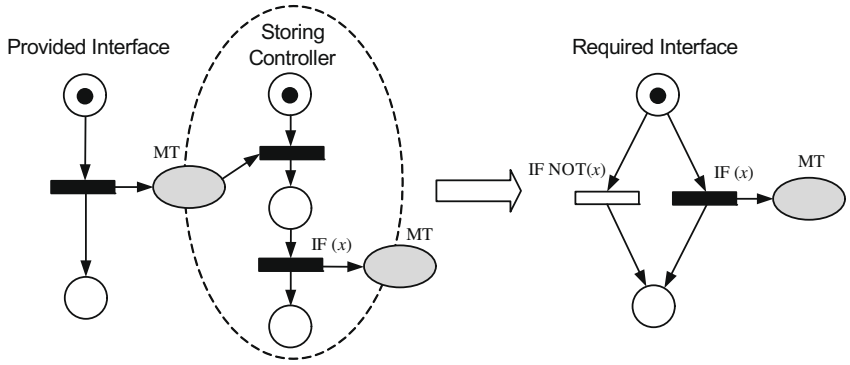
Fig. 11. Scenarios of using Merger pattern.

in this paper has two special places, i.e., the initial place and the end place. 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.

The ordering of messages mismatch, as shown in Fig. 3a, is a complex protocol mismatch that cannot be reconciled by the basic mediators. To resolve it, we can use a composite mediator constructed by a Merger pattern and a Splitter pattern. Fig. 16

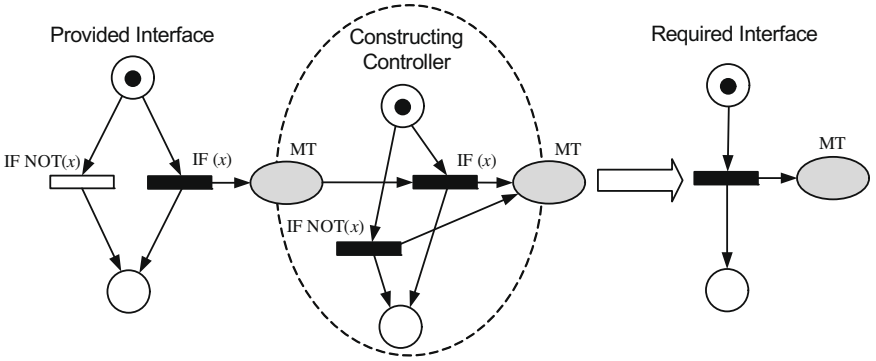


(a) Extra condition of receiving message scenario

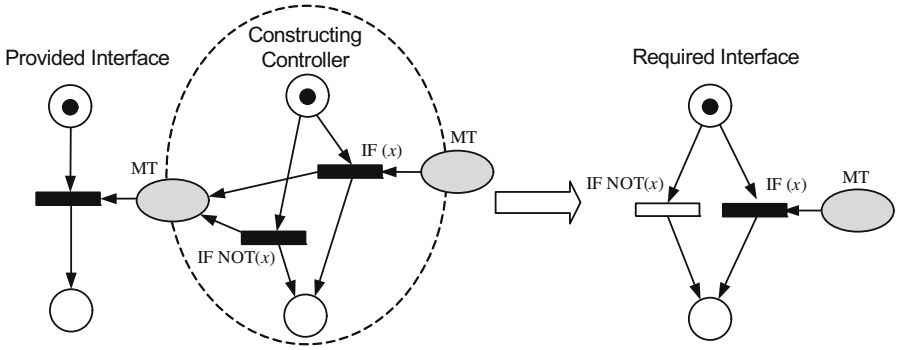


(b) Missing condition of sending message scenario

Fig. 12. Scenarios of using Storing Controller pattern.



(a) Extra condition of sending message scenario



(b) Missing condition of receiving message scenario

Fig. 13. Scenarios of using Constructing Controller pattern.

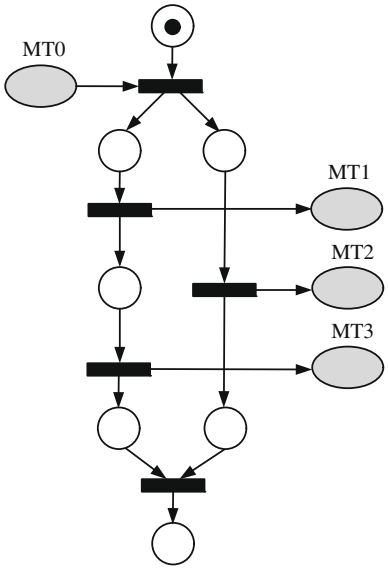


Fig. 14. Splitter pattern with three partial messages.

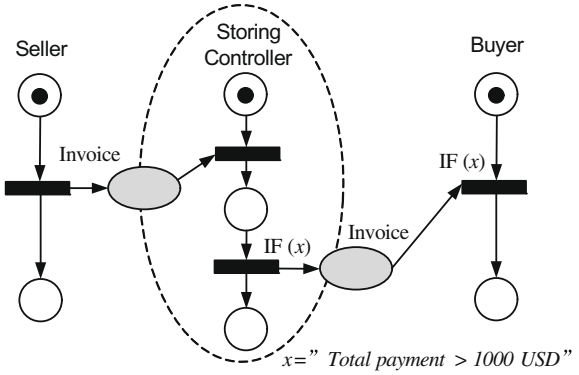


Fig. 15. Storing Controller pattern with specified condition.

illustrates the scenario of using the composite mediator consisting of a Merger and a Splitter to resolve the ordering of messages mismatch.

Fig. 17 shows another composite mediator with iteration structure, namely *Merging Repeater*, which is composed by two Storing Controller patterns. Merging Repeater iteratively receives messages of the type MT 1 until the completing condition *x* occurs. Merging Repeater can be used as a mediator pattern to reconcile protocol mismatches with iteration structure.

Fig. 18 presents such scenario that the provided interface iteratively sends some message of the type MT 1 under certain condition *x* and sends a notification when the condition *x* does not hold. The message type of the notification is also MT 1, but its value is different from that sent message under condition *x*. However, the required interface only expects to send a whole message of the type MT. Note that this scenario depicts the third protocol mismatch in the motivating example (see Section 1.1). As shown in Fig. 18, the mismatch with iteration structure is resolved by Merging Repeater pattern. The condition constraint of Merging Repeater is different from that of the provided interface and it should be specified by developers according to the certain control logics. It is worth noting that Merging Repeater implements the functionality of the *Collapse* operator [9]. Generally, all possible protocol mismatches can be resolved by composite mediators that are composed by the basic mediator patterns.

4. Proposed approach to protocol mediation

4.1. Overview of mediation process

The protocol mediation process works as the following. We take BPEL files of two partially compatible services as the input. Then, we produce executable mediators as the output for reconciling protocol mismatches and gluing the two services together if the correct mediator exists. Fig. 19 shows the mediation approach consisting of five steps.

4.1.1. Service model transformation

As the first step, BPEL-based services are transformed to formal models for the purpose of generating and verifying mediators. The formulism of CPN models cannot 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 [24,25].

4.1.2. 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 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 4.2.

4.1.3. Mediator configuration and composition

The structures and control logics of the mediator patterns need to be configured as parameters by 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 reuse. Both mediator patterns and composite mediators are depicted as underlying CPN models for the following formal verification.

4.1.4. 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. Methods to formal verification of service composition and mediation are presented in our previous work [16,27]. Details of mediation verification are beyond the scope of this paper.

4.1.5. 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

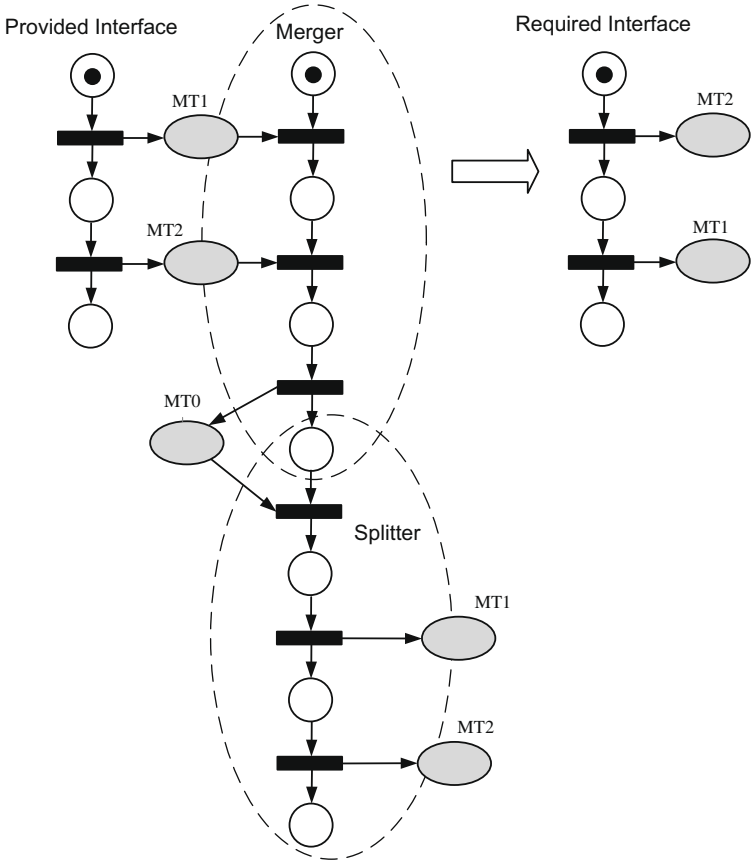


Fig. 16. A composite mediator for reconciling ordering messages mismatch.

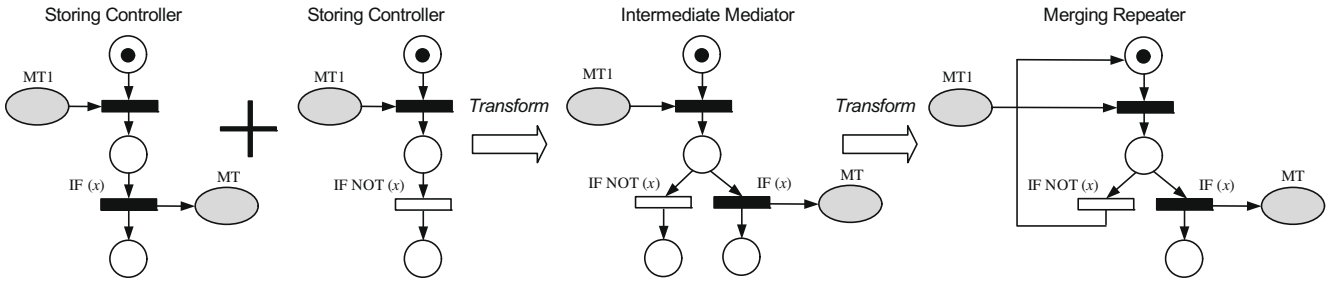


Fig. 17. Merging Repeater pattern composed by two Storing Controller patterns.

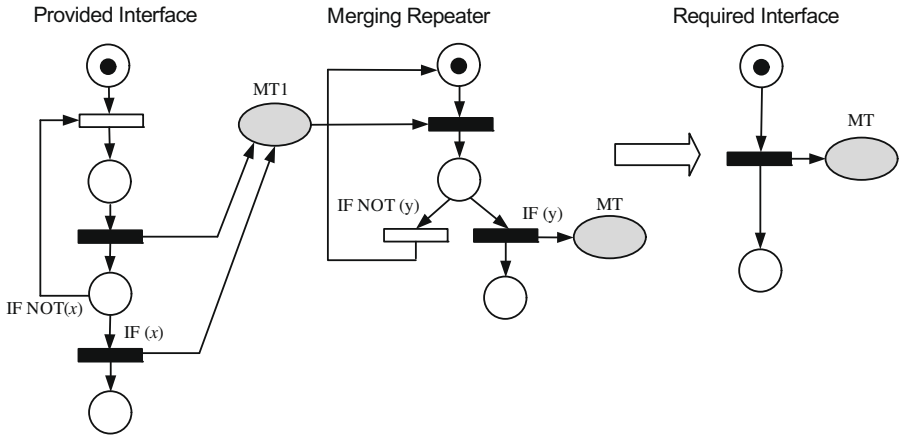


Fig. 18. Scenario of using Merging Repeater pattern.

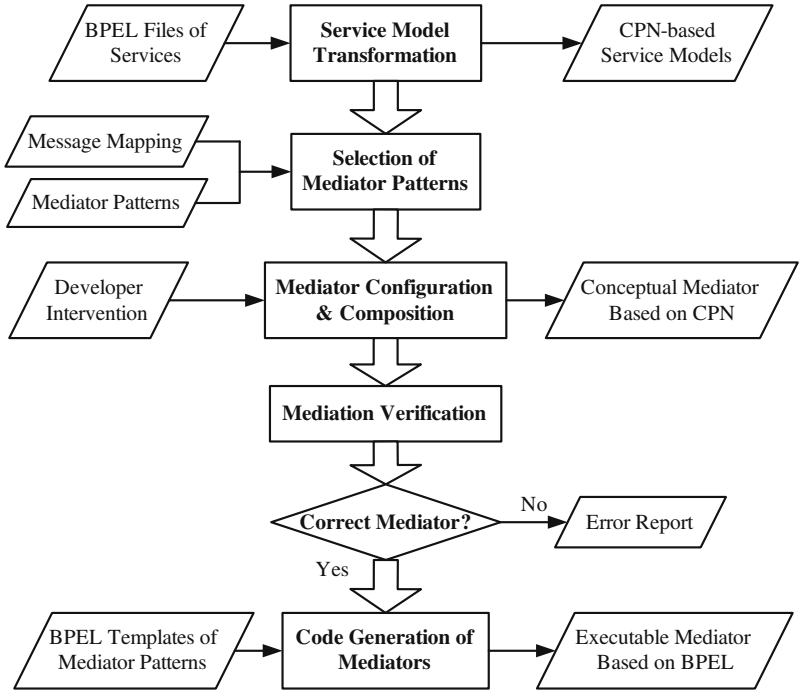


Fig. 19. Overview of the proposed approach.

patterns are developed. With these BPEL templates, the pseudo-code for protocol mediation can be produced automatically.

4.2. 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. To this end, we propose a heuristic technique based on message mapping for semi-automatic selection of mediator patterns. By semi-automation, we mean that 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 [28], 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 does not 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) $source(mr_i) \neq$ NULL, if $target(mr_i)$ = NULL;
(ii) $target(mr_i) \neq$ NULL, if $source(mr_i)$ = 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:

(i) $cnst\_s(mr_i) = cnst\_s(mr_j)$, if $prefix(source(mr_i)) = prefix(source(mr_j))$;
(ii) $cnst\_t(mr_i) = cnst\_t(mr_j)$, 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.

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($\bar{x}$)" indicates that the message "S_E.ntf" is sent when the condition x does not hold. We also denote that $cnst\_s(mr_5)$ = condition($\bar{x}$) 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:

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($\bar{x}$)	NULL	NULL

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 is not 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 is not 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.sreq = S_E.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 3.3).
(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.

As mentioned above, 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. 20, a composite mediator composed by the above three mediator patterns sits between S_E and S_C and reconciles their mismatches. The three mediator patterns are circled with dashed ellipses. Thanks to the CPN formalism, it is easy to verify that S_E and S_C can successfully interact through the composite mediator.

4.3. 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 code of mediators, corresponding BPEL templates are provided for the mediator patterns. With these BPEL templates, the pseudo-code for protocol mediation can be generated automatically. Each mediator pattern has its corresponding BPEL template. In the following, we present the BPEL templates of Simple Constructor pattern, Splitter pattern and Storing Controller pattern. Others can be found in the Appendix A.

4.3.1. 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.

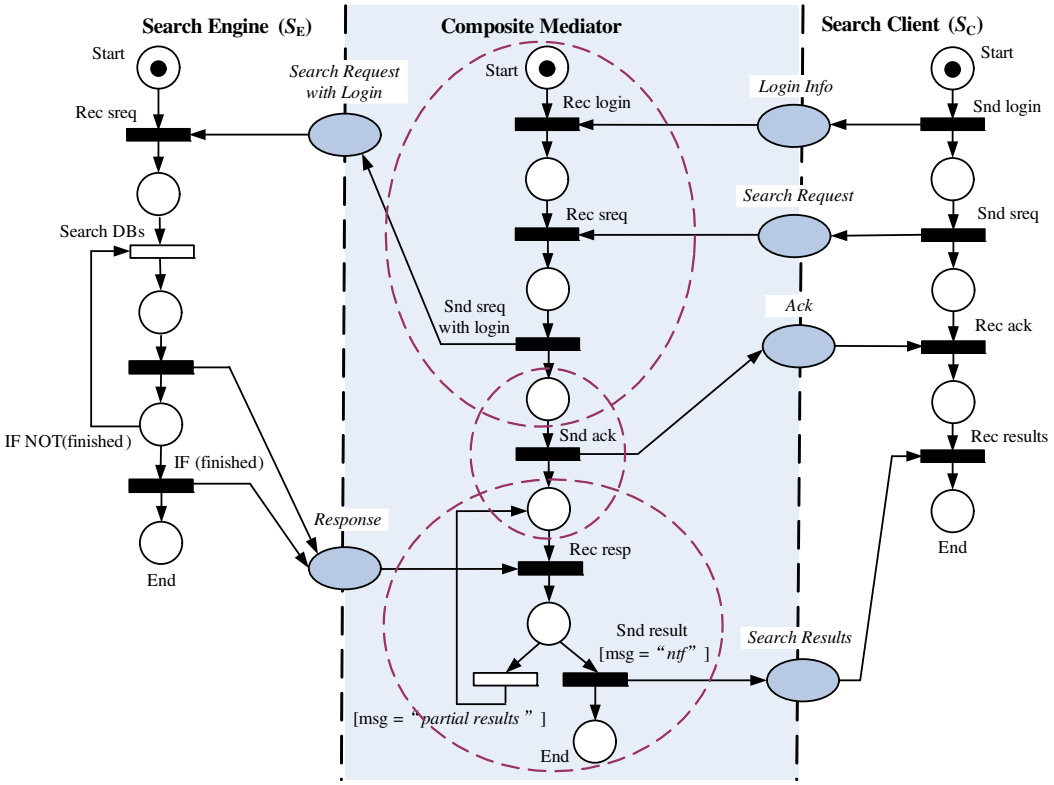


Fig. 20. A composite mediator for the composition of S_E and S_C.

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

4.3.2. 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. 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>
```

4.3.3. 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>
  ...
```

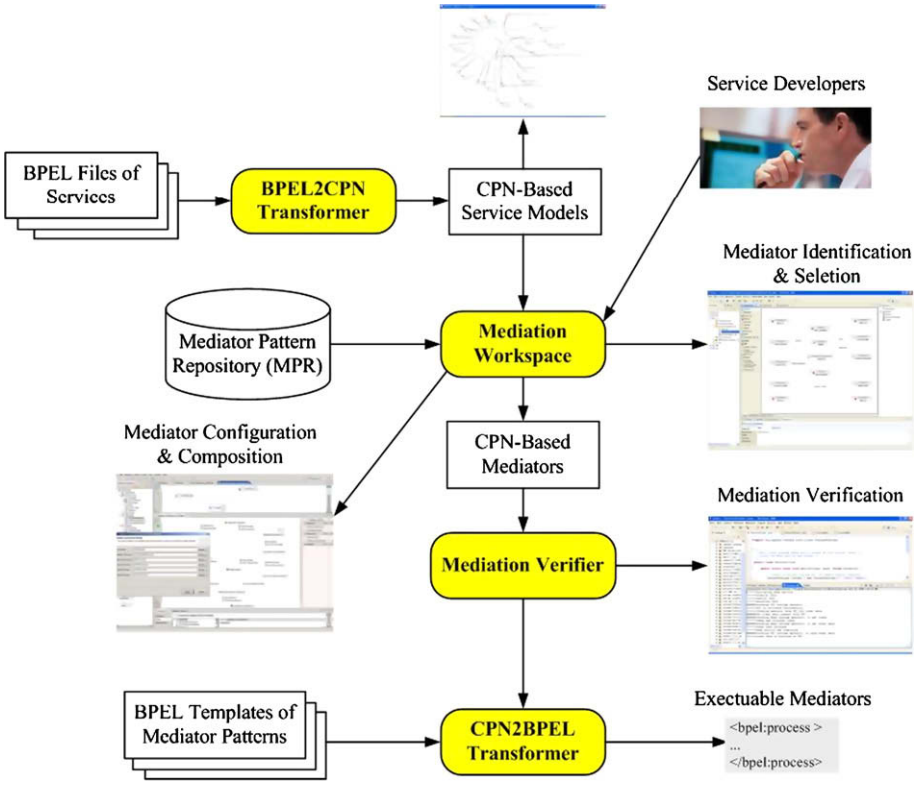


Fig. 21. Architecture of Service Mediation Toolkit (SMT).

```
    </otherwise>
  </sequence>
```

It should be pointed out that the BPEL templates developed above are pseudo-codes for protocol mediation. To get executable BPEL code for deployment, developers need refine the pseudo-codes, e.g., they need specify the definitions of appropriate variables, operations, partnerLinks, portTypes, etc.

5. Prototype implementation and validation

5.1. Service Mediation Toolkit

Mediation-aided service composition is quite complex and needs a comprehensive solution and supporting tools. To validate our proposed solution, we have developed a prototype system, namely *Service Mediation Toolkit* (*SMT*). Fig. 21 shows the architecture of SMT consisting of four main components. In this section, we describe the main components and how SMT is used to facilitate the mediation of BPEL-based services. We focus the validation work on the Mediation Workspace.

5.1.1. BPEL2CPN Transformer

BPEL2CPN Transformer is the first component of SMT and transforms BPEL-based services to CPN models. In SMT, the CPN models are specified using the specification language Petri Net Markup Language (PNML).[4] PNML is an XML-based interchange format for Petri nets, which supports the exchange of all kinds and versions of Petri nets among different PN tools. Currently, existing efforts have provided an open-source Java-based tool BPEL2PNML[5] that can be used to translate process definitions represented in BPEL to Petri nets in PNML. We utilize this tool with minor modification to implement the BPEL2CPN Transformer. For example, BPEL files of S_E and S_C in the motivating example are translated to two PNML files representing their CPN models.

5.1.2. Mediation Workspace

As the core component of SMT, Mediation Workspace is a separate tool and provides a GUI workbench for developers to manipulate services and mediators. Mediator patterns are depicted using CPN models as an underlying formalism, but the protocols of services and mediators are graphically represented by means of an intuitional notation, like Business Process Modeling Notation (BPMN). Developers specify 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 mediator patterns stored in a certain repository Mediator Pattern Repository (MPR). The 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 reuse. By means of the selection rule (see Section 4.2), appropriate mediator patterns are identified and selected from MPR. Developers configure the selected patterns if needed. Then, the selected mediator patterns are composed to produce a composite mediator. The composite mediator, depicted as the intuitional notation with underlying CPN models, is constructed in the Mediation Workspace.

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

[4] http://www2.informatick.hu-berlin.de/top/pnml/about.html/.
[5] http://www.bpm.fit.qut.edu.au/projects/babel/tools/.
[6] http://docs.jboss.org/jbpm/v3/userguide/index.html.

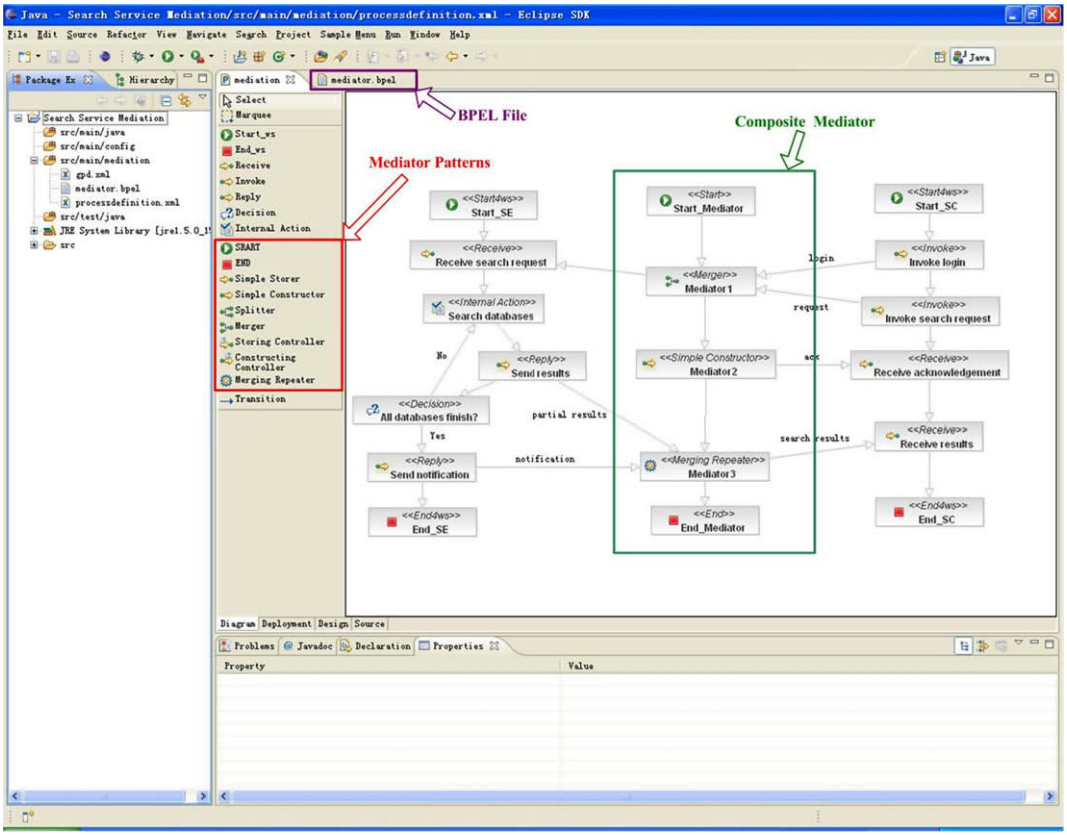


Fig. 22. Screenshot of Mediation Workspace.

applications. Fig. 22 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. The 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 automatically generated in the mediation project.

5.1.3. Mediation Verifier

The composite mediator constructed in the Mediation Workspace may not successfully compensate all possible mismatches and deadlocks may occur. 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 [16,27] are implemented in the Mediation Verifier. The Mediation Verifier is implemented as an analysis module of Platform Independent Petri net Editor (PIPE).[7] PIPE is an open-source Java-based Petri net tool for creating, editing and analyzing Petri nets. Like BPEL2PNML, the Petri nets manipulated by PIPE are also represented in PNML.

5.1.4. CPN2BPEL transformer

Only successful mediator will be performed on the CPN2BPEL. BPEL templates of mediator patterns (see Section 4.3) are used to generate mediation code. Note that the BPEL-based mediator obtained as the output of CPN2BPEL is only pseudo-code. Developers should refine the pseudo-code and generate executable codes.

[7] Platform Independent Petri net Editor 2 - http://pipe2.sourceforge.net/.

5.2. Validation for SWS Challenge scenario

Besides the motivating example, we also used the Semantic Web Service (SWS) Challenge scenario as a further validation to our approach. The SWS Challenge provides a mediation scenario as test bed for applying semantic Web service technology to a set of realistic and working Web services [29]. In the SWS Challenge mediation scenario, the buyer Web service (named Blue) needs to interact with the manufacturer Web service (named Moon) for purchasing goods. As specified using RosettaNet, Blue needs to send Purchase Order (PO) and receives the acknowledgement. Then, Blue waits to receive Purchase Order Confirmation (POC) and sends an acknowledgement. However, in order for Moon, implemented by the legacy system, to be able to process an order, several steps need to be made. Moon requires searching the customer details before an order can be created. Once the customer is identified, Moon waits for the message to create an order. Most differently, Moon requires line items to be submitted individually after the order is created. When receiving the Close Order message, Moon will close the order and send the acknowledgement. The details of the scenario can be found in [29]. With the focus on the protocol mediation, we assumed the exchanged messages are compatible and specified their protocols in BPEL. Their protocols were then translated as CPN models, as shown in the left and right parts of Fig. 23. To mediate protocol mismatches between Blue and Moon, we specified the message relations according to scenario descriptions given in [29]. Using the selection rule, we identified several mediator patterns: six Simple Storer patterns, five Simple Constructor patterns and one Splitting Repeater pattern[8]. Splitting

[8] Similar to Merging Repeater, developer intervention is needed to compose the Splitting Repeater pattern. Once developed, the Splitting Repeater pattern is stored in MPR for further reuse.

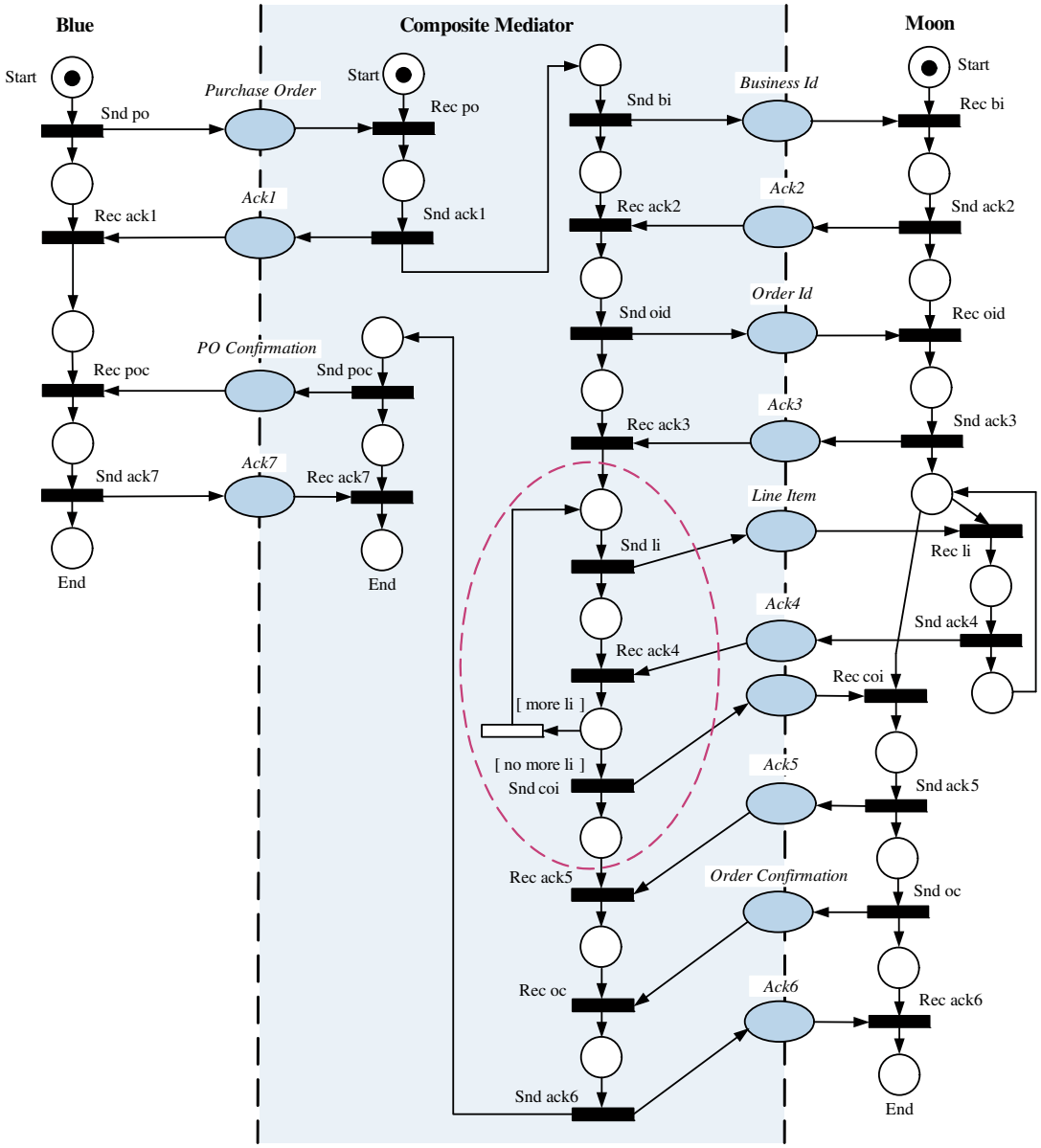


Fig. 23. Mediation for SWS Challenge scenario.

Repeater, circled with a dashed ellipse in Fig. 23, receives a single message and splits it to a number of partial messages to be sent iteratively. The composite mediator were constructed by these mediator patterns and put between Blue and Moon. It is easy to see that the composite mediator reconciles their protocol mismatches. For illustration, Fig. 23 only shows the conceptual CPN models of the services and the composite mediator, rather than the BPEL code.

From the empirical analysis and validation, we have to admit that our solution needs certain manual intervention at the current stage. When dealing with complex protocol mismatches with choice and/or iteration structures, the challenge lies in that the composition of mediator patterns may have multiple variations. In sophisticated cases, developers have to configure and compose mediator patterns for the reconciliation, even though the mediator patterns are identified using mapping relations. Also, some mediator patterns (e.g., Simple Constructing pattern) require evidences to construct the needed messages. For example, the composite mediator in Fig. 23 has to receive Purchase Order (PO) first. Then, it is able to send individual line items iteratively. The observation indicates that mediators cannot be successfully constructed if sufficient evidences are not gathered for constructing messages.

Despite the weakness of possible developer intervention, our solution along with the user-friendly GUI mediation tool can assist developers to alleviate their efforts on several tasks for resolving complex protocol mismatches, such as mismatch/mediator identification, complex mediator composition, and mediation code generation.

6. Related work and discussion

A large number of research works have been developed for service mediation to address various composition mismatches [10]. Signature mediation has received considerable attention [13,14,30] and many commercial tools have been developed. However, protocol mediation is still a challenging issue.

Formal approaches have been developed to conquer this challenge, such as Automata [15], Process Algebra [31] and Petri nets [16]. Yellin et al. [15] present the method to generate adaptors to reconcile the differences between object-oriented components

interfaces. Bracciali et al. [31] use Pi-calculus to specify component interfaces and describes an automatic approach to component adaptation. Considering Automata and Process Algebra have limitations in modeling concurrent behaviors of Web services and their message exchange styles, Tan et al. [16] propose their method to produce mediators for BPEL-based service composition using the CPN formalism. However, the mediators developed by these approaches are conceptual models, rather than executable code. Further, the mediators have no control logics and thus cannot reconcile complex protocol mismatches, such as mismatches with extra condition, missing condition, or iteration structure [17].

It has been recognized that patterns can be used to reconcile composition mismatches and address protocol mediation [9,10,21,32]. Becker et al. [10] present a taxonomy of composition mismatches and a selection of patterns to eliminate these mismatches. The taxonomy, however, does not sufficiently discuss protocol mismatches or mediator patterns. As the first effort in this thread, Benatallah et al. [21] present five mismatch patterns at the protocol level and provide templates of BPEL code for developers to build adapters (i.e., mediators). Based on the work [21], Dumas et al. [9] introduce two more mediator patterns derived from mismatches with iteration structure, namely *Collapse* and *Burst*. The work in [9] presents the mediator patterns as algebra operators and provides visual mapping expressions that allow linking pairs of provided and required interfaces. But there are some mismatches with choice structure that are not discussed in the literature, such as mismatches of extra condition pattern (see Fig. 2e), missing condition pattern (see Fig. 2f), and missing exclusive choice (see Fig. 3b). Both works do not describe how to identify the mediator patterns or how to use the mediator patterns to address complex protocol mismatches. Pokraev et al. [32] presents the initial effort to compose simple mediator patterns in order to resolve complex protocol mismatches, but detailed investigation is not described. The work of this paper is developed towards this direction.

In order to identify possible mismatches and generate adapters, Nezhad et al. [26] develop a significant work by using schema matching techniques. Simple mismatches and appropriate adapters can be automatically identified, but complex protocol mediators (e.g., *Collapse* and *Burst*) are not discussed. It is worth noting that the authors propose semi-automatic methods to analyze certain evidences (including service interfaces, protocols and execution logs) in order to construct required messages and resolve challenging mismatch with deadlocks, such as *ordering of messages mismatch* shown in Fig. 3a. Developer intervention and input are needed for reconciling complex mismatches. Also, the adapters are produced as conceptual models, rather than executable code.

It is difficult to enumerate all possible protocol mismatches with complex control logics or to enumerate sufficient mediators to address those mismatches. Based on our observation and investigation, most of complex protocol mismatches and mediators are subject to the control logics of choice and iteration. Considering the notion of workflow patterns, we realize that complex protocol mediators can be constructed (i.e., composed) by simple ones.

The work of this paper, largely inspired by the above researches [9,21,26,32], complements the state of the art in multiple aspects. (1) Basic mismatch patterns are described and can be used to identify more complex protocol mismatches. Basic mediator patterns presented in this paper resolve the basic mismatches. In particular, several mismatches with choice structure have been first identified and addressed in this paper, such as Extra Condition pattern and Missing Condition pattern. (2) The configurability and composability of basic mediator patterns are investigated. To facilitate mediator composition, the basic mediator patterns are deliberately designed as simple as possible, so that they can be flexibly composed. Complex protocol mediators, e.g., *Merging Repeater* and *Splitting Repeater*, are constructed by composing basic mediator patterns which may need some configuration. It is worth noting that *Merging Repeater* and *Splitting Repeater* developed in this paper can resolve mismatches with iteration structure and implement the functionalities of *Collapse* and *Burst* [9], respectively. (3) The rule-based, heuristic method is developed to assist developers in identifying possible mismatches and selecting appropriate mediators. (4) Both service protocols and mediator patterns are modeled using the CPN formalism. The CPN models can not only depict the internal logics and message exchanging sequences, but also support solid verification of protocol mediation. (5) The pseudo-code of the mediators can be produced using the BPEL templates. (6) The mediation solution and prototype system, *Service Mediation Toolkit* (*SMT*), are developed as the proof-of-concept. SMT provides the user-friendly GUI to assist developers in alleviating the mediation efforts. In order to validate the scalability and applicability, we demonstrate in the paper a few complex protocol mismatch scenarios. These mismatches are resolved by complex mediator patterns that are composed using the basic mediators, such as the mediator pattern for ordering messages mismatch (see Fig. 16) and two mediator patterns for mismatches with iteration structure (see Figs. 18 and 23). It is worth mentioning that mediator patterns, if constructed, can be stored in the Mediator Pattern Repository (MPR) for further reuse.

Table 2
Comparison of related work on protocol mediation.

Mediation approach	FM	CI	AM	MC	CG	DI
Yellin et al. [15]	FSM	N	N	N	N	N
Tan et al. [16]	CPN	N	N	N	N	Y
Benatallah et al. [21]	N/A	N	N	N	Y	N/A
Dumas et al. [9]	UML	Y	Partial	Partial	N	N/A
Pokraev et al. [32]	N/A	N	N	Partial	N	N/A
Nezhad et al. [26]	FSM	N	N	N	N	Y
Our approach	CPN	Y	Y	Y	Y	Y

A comparative summary of previous efforts in this area is presented in Table 2. The columns of the table correspond to the following criteria, where Y means yes and N means No.

- FM indicates the formal model that is used to support mediation verification.
- CI indicates whether complex mismatches are identified, e.g., mismatches with iteration structure.
- AM indicates whether advanced mediators are developed to resolve complex mismatches, e.g., mismatches with iteration structure.
- MC indicates whether complex mediators can be constructed by mediator composition.
- CG indicates whether code generation of mediators is supported.
- DI indicates whether developer intervention is needed to handle complex mismatches.

7. Conclusion

Service mediation is one of the most essential components of Enterprise Service Bus (ESB) and thus becomes a key working area in SOA. In this paper, we have proposed a systematic approach to protocol mediation for BPEL-based services composition. The approach involves several basic mediator patterns and their composition. The pattern-based approach can be used to reconcile all possible protocol mismatches in an engineering way, especially such mismatches with complicated control logics. The method based on message mapping is developed for identifying protocol mismatches and selecting appropriate mediator patterns. We have provided BPEL templates of the mediator patterns for code generation of executable mediators. In addition, the prototype system

SMT has been implemented to validate the feasibility and effectiveness of our approach. The major advantage of the pattern-based approach along with the prototype system SMT lies in the fact that it assists developers in identifying protocol mismatches and alleviates their efforts on the code generation of executable mediator for protocol mediation.

The future work will focus on two aspects. On the one hand, message mapping relations need to be specified by developers in the current stage. In certain complicated situations, developer intervention is needed to select appropriate mediator patterns and compose them. The challenge is that current Web services standards lack of semantic descriptions. 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 develop more advanced mediator patterns and improve our prototype system in handling more complex, real-world cases.

Acknowledgements

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).

Appendix A

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

A.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>
```

A.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. Herein, the BPEL template of the Merger pattern with two sequential partial messages is given.

```
<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>
```

A.3. BPEL template of Constructing Controller pattern

Constructing Controller pattern conditionally constructs and sends a message in terms of logics.

```
<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>
```

A.4. BPEL template of Merging Repeater pattern

Merging Repeater pattern receives messages iteratively and merges the received messages under certain condition. In case the condition does not hold, it sends the whole merged message to its partner.

```
<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>
```

References

[1] M.P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, Service-oriented computing: state of the art and research challenges, IEEE Computer 40 (2007) 38–45.

[2] M.P. Papazoglou, W.J. van den Heuvel, Service oriented architectures: approaches, technologies and research issues, The International Journal on Very Large Data Bases (VLDB Journal) 16 (2007) 389–415.

[3] Z. Maamar, On coordinating personalized composite web services, Information and Software Technology 48 (2006) 540–548.

[4] S.M. Huang, Y.T. Chu, S.H. Li, D.C. Yen, Enhancing conflict detecting mechanism for web services composition: a business process flow model transformation approach, Information and Software Technology 50 (2008) 1069–1087.

[5] Q. Yu, X. Liu, A. Bouguettaya, B. Medjahed, Deploying and managing web services: issues, solutions, and directions, The International Journal on Very Large Data Bases (VLDB Journal) 17 (2008) 537–572.

[6] Y. Shinjo, T. Kubo, C. Pu, Efficient mediators with closures for handling dynamic interfaces in an imperative language, Information and Software Technology 46 (2004) 351–357.

[7] C. Wu, E. Chang, An analysis of web services mediation architecture and pattern in synapse, in: Proceedings of the 21st International Conference on Advanced Information Networking and Applications (AINA 2007), 2007.

[8] M.T. Schmidt, B. Hutchison, P. Lambros, R. Phippen, The enterprise service bus: making service-oriented architecture real, IBM Systems Journal 44 (2005) 781–797.

[9] M. Dumas, M. Spork, K. Wang, Adapt or Perish: algebra and visual notation for service interface adaptation, in Proceedings of the 4th International Conference on Business Process Management (BPM 2006), 2006, pp. 65–88.

[10] S. Becker, A. Brogi, I. Gorton, S. Overhage, A. Romanovsky, M. Tivoli, Towards an engineering approach to component adaptation, in: Architecting Systems with Trustworthy Components, 2006, pp. 193–215.

[11] I.G. Kim, D.H. Bae, J.E. Hong, A component composition model providing dynamic, flexible, and hierarchical composition of components for supporting software evolution, The Journal of Systems & Software 80 (2007) 1797–1816.

[12] S.C. Chou, Dynamic adaptation to object state change in an information flow control model, Information and Software Technology 46 (2004) 729–737.

[13] A.M. Zaremski, J.M. Wing, Signature matching: a tool for using software libraries, ACM Transactions on Software Engineering and Methodology (TOSEM) 4 (1995) 146–170.

[14] M. Szomszor, T.R. Payne, L. Moreau, Automated syntactic mediation for web service integration, in: Proceedings of the International Conference on Web Services (ICWS 2006), 2006.

[15] D.M. Yellin, R.E. Strom, Protocol specifications and component adaptors, ACM Transactions on Programming Languages and Systems (TOPLAS) 19 (1997) 292–333.

[16] W. Tan, Y.S. Fan, 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 6 (2009) 94–106.

[17] X. Li, Y. Fan, F. Jiang, A classification of service composition mismatches to support service mediation, in: Proceedings of the 6th International Conference on Grid and Cooperative Computing (GCC 2007), 2007, pp. 315–321.

[18] X. Li, Y. Fan, J. Wang, L. Wang, F. Jiang, A pattern-based approach to development of service mediators for protocol mediation, in: Proceedings of the 7th Working IEEE/IFIP Conference on Software Architecture (WICSA 2008), 2008, pp. 137–146.

[19] F. Jiang, Y. Fan, X. Zhang, Rule-based automatic generation of mediator patterns for service composition mismatches, in: Proceedings of the 3rd International Conference on Grid and Pervasive Computing Symposia/Workshops (GPC2008), Kunming, China, 2008, pp. 3–8.

[20] X. Li, Y. Fan, S. Madnick, Q.Z. Sheng, Reconciling Protocol Mismatches of Web Services by Using Mediators, in: Proceedings of the 18th Workshop on Information Technologies and Systems (WITS 2008), Paris, France, 2008, pp. 3–8.

[21] B. Benatallah, F. Casati, D. Grigori, H.R.M. Nezhad, F. Toumani, Developing adapters for web services integration, in: Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE 2005), 2005.

[22] W.M.P. Van Der Aalst, A.H.M. Ter Hofstede, B. Kiepuszewski, A.P. Barros, Workflow patterns, Distributed and Parallel Databases 14 (2003) 5–51.

[23] K. Jensen, Coloured petri nets: basic concepts, analysis methods and practical use, Basic Concepts, Monographs in Theoretical Computer Science, vol. 1, Springer-Verlag, 1997.

[24] C. Ouyang, E. Verbeek, W.M.P. van der Aalst, S. Breutel, M. Dumas, A.H.M. ter Hofstede, Formal semantics and analysis of control flow in WS-BPEL, Science of Computer Programming 67 (2007) 162–198.

[25] W.M.P. van der Aalst, K. Bisgaard Lassen, Translating unstructured workflow processes to readable BPEL: theory and implementation, Information and Software Technology 50 (2008) 131–159.

[26] H.R.M. Nezhad, B. Benatallah, A. Martens, F. Curbera, F. Casati, Semi-automated adaptation of service interactions, in: Proceedings of the 16th International Conference on World Wide Web (WWW 2007), 2007, pp. 993–1002.

[27] X. Li, Y. Fan, Modeling and logical correctness verification of web service processes, Computer Integrated Manufacturing Systems 14 (2008) 675–682 (in Chinese).

[28] E. Rahm, P.A. Bernstein, A survey of approaches to automatic schema matching, The International Journal on Very Large Data Bases 10 (2001) 334–350.

[29] Semantic Web Services Challenge. <http://sws-challenge.org/>.

[30] M. Younas, K.M. Chao, C. Laing, Composition of mismatched web services in distributed service oriented design activities, Advanced Engineering Informatics 19 (2005) 143–153.

[31] A. Bracciali, A. Brogi, C. Canal, A formal approach to component adaptation, The Journal of Systems & Software 74 (2005) 45–54.

[32] S. Pokraev, M. Reichert, Mediation patterns for message exchange protocols, in: Open INTEROP-Workshop on Enterprise Modeling and Ontologies for Interoperability (EMOI 2006), 2006, pp. 659–663.