A Pattern-Based Approach to Development of Service Mediators for Protocol Mediation

Xitong Li[1], Yushun Fan[1], Jian Wang[2], Li Wang[2], Feng Jiang[1]
[1]*Department of Automation*
Tsinghua University
Beijing, 100084, P.R. China
{lxt04, jiangf00}@mails.tsinghua.edu.cn
fanyus@tsinghua.edu.cn
[2]*IBM China Research Lab*
Beijing, 100094, P.R. China
{wangwj,wanglcrl}@cn.ibm.com

Abstract

Service composition is one of the key objectives for adopting Service Oriented Architecture. Today, web services, however, are not always perfectly compatible and composition mismatches are common problems. Service mediation, generally classified into signature and protocol ones, thus becomes one key working area in SOA. While the former has received considerable attention, protocol mediation is still open and current approaches provide only partial solutions. In this paper, a pattern-based approach is proposed for developers to semi-automatically generate mediators and glue partially compatible services together. Based on the investigation on workflow patterns and message exchanging sequences in service interactions, several basic mediator patterns are developed and can be used to modularly construct advanced mediators that can resolve all possible protocol mismatches, especially such mismatches about complicated control logics. Moreover, the architecture for the service mediation system is designed and implemented to prove the feasibility of our approach.

1. Introduction

Service composition is one of the key objectives of Service Oriented Architecture (SOA) which provides a loosely-coupled environment and enables flexible assembly of pre-produced services. Today, web services, however, are not always perfectly compatible and can not be straightly composed together. Thus composition mismatches are common problems.

An effective solution to this challenge is service mediation which is recognized as the act of retrofitting existing services by intercepting, storing, transforming, and (re-) routing messages going into and out of these services [1]. Recently, service mediation has become the status of a definite working area in the field of SOA and attracted much attention [2]. Generally, service mediation can be classified into signature and protocol ones. Signature mediation, where the focus is on message types, has received considerable attention [3]. In comparison, protocol mediation, where the focus is on resolving mismatches occurring at the message exchanging sequences, is still open. Current approaches provide only partial solutions and need further research, since mediators developed by these approaches have no control logics and can not compensate complicated protocol mismatches.

In our previous work, we have presented a comprehensive identification of all possible composition mismatches. Particularly, we have proposed six basic mismatch patterns and pointed out that all possible protocol mismatches can be composed by these basic patterns [4]. In this paper, we develop several basic mediator patterns to address these basic mismatches and present a mechanism about the composition of the mediator patterns. Moreover, the structures and control logics of the mediator patterns can be configured as parameters when developers use them to construct advanced mediators. By using these basic mediators as patterns, service mediators can be modularly constructed and contain control logics. The most advantage of our approach lies in the capability of resolving all possible protocol mismatches, especially mismatches of complicated control logics.

1.1. Motivating Example

A motivating example comes from a composition scenario of a search client (*SC*) and a search engine (*SE*). *SC* sends its login information followed by the search request. After that, *SC* waits for receiving acknowledgement as well as the information of searched items from *SE*. On the other hand, after receiving search request, *SE* starts to search in several distributed databases one by one. Once *SE* finishes a partial database and obtains some searched items, it sends information of these items immediately. If all databases have been searched, *SE* sends a completing notification to its client and the search work is finished. For the sake of clear representation, we abstract the specific definitions of the protocols of the two services, such as BPEL code. And their message exchanging sequences are illustrated in Figure 1.

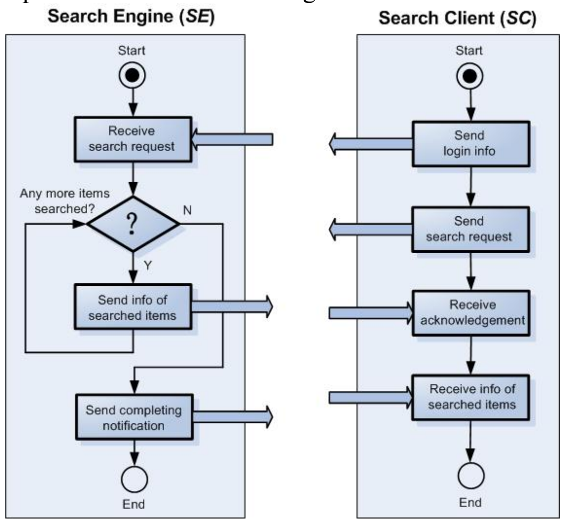


Figure 1. A motivating example of service composition with protocol mismatches

It is easy to see that the two services, *SE* and *SC*, provide complementary functionality. However, they do not fit each other exactly, due to protocol mismatches identified as follows:

i) *SE* expects a whole message of search request containing login information, while *SC* sends login information and search request separately.

ii) *SE* does not send any acknowledgement after receiving search request, but *SC* waits for it.

iii) *SE* sends searched items one by one according to the results of partial databases, while *SC* expects to receive a whole message of all searched items.

The first and second mismatches in the motivating example have been identified in the existing work [5]. As far as we've known, however, no approach addresses the third mismatch which is considered as iteration mismatch in this paper.

1.2. Protocol Mismatch Patterns

Protocol mismatches refer to mismatches occurring at the message exchanging sequences of the services to be composed together. The existing work has identified this kind of mismatches [5]. However, few paper claims its identification is complete in any sense. To achieve a complete identification, we have proposed six basic mismatch patterns, which are derived from basic workflow patterns and message exchanging sequences. We have illustrated that the six basic mismatch patterns can be viewed as basic constructs of all possible protocol mismatches in our previous work [4]. Particularly, protocol mismatches in the motivating example can be composed by the basic mismatch patterns, which is not illustrated due to the space limitation. In this paper, we adopt the irreplaceability perspective to represent protocol mismatches, which is based on the scenario that the required interface can not be exactly replaced by the provided interface. Basic mismatch patterns are presented as follows:

(1) Mismatches of extra messages: the provided interface has some extra messages that the required interface does not expect to send/receive.

(2) Mismatches of missing messages: the provided interface does not have some messages that the required interface expects to send/receive.

(3) Mismatches of splitting messages: the provided interface has some messages that the required interface expects to split to send/receive.

(4) Mismatches of merging messages: the provided interface has some messages that the required interface expects to merge to send/receive.

(5) Mismatches of extra conditions: 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.

(6) Mismatches of missing conditions: 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.

1.3. Contributions and Structures

The main contributions that we have achieved in the paper are:

1) We have proposed a pattern-based solution approach for developers to resolve all possible protocol mismatches and glue partially compatible services together. The approach can semi-automatically produce pseudo-codes for developers to generate executable codes, like BPEL code. Since we abstract the specific definitions of the protocols, the approach is not limited to BPEL-based services and can be used with other definition languages.

2) We have presented several basic mediator patterns which are derived from the protocol mismatch

patterns identified in our previous work. The well-defined basic mediator patterns can be configured and composed by developers, according to the specific protocol mismatches. Based on a deep investigation on typical workflow patterns and message exchanging sequences in service interactions, the basic mediator patterns presented in the paper are considered to be sufficient to construct mediators for resolving all possible protocol mismatches.

3) We have designed the architecture to implement the service mediation system (SMS) which is an ongoing development and integrated with IBM WID (WebSphere Integration Developer). The development of SMS can be used to prove the feasibility and effectiveness of our solution approach.

The rest of the paper is structured as follows. The solution approach is presented in Section 2. Next, basic mediator patterns and their configurability and compositionability are proposed in Section 3. In Section 4, a service mediator to resolve mismatches in the motivating example is presented and the architecture of prototype system for service mediation is developed. Related work and comparisons with ours are presented in Section 5. Finally, conclusions and the future work are drawn up in Section 6.

2. Solution Approach

In this section, we present a solution approach to address protocol mediation, as shown in Figure 2. Since protocols of services are usually specified in WS-BPEL which has been approved as an OASIS Standard for web services [6], we focus on addressing the mediation of BPEL-based services. But the approach can be easily applied to other definition languages. Given two interacting services, several procedures should be performed to produce deployable service mediators for resolving the protocol mismatches and compatibly gluing the two services together, if the correct mediator exists.

(1) Service model transformation

For the purpose of mismatch identification, BPEL-based services are transformed to formal models. In the paper, we adopt Colored Petri Nets (CPNs) as an underlying formulism to represent the protocols of services and mediators. The formulism of CPNs can not only depict the internal logics and message exchanging sequences, but also provides rich analysis capability to support solid verification of correctness for protocol mediation. Existing papers have presented approaches to the transformation from BPEL-based service models to CPN models [7].

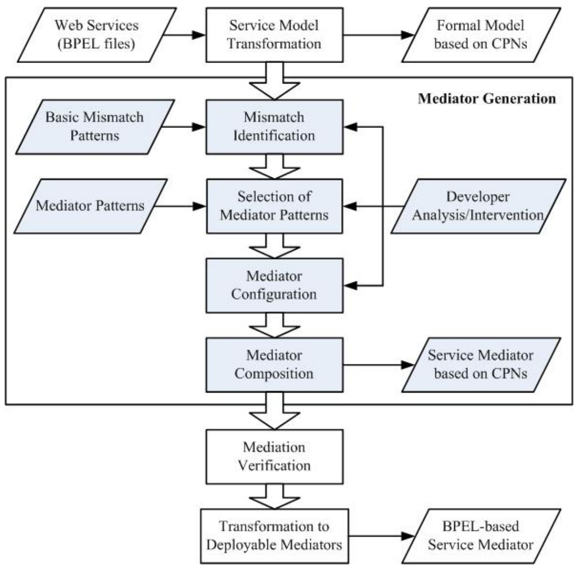

Web Services (BPEL files)
Service Model Transformation
Formal Model based on CPNs
Mediator Generation
Basic Mismatch Patterns
Mismatch Identification
Mediator Patterns
Selection of Mediator Patterns
Developer Analysis/Intervention
Mediator Configuration
Mediator Composition
Service Mediator based on CPNs
Mediation Verification
Transformation to Deployable Mediators
BPEL-based Service Mediator

Figure 2. Solution approach to protocol mediation

(2) Mediator generation

There are four sub-steps to achieve mediator generation. Firstly, in terms of basic mismatch patterns, developers analyze the models of two interacting services and identify all possible protocol mismatches. Secondly, with basic protocol mismatches, developers select corresponding mediator patterns which are proposed in the next section. Thirdly, the structures and control logics of the mediator patterns need to be configured as parameters by developers, according to the identified mismatches. Finally, the configured mediator patterns are composed to construct a composite mediator that can resolve all identified protocol mismatches. It should be pointed out that a composite mediator can be treated as an advanced pattern for further use. Both mediator patterns and composite mediators are represented as CPN models. Since the former three sub-steps need developers' intervention, the procedure is considered as semi-automatic mediator generation, which can be used to resolve all possible protocol mismatches and is one of the main contributions of this paper.

(3) Mediation verification

The mediator generated in the above procedure is only conceptual and should be placed between the two interacting services. The composition model of the two services and the mediator need to be formally verified. If any deadlock exists, we consider that the mediation has failed. Otherwise, the mediation is successful. In the existing work [8] [9], some verifying approaches to service composition and compatibility have been proposed based on the Petri net formulism.

(4) Transformation to deployable mediators

Only successful mediation will be performed in this procedure. The conceptual mediator is transformed to deployable/executable service mediators, like BPEL-based mediators, which are pattern-specific codes and need developers' refinement.

Note that the focus of this paper is the development of basic mediator patterns and how to use these patterns to construct service mediators for protocol mediation. A deeper research of the third and fourth procedures of the solution approach is beyond the scope of this paper and subject of future work.

3. Mediator Patterns for Protocol Mediation

To a certain extent, the dependencies of message exchanging sequences between two interacting services are similar to control flows of their internal logics. Thus, we can 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 [10] [11], namely sequence, parallel, exclusive choice and iteration. And advanced workflow constructs are supposed to be composed by using basic workflow patterns. Derived from basic workflow patterns and message exchanging sequences, six basic mediator patterns are proposed in this section. It should be pointed out that the six basic mediators can be treated as basic patterns to modularly construct service mediators which can be used to resolve all possible protocol mismatches. Therefore, the basic patterns identified herein are considered to be sufficient. And a composite mediator can be stored as a mediator pattern for further use. Moreover, the configurability and compositionability of the mediator patterns are presented to flexibly construct advanced mediators.

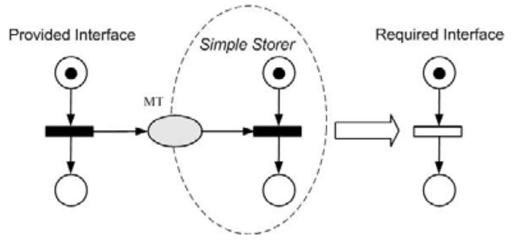
Both basic and composite mediators presented in this paper are conceptual patterns which can provide pseudo-code to develop executable codes for mediation, like BPEL code. The intended benefit of this work is to help developers produce service mediators through an engineering methodology and semi-automatically generate mediation codes by using these patterns. Although mediator patterns depicted in this section are based on CPN models, the protocols of services and mediators are represented for developers by using an intuitional and graphics-based notation, like Business Process Modeling Notation (BPMN) [12].

3.1. Basic Mediator Patterns

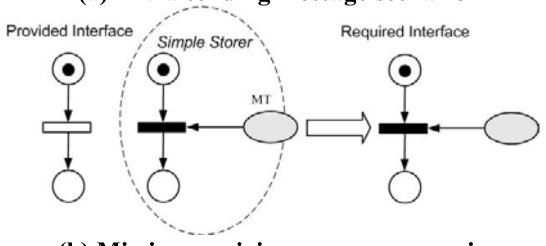
(1) Simple Storer pattern

Description: A service 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 respectively illustrated in Figure 3(a) and Figure 3(b). And the structures of Simple Storer pattern are circled with dashed ellipses. In the figures of this paper, the white transitions depict those actions without sending/receiving any message and the symbol "MT" stands for a certain message type for short.



(a) Extra sending message scenario



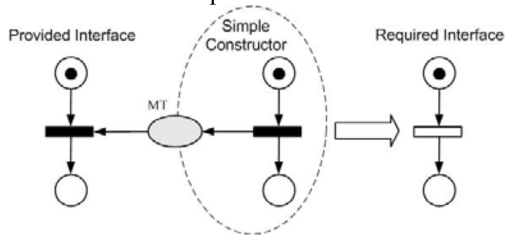
(b) Missing receiving message scenario

Figure 3. Scenarios of using Simple Storer pattern

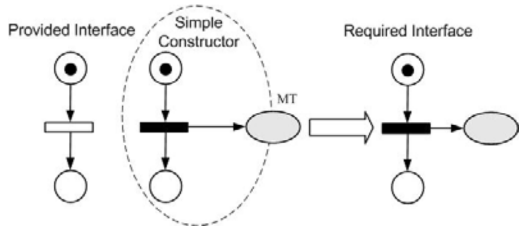
(2) Simple Constructor pattern

Description: A service 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 [13].

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 respectively illustrated in Figure 4(a) and Figure 4(b). And the structures of Simple Constructor pattern are circled with dashed ellipses.



(a) Extra receiving message scenario

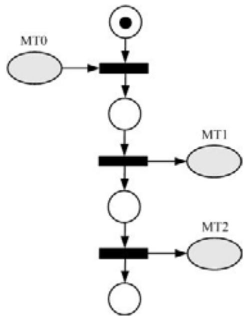


(b) Missing sending message scenario
Figure 4. Scenarios of using Simple Constructor pattern

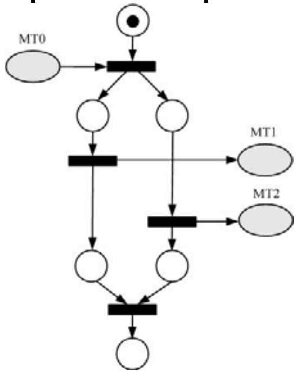
(3) Splitter pattern

Description: A service 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 which may be sequential, parallel or mixed structure. If splitting to two partial messages, the structure of Splitter pattern can be two types, as shown in Figure 5(a) and Figure 5(b).

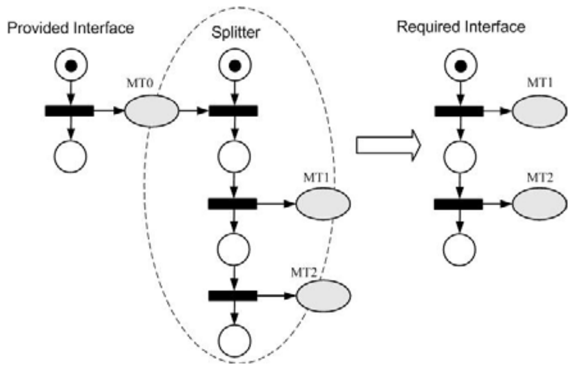
Illustration: The Splitter pattern can be used to resolve mismatches of splitting sending messages and merging receiving messages. The two scenarios of using Splitter pattern with sequential structure are respectively illustrated in Figure 6(a) and Figure 6(b). And the structures of Splitter pattern are circled with dashed ellipses.



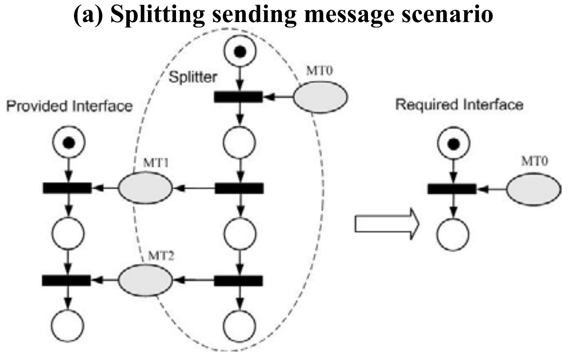
(a) Splitter pattern with sequential structure



(b) Splitter pattern with parallel structure
Figure 5. Two types of structures of Splitter pattern with two partial messages



(a) Splitting sending message scenario

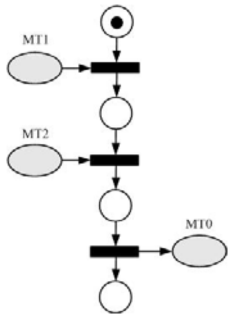


(b) Merging receiving message scenario
Figure 6. Scenarios of using Splitter pattern

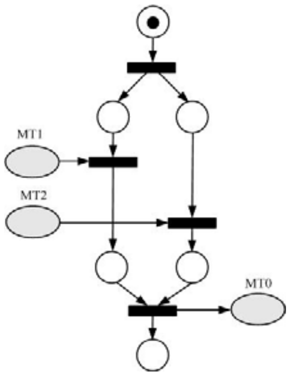
(4) Merger pattern

Description: A service 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 which may be sequential, parallel or mixed structure. If merging two messages, the structure of Merger pattern can be two types, as shown in Figure 7(a) and Figure 7(b).

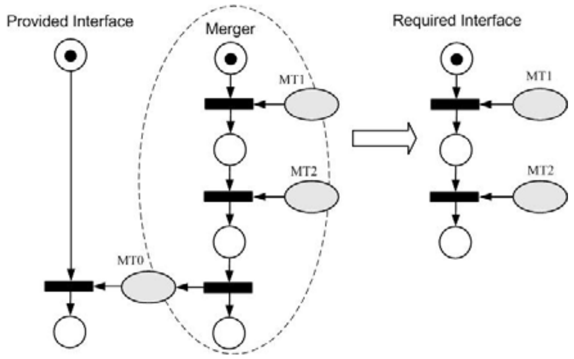
Illustration: The Merger pattern can be used to resolve mismatches of splitting receiving messages and merging sending messages. The two scenarios of using Merger pattern with sequential structure are respectively illustrated in Figure 8(a) and Figure 8(b). And the structures of Merger pattern are circled with dashed ellipses.



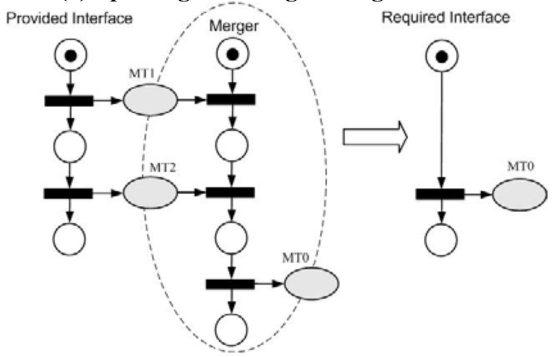
(a) Merger pattern with sequential structure



(b) Merger pattern with parallel structure
Figure 7. Two types of structures of Merger pattern with two merged messages



(a) Splitting receiving message scenario

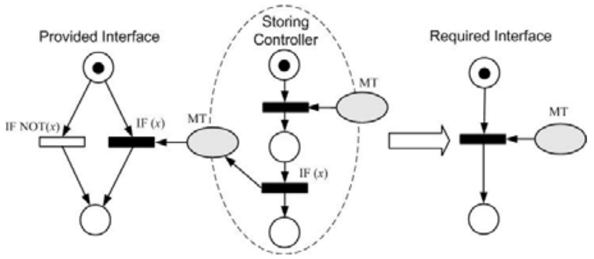


(b) Merging sending message scenario
Figure 8. Scenarios of using Merger pattern

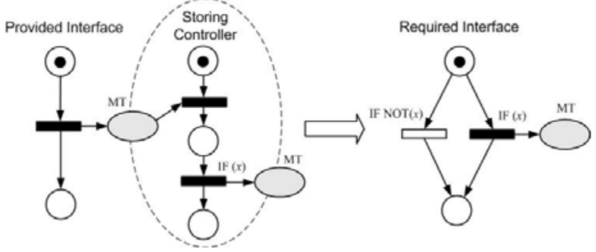
(5) Storing Controller pattern

Description: A service 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 respectively illustrated in Figure 9(a) and Figure 9(b). And the structures of Storing Controller pattern are circled with dashed ellipses.



(a) Extra condition of receiving message scenario

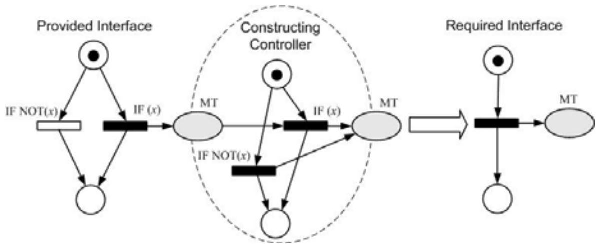


(b) Missing condition of sending message scenario
Figure 9. Scenarios of using Storing Controller pattern

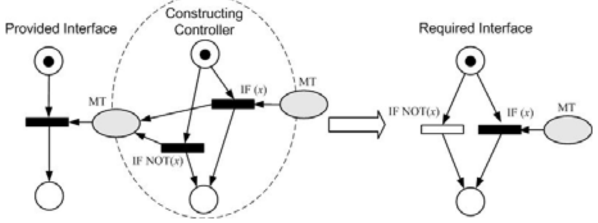
(6) Constructing Controller pattern

Description: A service 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 respectively illustrated in Figure 10(a) and Figure 10(b). And the structures of Constructing Controller pattern are circled with dashed ellipses.



(a) Extra condition of sending message scenario



(b) Missing condition of receiving message scenario
Figure 10. Scenarios of using Constructing Controller pattern

3.2. Configurability of Mediator Patterns

As mentioned above, the specific structures of the Splitter and Merger patterns may be variable according to the sequences of partial messages. Also, the condition constraints of control logics of the Storing Controller and Constructing Controller patterns are not pre-established. Thus, we device specific interfaces for the basic mediator patterns to configure their structures and control logics.

Before using the Splitter and Merger patterns, developers should specify how many partial messages involved as well as the sequence of these messages, that is, sequential, parallel or mixed structure. After configuration, the specific structures of the Splitter and Merger patterns can be identified and concretized.

When resolving 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 such BPEL elements as <switch>, <pick>, <while> or <repeatUntil>.

3.3. Compositionability of Mediator Patterns

Basic protocol mismatches can be resolved by the abovementioned basic mediator patterns. In practical environments, however, protocol mismatches are more complicated and should be addressed by advanced mediators with control logics that are composed by these basic mediators. Then a composite mediator can be considered as a new pattern and be used in the future. Each mediator presented in this paper has two special places that are an initial place and an end place. Informally, the composition of two mediators is performed by merging the end place of one mediator with the initial place of the other as well as the common parts of the two mediators. To illustrate the composition of mediators, herein take a mediator with iteration structure, namely Merging Repeater, for example, as shown in Figure 11. It's easy to see that Merging Repeater can iteratively receive messages of the type MT1 until the completing condition *x* occurs.

Merging Repeater pattern can be used as a mediator to resolve protocol mismatches with iteration structure. Figure 12 presents such scenario that the provided interface iteratively sends some message of the type MT1 under certain condition *x* and sends a notification when that condition *x* doesn't hold. The message type of the notification is also MT1, but its specific value is different with that sent 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). Figure 12 shows that the mismatch with iteration structure can be compensated by using Merging Repeater pattern. Moreover, we believe all possible protocol mismatches can be resolved by the advanced mediators that are composed by mediator patterns presented in this paper.

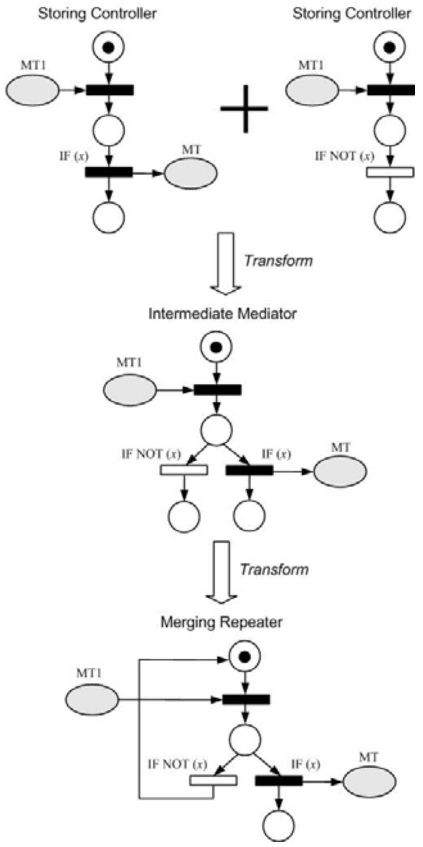


Figure 11. Merging Repeater pattern composed by two Storing Controller patterns

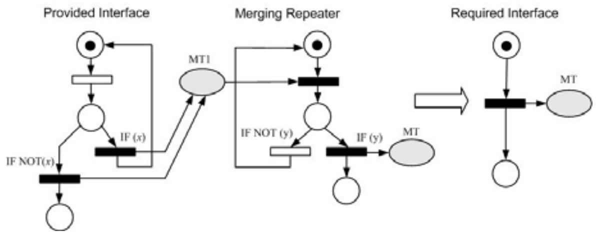


Figure 12. Scenario of using Merging Repeater pattern

4. Prototype System

To prove the feasibility of our approach, the protocol mismatches in the motivating example are resolved by using a composite mediator that is constructed by the mediator patterns. And also, a prototype system has been designed and being developed, which is known as Service Mediation System (SMS).

4.1. A Service Mediator to Resolve Mismatches in the Motivating Example

There are three protocol mismatches in the motivating example (see Section 1.1) and three mediator patterns can be respectively used to address these mismatches as follows:

i) A Merger can be used to receive the login information and search request from *SC*, and then it sends *SE* a whole message of search request with login information.

ii) A Simple Constructor can be used to construct the acknowledgement of specific type and send *SC* the acknowledgement that *SC* expects to receive.

iii) A Merging Repeater can be used to iteratively receive the search results from *SE* until the notification of no more items arrives. And then, the Merger Repeater merges all the search results together and sends *SC* a whole message of searched results.

As shown in Figure 13, a composite mediator composed by the above three mediator patterns sits between the two interacting services, *SE* and *SC*, and compensates their protocol mismatches. The three mediator patterns are circled with dashed ellipses. Since the protocols of *SE*, *SC* and mediators are modeled by CPNs formalism, it is easy to verify that *SE* and *SC* can successfully interact through the composite mediator and no deadlock exists.

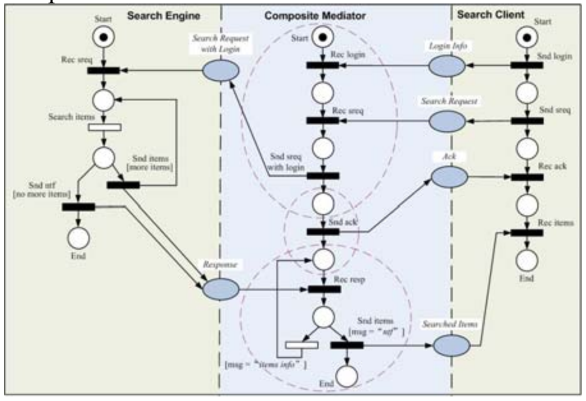


Figure 13. A composite mediator for protocol mediation of *SE* and *SC*

4.2. Architecture of Service Mediation System

The solution approach presented in this paper is developed inside IBM WID (Websphere Integration Developer) which is an Eclipse-based IDE for development of composite applications based on Service Component Architecture (SCA) [14]. The architecture for the Service Mediation System (SMS), as shown in Figure 14, currently supports mediation of BPEL-based services. There are four main components in SMS which are introduced as follows.

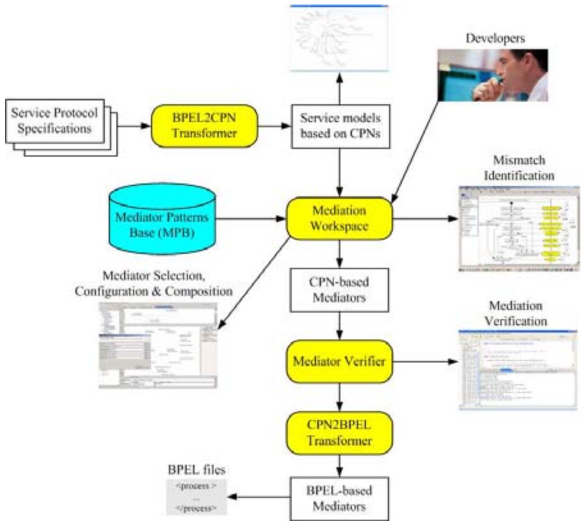


Figure 14. Architecture of SMS

(1) BPEL2CPN Transformer

Web services to be composed together are implemented with BPEL and wrapped as SCA components. The BPEL2CPN Transformer is responsible to transform BPEL-based service models to CPN models. Recently, current tools have provided similar functionalities, such as BPEL2PNML [15].

(2) Mediation Workspace

The Mediation Workspace is the core component of our mediation system and provides a convenient workspace for developers to manipulate services and mediators. Although mediator patterns are depicted by using CPN models as an underlying formalism in Section 3, the protocols of services and mediators are graphically represented in the Mediation Workspace by means of an intuitional notation, like Business Process Modeling Notation (BPMN) [12]. The Mediation Workspace provides a GUI to illustrate the protocols of the two services to be composed. With the mismatch patterns presented in Section 1.2, developers identify all possible protocol mismatches between the services. We have pre-established the basic mediator patterns developed in this paper in a certain base, that is, Mediator Patterns Base (MPB). The basic mediator patterns are well-defined and can be used as modular constructs to develop advanced mediators. And composite mediators can also be stored as patterns in MPB for further use. MPB provides the functionality of flexible extension for mediator patterns. With the identified mismatches, developers select specific mediator patterns from MPB and configure the selected patterns if needed. After that, developers compose the selected mediator patterns to produce a composite mediator. The composite mediator is also based on the underlying CPN models, which is automatically constructed by the Mediation Workspace. Existing tools have provided functionalities that transform service models between BPNN, BPEL and CPNs [15], which can be integrated in our Mediation Workspace.

(3) Mediator Verifier

Since mismatches are identified by developers informally, the service mediator produced in the Mediation Workspace 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. The Mediator Verifier checks whether any deadlock may occur. If any deadlock exists, we consider that the mediation has failed. Otherwise, the mediation is successful.

(4) CPN2BPEL Transformer

Only successful mediator will be performed to the CPN2BPEL. Note that the BPEL-based mediator obtained as output of the CPN2BPEL is only pattern-specific BPEL code. Developers should refine the pseudo-code and generate executable codes.

5. Related Work and Comparisons

Recently there have been a significant number of research works on service mediation, which attempts to address various kinds of composition mismatches [16]. Signature mediation has received considerable attention [3] [17], while protocol mediation is still open. Several formal approaches have been developed to conquer this challenge, such as Automata [18], Process Algebra [19] and Petri nets [9], etc. In [20], an architecture-based approach that can detect and semi-automatically resolve integration mismatches is proposed. And a framework for selecting software components and connectors (mediators) ensuring their interoperability is developed in [21]. The very recent work presented in [13] identifies a few ordering mismatches and provides a semi-automated support to resolve these mismatches. The existing approaches, however, provide only partial solutions and few of them can sufficiently address all possible mismatches. Particularly, mediators developed by these approaches have no control logics and can not resolve complicated protocol mismatches, like mismatches of extra condition, missing condition, or iteration structure, etc.

It has been recognized that patterns can be used to resolve composition mismatches and address protocol mediation [5] [16]. The work in [5] identifies five mismatch patterns and provides templates of BPEL code for developer to build mediators, but these patterns are insufficient. Although two more protocol mismatches derived from repetition structure, namely *Collapse* and *Burst*, are introduced in [1], no approach is proposed to address the two types of mismatches. In [16], the taxonomy of composition mismatches is presented and several patterns are proposed that can be used to eliminate these mismatches. The taxonomy, however, does not sufficiently address protocol mismatches and the compositionability of these patterns is not considered.

6. Conclusions and Future Work

Our ultimate objective is to develop a systematic engineering solution to (semi-) automatically generate service mediators in order to resolve all possible composition mismatches. To achieve it, we have proposed a pattern-based approach for developers to resolve all possible protocol mismatches and glue partially compatible services together. We have presented basic mismatch patterns which can help developers identify the differences between protocols of two services. Based on the identified mismatch patterns, we have devised six basic mediator patterns to resolve these mismatches. The mediator patterns can be flexibly configured by developers, according to the identified mismatches. Moreover, we have addressed the mechanism about the composition of the mediator patterns and pointed out that the patterns can be used to modularly construct advanced mediators. The most advantage of the pattern-based approach proposed in this paper lies in that it can be used to successfully resolve all possible protocol mismatches, especially such mismatches about complicated control logics. To the best of our knowledge, however, few papers in the existing literature present some mediators developed in this paper, like *Storing Controller*, *Constructing Controller* or *Merging Repeater*, or discuss the compositionability of these mediators. Moreover, we have designed the architecture of the Service Mediation System (SMS).

In the future, we plan to focus on the formal approach to verification of the correctness of service mediation. And a systematic solution is expected to be investigated. In addition, further effort will be made to implement the prototype system integrated with the existing IBM WebSphere products.

Acknowledgements

Our work was granted by National Natural Science Foundation of China (No. 60674080), and National High-Tech R&D (863) Plan of China (No. 2006AA04Z151 and No. 2006AA04Z166). In addition, the work was supported by IBM-Tsinghua joint project "Mediator-aided Service Composition" co-sponsored by IBM China Research Lab and National CIMS Engineering Research Center, Tsinghua University.

References

[1] 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.

[2] C. Canal, P. Poizat, and Gwen Salaun, "Adaptation of Component Behaviour using Synchronous Vectors", www.lami.univ-evry.fr/~poizat/documents/publications/RR-CPS05.pdf.

[3] A.M. Zaremski, and J.M. Wing, "Signature Matching: A Tool for Using Software Libraries", ACM Transactions on Software Engineering and Methodology, (1995) 4(2), pp. 146-170.

[4] X.T. Li, Y.S. 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.

[5] B. Benatallah, F. Casati, and D. Grigori, et al., "Developing Adapters for Web Services Integration", Proc. of the 17th Intl. Conf. on Advanced Information System Engineering, (CAiSE 2005), pp. 415-429.

[6] WS-BPEL, OASIS Web Service Business Process Execution Language, in: www.oasis-open.org/committees/wsbpel/.

[7] C. Ouyang, E. Verbeek, and W.M.P. van der Aalst, et al., "Formal Semantics and Analysis of Control Flow in WS-BPEL", Science of Computer Programming, (2007) 67(2-3), pp. 162-198.

[8] A. Martens, "On Compatibility of Web Services", Petri Net Newsletter, (2003) 65, pp. 12-20.

[9] W. Tan, F.Y. Rao, and Y.S. Fan, et al., "Compatibility Analysis and Mediation-aided Composition for BPEL Services", Proc. of the 12th Intl. Conf. on Database Systems for Advanced Applications (DASFAA 2007), pp. 1062-1065.

[10] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros, "Workflow Patterns", Journal of Distributed and Parallel Databases (2003) 14(1), pp. 5-51.

[11] W.M.P. van der Aalst, "The Application of Petri Nets to Workflow Management", Journal of Circuits, Systems and Computers, (1998) 8(1), pp. 21-66.

[12] BPMN, OMG/Business Process Modeling Notation, in: http://www.bpmn.org/.

[13] H.R. Motahari Nezhad, A. Martens, and F. Curbera, et al., "Semi-Automated Adaptation of Service Interactions", Proc. of the 16th Intl. World Wide Web Conerence (WWW 2007), pp. 993-1002.

[14] SCA, Service Component Architecture specifications. In: www.ibm.com/developerworks/library/speci_cation/ws-sca/.

[15] BABEL - Tools, in: http://www.bpm.fit.qut.edu.au/projects/babel/tools/.

[16] S. Becker, A. Brogi, and I. Gorton, et al., "Towards an Engineering Approach to Component Adaptation", Proc. of Architecting Systems with Trustworthy Components, LNCS, Vol. 3938, Berlin, (2006), pp. 193-215.

[17] X. Xie, W. Zhang, "A Checking Mechanism of Software Component Adaptation", Proc. of the 5th Intl. Conf. on Grid and Cooperative Computing, (GCC 2006), pp. 347-354.

[18] D. Yellin, R. Strom, "Protocol Specifications and Component Adaptors", ACM Transactions on Programming Languages and Systems 19 (1997), pp. 292-333.

[19] A. Bracciali, A. Brogi, and C. Canal, "A Formal Approach to Component Adaptation", Journal of Systems and Software (2005) 74(1), pp. 45-54.

[20] M. Tivoli, and D. Garlan, "Adaptor Synthesis for Protocol-Enhanced Component Based Architectures", Proc. of the 5th Working IEEE/IFIP Conf. on Software Architecture, (WICSA 2005), pp. 276-277.

[21] J. Bhuta, C.A. Mattmann, and N. Medvidovic, et al., "A Framework for the Assessment and Selection of Software Components and Connectors in COTS-Based Architectures", Proc. of the 6th Working IEEE/IFIP Conf. on Software Architecture, (WICSA 2007), pp. 44-53.