Mismatch Patterns and Adaptation Aspects: A Foundation for Rapid Development of Web Service Adapters

Woralak Kongdenfha, Hamid Reza Motahari-Nezhad, *Member*, *IEEE*,
Boualem Benatallah, Fabio Casati, and Régis Saint-Paul, *Member*, *IEEE*

Abstract—Standardization in Web services simplifies integration. However, it does not remove the need for adapters due to possible heterogeneity among service interfaces and protocols. In this paper, we characterize the problem of Web services adaptation focusing on business interfaces and protocols adapters. Our study shows that many of the differences between business interfaces and protocols are recurring. We introduce *mismatch patterns* to capture these recurring differences and to provide solutions to resolve them. We leverage mismatch patterns for service adaptation with two approaches: by developing stand-alone adapters and via service modification. We then dig into the notion of *adaptation aspects* that, following aspect-oriented programming paradigm and service modification approach, allow for rapid development of adapters. We present a study showing that it is a preferable approach in many cases. The proposed approach is implemented in a proof-of-concept prototype tool, and evaluated using both qualitative and quantitative methods.

Index Terms—Web services adaptation, mismatch patterns, business protocols, aspect-oriented programming.

1 Introduction

Application integration has been one of the main drivers in the software market during the late 1990s and into the new millennium. The typical approach to integration and to process automation is based on the use of adapters [39]. Adapters wrap the various applications (which are, in general, heterogeneous, e.g., have different interfaces, speak different protocols, and support different data formats) so that they can appear as homogeneous and, therefore, easier to be integrated.

Web services were born as a solution to (or at least as a simplification of) the integration problem [2]. The main benefit they bring is that of standardization. Standardization reduces heterogeneity and makes it easier to develop business logic that integrates different (Web service-based) applications. The possible interactions that a Web service can support are specified at design time, using what is called a *business protocol* [6]. A business protocol specifies message exchange sequences that are supported by the service, for example, expressed in terms of constraints in the order in which service operations should be invoked.

While standardization simplifies interoperability, it does not remove the need for adapters [31]. In fact, although the lower levels of the interaction stacks (e.g., messaging) are standardized, at the higher levels (e.g., business-level interfaces and protocols) what have been standardized are *languages* (e.g., WSDL and BPEL) for their definition, not the specific interfaces or protocols [2], [31]. The result is that services that are functionally similar may have heterogeneous interface and protocol specifications. For example, although different map or driving direction services support XML and use SOAP over HTTP as transport mechanism, they may provide operations that have different names, different parameters, and different business protocols.

In general, there are two ways to approach the adaptation problem: either we develop a third service that mediates the interactions between the two incompatible services (we call it a *stand-alone adapter*), or we modify one of the services to make it compatible with the other. This paper presents a method and a platform for Web services adaptation. In particular, we make the following novel contributions, some of which extend our previous work [5], [29] in this area:

- We study and characterize the problem of adaptation by identifying and classifying different kinds of adaptation scenarios in Web services, focusing on the interface and business protocol levels. Our study shows that many of the differences between interface and protocol specifications are in fact recurring. Therefore, we propose an adaptation methodology by introducing *mismatch patterns* to

- W. Kongdenfha is with the School of Computer Science and Engineering, University of New South Wales Campus, Room 427, K17, Sydney, NSW 2052, Australia. E-mail: woralakk@cse.unsw.edu.au.
- H.R. Motahari-Nezhad is with HP Labs, 1501 Page Mill Rd., MS 1182, Palo Alto, CA 94304. E-mail: hamid.motahari@hp.com.
- B. Benatallah is with the School of Computer Science and Engineering, University of New South Wales (UNSW) Campus, Room 407, L17, Sydney, NSW 2052, Australia. E-mail: boualem@cse.unsw.edu.au.
- F. Casati is with the Department of Information and Communication Technologies, University of Trento, Sommarive st. 14, Povo, 38100 Trento, Italy. E-mail: fabio.casati@dit.unitn.it.
- R. Saint-Paul is with CREATE-NET, Via alla Cascata 56/D, Povo, 38100 Trento, Italy. E-mail: regis.saint-paul@create-net.org.

Manuscript received 11 Mar. 2008; revised 4 Dec. 2008; accepted 2 May 2009; published online 14 May 2009.
For information on obtaining reprints of this article, please send e-mail to: tsc@computer.org, and reference IEEECS Log Number TSC-2008-03-0035.
Digital Object Identifier no. 10.1109/TSC.2009.12.

capture and formalize these recurring differences. Patterns help adapter developers in identifying the actual differences between interface and protocol specifications and in resolving them. Among other information, patterns include a *template* of adaptation logic that resolves the captured mismatch. Developers can instantiate the proposed templates to develop adapters.

- We discuss situations in which modification of the service is preferable to the development of stand-alone adapters. We motivate why in particular an aspect-oriented approach can be leveraged, by generating adaptation logic in the form of *adaptation aspects* woven into the runtime instances of the adapted service. We present an aspect-oriented language for the formulation of mismatch patterns and in particular for specifying the adaptation template of each pattern.
- We present the implementation of our approach in a tool that helps adapter developers in the semiautomatic generation and deployment of adaptation logic. Our implementation supports the browsing and updating of an extensible library of built-in mismatch patterns that assist users in the generation of adaptation logic. The tool allows to develop both aspect-oriented and stand-alone adapters.

With respect to our previous work [5], [29], this paper makes the following extensions and contributions:

1. we discuss stand-alone versus aspect-oriented approach for adaptation and provide guidelines to help developers to decide on situations in which each of the approaches is preferable,
2. we provide a more comprehensive description of aspect-oriented service adaptation,
3. we characterize mismatch patterns and adaptation templates using aspect-oriented approach for adaptation,
4. we present the usage, the implementation, which is only sketched in previous work with an earlier version of the tool [29], and the evaluation of this approach.

The paper is organized as follows: In Section 2, we identify common mismatches between service interfaces and protocols, and propose mismatch patterns for characterizing these mismatches and for adapter development. In Section 3, we propose an aspect-oriented language for adapter specification through service modification. In Section 4, we use the proposed framework to represent the identified mismatches as built-in patterns. Section 5 presents the implementation and the evaluation of the prototype tool, describes how adapter developers can use it through a case study, and presents a comparative study between main approaches for developing adaptation logic. Finally, we discuss related work in Section 6, and conclude and present future work in Section 7.

2 Mismatch Patterns for Web Services Adapter Development

In this section, we propose a methodology for adapter development by providing a classification of common mismatches between service interfaces and business protocols, and introducing mismatch patterns. The intended benefit of this work is to identify possible mismatches between Web services interface and protocol specifications, and to help programmers develop adapters by assisting them through a methodology and semiautomated code development, starting from the interface and protocol definitions. The adapters have the goal of making a service S_r, characterized by interface I_r and protocol P_r, "look like" (interact as) another service S that has interface I and protocol P, so that S_r can then interact with any client that S can interact with.

2.1 Common Mismatches Between Web Service Specifications

Our analysis of the real-world Web services interfaces and protocols shows that many differences between them are recurring. Examples of services in our study include *Mappoint*,[1] *Arcweb*,[2] *Google Checkout*,[3] *XWebCheckout*,[4] *Amazon Web Service*,[5] *Amazon Ecommerce Service*,[6] *PayPal Web Service*,[7] and *PaymentExpress Web Service*.[8] In the following, we characterize these common mismatches (see [5] for details).

2.1.1 Interface-Level Mismatches

To characterize mismatches at this level, we use, as a concrete example, *Mappoint* and *Arcweb* route Web services. Both offer similar functionalities for finding driving routes between two locations but through different WSDL interfaces (operations CalculateRoute and findRoute, respectively). We identify two types of mismatches that are recurring at the level of service interfaces:

Signature mismatch. This type of mismatch concerns the differences that occur when two services with interfaces I and I_r have operations that have the same functionality but differ in operation names, number, order, or type of input/output parameters. In the route web services example, the operation CalculateRoute of *Mappoint* requires one input parameter called Specification whose type is SegmentSpecification. The operation findRoute of *ArcWeb* requires two parameters: routeStops and routeFinderOptions whose types are RouteStops and RouteFinderOptions, respectively. Hence, there is a signature mismatch between the two services.

Parameter constraint mismatch. This mismatch occurs when the operation O of interface I imposes constraints on input parameters, which are less restrictive than those of O_r input parameter in I_r (e.g., differences in value ranges).

2.1.2 Protocol-Level Mismatches

We use the following example from supply chain domain to illustrate mismatches at this level. Assume that protocol P_r of service S_r expects to exchange messages in the following order: clients can invoke login, then getCatalogue to receive

1. www.microsoft.com/mappoint/.
2. www.esri.com/software/arcwebservices.
3. code.google.com/apis/checkout.
4. www.xwebservices.com/Web_Services/XWeb-CheckOut.
5. soap.amazon.com/schemas2/Amazon-WebServices.wsdl.
6. webservices.amazon.com/AWSE-CommerceService.
7. www.paypal.com/wsdl/PayPalSvc.wsdl.
8. www.paymentexpress.com/WS/-PXWS.asmx?WSDL.

the catalogue of products including shipping options and preferences (e.g., delivery dates), followed by submitOrder, sendShippingPreferences, issueInvoice, and makePayment operations. In contrast, protocol P of the client allows the following sequence of operations: login, getCatalogue, submitOrder, issueInvoice, makePayment, and sendShippingPreferences. This is because service S_r does not charge differently according to the shipping preferences. Therefore, clients are allowed to specify their shipping preferences at a final step. We characterize the following mismatches at the service protocol level:

Ordering mismatch. This type of mismatch occurs when protocols P and P_r support the same messages but in different orders.

Extra message mismatch. This mismatch occurs when protocol P_r sends a message that protocol P does not send. In the example above, assume that protocol P_r sends an acknowledgment after receiving message issueInvoiceIn, but protocol P does not produce it.

Missing message mismatch. This mismatch occurs when protocol P_r does not issue a message specified in the protocol P. Consider the opposite case of the previous example.

One-to-many message mismatch. This mismatch occurs when protocol P specifies a single message to achieve a functionality, while protocol P_r requires several messages for the same functionality. Suppose that protocol P requires to receive the purchase order as well as shipping preferences in one message called submitOrderIn, while protocol P_r needs two separate messages for this purpose, namely, sendShippingPreferencesIn and submitOrderIn.

Many-to-one message mismatch. This mismatch occurs when protocol P specifies several messages to achieve a functionality, while protocol P_r requires only one message for the same functionality. It is the opposite case of the previous example.

In the following, we propose the concept of *mismatch patterns* to formalize these differences and also to provide solutions to resolve such recurring problems.

2.2 Mismatch Patterns

Mismatch pattern is a similar notion to that of *design pattern* in software engineering [23]. Mismatch patterns provide a simple and effective abstraction for capturing and resolving differences: besides capturing differences, a mismatch pattern contains the description of an adapter (called *adapter template*) used to resolve that type of captured mismatch. Adapter templates should be instantiated to resolve mismatches for a given pair of services. Indeed, patterns can be used both as guidelines for developers in developing adapters and as input to a tool that generates the adapter code. Table 1 summarizes the structure of a mismatch pattern.

A mismatch pattern has a *name*, a *mismatch type* part that provides a description of the mismatch that is captured, *adapter template*, *template parameters*, and a *sample usage*. The adapter template is parametric (parameters are part of *template parameters* field): to instantiate it for a given pair of interfaces or protocols, the developer needs to provide the template parameters. These are used to generate the adaptation logic from the template. The developer may then use directly the generated adaptation or further customize it with additional business logic.

TABLE 1
The Structure of a Mismatch Pattern

Part	Description
Name	Name of the pattern
Mismatch Type	A description of the type of difference captured by the pattern
Template parameters	Information that needs to be provided by the user when instantiating an adapter template to derive the adapter code
Adapter template	Code or pseudo-code that describes the implementation of an adapter that can resolve the difference captured by the pattern
Sample usage	The sample usage section contains information that guides the developer in customizing (or manually generating) the adapter, by providing examples on how to instantiate the template

The exact specification of adaptation aspect depends on the adapter development approach. In the next section, we present formalisms for the specification of adapter templates. By providing adapter templates for each pattern accompanied with their sample usage and the support for adapter template instantiation in a prototype tool, our approach offers a platform for rapid development of adapters.

3 Aspect-Oriented Service Adaptation

To enable interaction of a service with its partner, one may modify the business logic of the service (e.g., to rewrite the BPEL code). In this case, the adaptation logic is tangled with the business logic. This code tangling makes it difficult to maintain and modify the business logic. When the business logic further evolves, e.g., for business reason or to interact with yet another partner, the developer needs to clean the adaptation logic previously added before modifying the business logic itself. This solution may be acceptable when the service needs to interact with only one partner. However, if we have to enable interactions with many incompatible partners, it would mean creating many versions of the service implementation. In this case, when a change at the business logic level is required, it has to be replicated in all the versions. This makes evolution expensive and error-prone.

From our perspective, it is important to separate the adaptation logic from the business logic. We also argue that adaptation can be seen as a cross-cutting concern, i.e., it is from the developer and project architecture point of view transversal to the other functional concerns of the service. Hence, adaptation logic should be captured in a separate module, called *adapter*, from the business logic. In a nutshell, two approaches can be adopted for adapter development: stand-alone and aspect-oriented adapters. Our framework enables both stand-alone and aspect-oriented approaches for adapter code generation based on adapter templates. However, in this paper, we focus on the use of the aspect-oriented approach for adapter development. The details of developing stand-alone adapters can be found in [5]. We compare these two approaches for adapter development in Section 5.2.2.

Aspect-Oriented Programming (AOP) is a technique that allows the separation of concerns in software development,

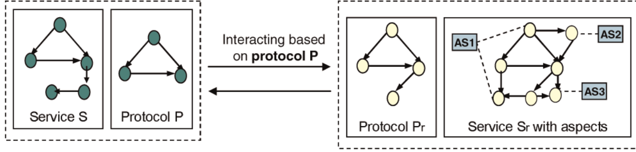

Fig. 1. Adaptation aspects $AS1, AS2$, and $AS3$ for enabling service interoperability.

TABLE 2
Ordering Constraint Mismatch Pattern (OCP)

Query	Generic Adaptation Advice
query(⟨operation⟩,⟨executionPath⟩) executes *before receive* when O_i^{sr} = ⟨operation⟩ AND S_i = ⟨executionPath⟩	OCPStore() { Receive $msgO^p$; Assign $msgO^{tmp} \longleftarrow msgO^p$; }
query(⟨operation⟩,⟨executionPath⟩) executes *around receive* when O_j^{sr} = ⟨operation⟩ AND S_j = ⟨executionPath⟩	OCPForward() { Assign $msgO_j^{sr} \longleftarrow msgO^{tmp}$; Reply $msgO_j^{sr}$ }

making it possible to modularize cross-cutting concerns of a system [27], [22]. As we consider the adaptation logic as a cross-cutting concern, we propose an aspect-oriented approach for Web service adaptation. In this case, each mismatch pattern consists of a template, called *aspect template*, which is specified by a collection of ⟨query, advice⟩ pairs, discussed in the following. When instantiated, the aspect template generates a collection of *adaptation aspects* that will be woven into the service at runtime. This approach is illustrated in Fig. 1, in which adaptation aspects $AS1, AS2$, and $AS3$ are integrated as extensions to a running instance of service S_r to enable its interaction with service S.

Adapters perform activities such as receiving messages, storing messages, transforming message data, and invoking service operations. These tasks can be very well modeled by process-centric service composition languages such as BPEL. We choose BPEL for defining the adapter template. Hereafter, we detail the structure of aspect templates, a collection of ⟨query, advice⟩ pairs.

Advice. An advice defines the adaptation logic for resolving the difference captured by a mismatch pattern. It requires parameters (e.g., a transformation function to mediate the difference between operation signatures) that are used to generate an adaptation code skeleton from the template. As mentioned before, BPEL provides notations and concepts that are appropriate for the adaptation specification and implementation. We chose it as the language to express adaptation advices.

To describe how adaptation logic can be modeled and implemented using aspect-oriented approach, we present the Ordering Constraint Pattern (OCP) in Table 2. This pattern is accompanied with an aspect template consisting of two ⟨query, advice⟩ pairs. The first advice, namely OCPStore, comprises of two actions that are used to resolve a mismatch occurs when a message $msgO^p$ is sent from service S, but service S_r does not expect it at this state. OCPStore, therefore, receives and stores message $msgO^p$ for later use. When the process execution reaches operation O_j^{sr}, OCPForward assigns the value of message $msgO^p$ to message $msgO_j^{sr}$ to enable the execution of the operation O_j^{sr}. The exact locations, where these adaptation advices need to be executed, are defined in the query section of the template, and is discussed in the following.

Query. A query expresses a process execution point, also known as *joinpoint* in the context of AOP [27], [22], where a set of actions defined in the advice section of the template will be executed to mediate the differences between services (e.g., when such a message is received, or when a message comes from a business partner, etc.) In general, there are two main approaches for joinpoint expression in the context of AOP [22]. The first approach consists in the expression of joinpoints only on the service code constructs. The second approach consists in directly expressing joinpoints on not only service code but also runtime execution context.

In the context of service adaptation, we have observed that the requirement of the query language for expressing joinpoints is not only limited to the identification of service code, but also on the actual messages exchanged with the client, and, in general, by the runtime execution context. To illustrate this requirement, consider the example of the supply chain scenario introduced in Section 2. Assume that the service S_r allows two different interaction paths with either unregistered (as described in Section 2) or registered clients. The interaction path for registered clients is as follows: after submitting an order, process S_r allows registered clients to send messages issueInvoice and makePayment, respectively. A client does not need to resend message sendShippingPreferences as it has already been provided and stored in the system when the client made the registration the first time. In this example, an ordering mismatch between service S_r and its client only happens when the client takes the unregistered interaction path, otherwise the two services are compatible. Thus, it is the choice of interaction path that triggers the adaptation need. This example shows that in the service adaptation context, the query language needs to be able to express conditions on the runtime context, i.e., by how the service is actually used by a client or how it is executed.

Intuitively, for the purpose of service adaptation, we expect the query language to be able to identify 1) operations (with or without a certain signature) to enable the resolution of interface-level mismatches, and 2) interaction paths (that are or are not presented in a protocol) to enable the handling of protocol-level mismatches. The latter means that the query language must be able to discriminate between the various execution paths that lead to or follow an activity of the service. In both cases, what is done is the identification of a BPEL activity where adaptation is needed, e.g., the activity where a signature mismatch occurs, or the first activity of a sequence that does not have any correspondence at the protocol level in the client.

Since we assume that services are implemented in BPEL, a query language that operates on BPEL code such as BPQL [4] could be a choice. However, using a query language that focuses on the identification of code constructs would force us to include, as part of the advice, some code to evaluate those runtime conditions. Hence, the approach that expresses runtime conditions directly in the

```
<query>           ::=  query( [<param>[,<param>]*] )
                       executes <location> <activity>
                       when <condition>
<param>           ::=  id[;id]*
<location>        ::=  before|after|around
<activity>        ::=  receive|reply|invoke
<condition>       ::=  <pred>[AND<pred>]
<pred>            ::=  <context object>=<param>
                       |<context object>!=<param>
<context object>  ::=  partnerLink|portType|operation|inputVariable
                       |outputVariable|type|executionPath
```

Fig. 2. Semiformal syntax for query language.

query language has been preferred. This is because it groups together all advice execution conditions in the query and frees the advice code from any runtime conditions, and thus results in a more readable code and advices that are more generic.

We, therefore, propose a joinpoint query language that can express the need of adaptation advices on the service code, as well as runtime execution context. We assume that services are implemented in BPEL, though the concepts and requirements are independent of the specific process language adopted. The query language is, therefore, designed specifically to BPEL constructs. Fig. 2 presents the syntax of our proposed query language that satisfies the above requirements. This query language allows the definition of joinpoints on the BPEL code constructs, such as operation, portType, etc.

While it shares some common characteristics with query languages that operate at the code level such as BPQL, the main differences are as follows: 1) It can express conditions on service interaction paths and 2) it includes keywords for specifying the relative location of the joinpoint to the BPEL activity that matches the specified conditions (i.e., the *before*, *after*, or *around* keywords). These concepts are needed to achieve a self-contained query language able to express all the conditions necessary for identifying joinpoints in the adaptation context.

As shown in Fig. 2, the query takes parameters (*param*) that correspond to BPEL constructs (i.e., operation, input variable, output variable, partnerLink, and portType), or an execution path (i.e., a sequence of previously exchanged messages). These parameters are matched against some conditions (*context object*) at runtime to identify joinpoints where adaptation advices should be executed. The `executes` statement specifies whether the execution of adaptation advice should be performed *before*, *after*, or *around* (i.e., in place of) a BPEL activity that matches the joinpoint query.

Consider again the supply chain example, in which the OCP shown in Table 2 is used to solve its ordering mismatch. In this case, the OCPStore needs to be executed before the *receive* activity of operation sendShippingPreference to receive and store the message issueInvoiceIn, which is not expected at this state. As mentioned before that the ordering mismatch only occurs when a specific interaction path (unregistered) is taken; hence, the query parameters of the OCPStore in this example are <operation>=sendShippingPreferences and <executionPath>=unregistered. These

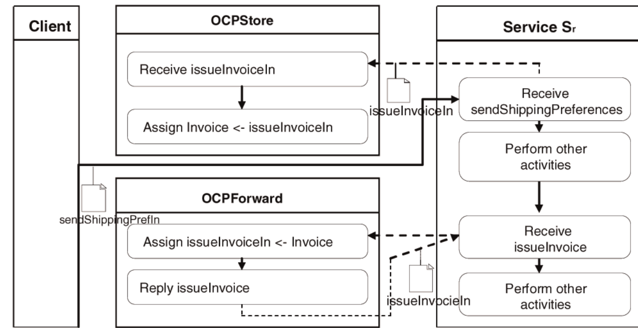

Fig. 3. Sample usage of the aspect template specified in the OCP.

parameters will be evaluated, at runtime, against the currently executing operation (O_i^{sr}) and the execution path (S_i) of the adapting service.

Fig. 3 shows a sample usage of OCP at runtime. Before the process executes the *receive* activity of operation sendShippingPreferences, OCPStore receives message issueInvoice and stores it in a temporary variable Invoice. After the completion of OCPStore, the process continues to execute the *receive* activity of operation sendShippingPreferences. When the message issueInvoice is required by the S_r, the OCPForward takes its value from variable Invoice. OCPForward is executed *around* (instead of) the *receive* activity of operation issueInvoice. Hence, after the completion of OCPForward, the process continues other activities without performing the *receive* activity of operation issueInvoice. This is because the message issueInvoiceIn has already been received earlier.

Deployment of adaptation aspects. The above discussion considers only the query language syntax, not the actual deployment of the solution. Choosing a query language that incorporates runtime conditions also allows for aspect weaving done either at compile time or at runtime. In the compile-time deployment model, a new BPEL code would be generated with advices preceded by runtime conditions. In a runtime deployment model, a specially modified query engine is required to evaluate runtime conditions based on the execution context it maintains, leaving the original code unmodified. While both models are viable, the first one (compile time) imposes to incorporate some additional logic in the advices. This logic is not the part of adaptation logic, but it is required to maintain information regarding the service's execution context (e.g., the interaction pattern taken by the client). We, therefore, chose the second (runtime) deployment model which, in addition to its greater simplicity, also allows dynamically plugging and unplugging adaptation aspects. The query engine for this deployment model is presented in Section 5.

4 Characterization and Resolution of Common Interface and Protocol-Level Mismatches

In this section, we use the proposed framework for mismatch pattern representation to capture solutions for the common mismatches between Web service specifications, identified in Section 2.1. We particularly focus on the

TABLE 3
Signature Mismatch Pattern (SMP)

Query	Generic Adaptation Advice
`query(`⟨inputType⟩`)` `executes` *before receive* `when` P^{sr}=⟨inputType⟩	SMPInput(⟨T⟩) { `Receive` $msgO^p$; `Assign` $msgO^{sr} \longleftarrow \langle T\rangle(msgO^p)$; `Reply` $msgO^{sr}$; }
`query(`⟨outputType⟩`)` `executes` *before reply* `when` P^{sr}=⟨outputType⟩	SMPOutput(⟨T⟩) { `Receive` $msgO^{sr}$; `Assign` $msgO^p \longleftarrow \langle T\rangle(msgO^{sr})$; `Reply` $msgO^p$; }

TABLE 4
Missing Message Pattern (MMP)

Query	Generic Adaptation Advice
`query(`⟨operation⟩`)` `executes` *after receive* `when` O^{sr} = ⟨operation⟩	MissingMessage(⟨T⟩, ⟨Var^{sr}⟩) { `Receive` ⟨Var^{sr}⟩; `Assign` $msgO^p \leftarrow \langle T\rangle(Var^{sr})$; `Reply` $msgO^p$; }

aspect-oriented approach for adapter development and use the language proposed in Section 3. However, due to space limitation, in this section, we only discuss three mismatch patterns in detail: Signature Mismatch Pattern (SMP), Missing Message Pattern (MMP), and One-to-Many Pattern (OMP). Readers can refer to [28] for the discussion on other patterns.

4.1 Signature Mismatch Pattern

This pattern is used to handle a signature mismatch. It consists of two parts, i.e., SMPInput and SMPOutput, as shown in Table 3. SMPInput intercepts an incoming message $msgO^p$ from a client with protocol P, then uses a transformation function $\langle T\rangle$ to transform the data type of message $msgO^p$ into a data type required by message $msgO^{sr}$ of protocol P_r. Similar actions are specified in SMPOutput to resolve mismatches on the outgoing messages of the service.

To instantiate the aspect template of SMP, the developer provides *inputType* (respectively, *outputType*) as query parameter and XQuery/XSLT transformation functions as advice parameters to SMPInput (respectively, SMPOutput). In the route Web services example described in Section 2, SMPInput takes CalculateRouteType as its query input and two transformation functions TransformStops and TransformOptions as advice inputs. These functions are responsible for actually transforming the data types of the message parameters.

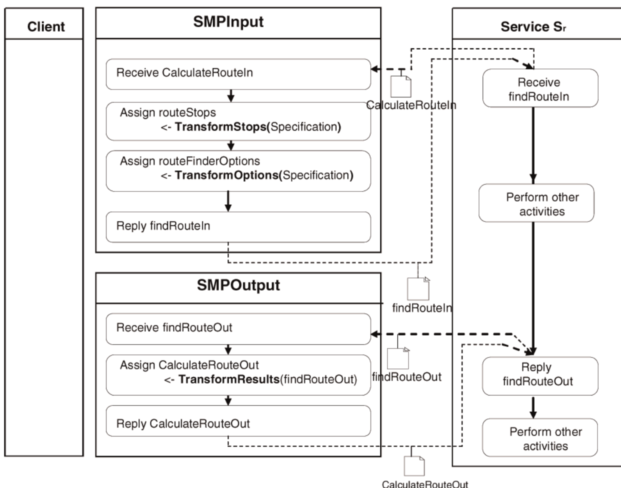

Fig. 4. Sample usage of SMP.

Fig. 4 presents a sample usage of SMP at runtime. SMPInput first intercepts an incoming message CalculateRouteIn of operation CalculateRoute specified by protocol P, then computes the values of routeStops and routeFinderOptions (i.e., input parameters of the operation findRoute) from the value of the parameter Specification, via XQuery transformation functions. After the message findRouteIn has been created, the SMPInput replies it to our system. It should be emphasized that, in general, our system passes data being sent or received by the current joinpoint activity (i.e., a messaging activity such as receive, reply, invoke) to the corresponding advices and vice versa (as explained in Section 5.1). In this case, our system passes the findRouteIn message of the SMPInput advice to the input variable of the receive activity as specified by protocol P_r such that service S_r can continue. Similar actions are specified in the SMPOutput to resolve mismatches on the outgoing messages of service S_r.

4.2 Missing Message Pattern

This pattern is used to solve the missing message mismatch. As shown in Table 4, MMP generates a message $msgO^p$, after the *receive* activity of operation O^{sr} of service S_r. This message is then sent to service S according to protocol P. The message generation is expressed by an XQuery function.

Fig. 5 shows a sample usage of MMP. After the *receive* activity of operation issueInvoice, MMP generates message InvoiceAcknowledgement and sends it to the client. This InvoiceAcknowledgement message requires the variable purchase order POVar of service S_r in its generation. The user needs to provide an XQuery function GenerateInvoiceAck to be used to generate the message InvoiceAcknowledgement from the variable POVar.

As we have described in Section 4.1 that, in general, our system passes the data being sent or received by the current joinpoint activity (i.e., a messaging activity) to the corresponding advices. However, it is also possible to pass to advices some additional contextual information, such as the internal execution data of service S_r, and a list of messages being sent or received by service S_r, which is maintained by

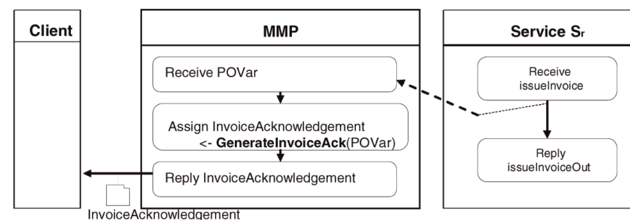

Fig. 5. Sample usage of MMP.

TABLE 5
One-to-Many Mismatch Pattern (OMP)

Query	Generic Adaptation Advice

query(⟨operation⟩)
executes *before receive*
when O_i^{sr}=⟨operation⟩
AND S_i = ⟨executionPath⟩
	OMPSplit(⟨T⟩) {
	Receive $msgO^p$;
	Assign $msgO_i^{sr} \leftarrow \langle T\rangle(msgO^p)$;
	While(j≤count(⟨T⟩)) {
	Assign $MSGO_j^{sr} \leftarrow \langle T\rangle(msgO^p)$;
	Reply $msgO_i^{sr}$; }

query(⟨operation⟩)
executes *around receive*
when O_j^{sr}=⟨operation⟩
AND S_j = ⟨executionPath⟩
	OMPForward() {
	Reply $MSGO_j^{sr}$;
	}

our system. For example, the variable POVar may not be part of message currently exchanged. Rather it is either an internal variable of the service S_r, or a part of a message previously sent or received by service S_r. In the latter case, both stand-alone adapter and aspect-oriented approaches are possible to generate the message InvoiceAcknowledgement since the stand-alone adapters can keep track of messages exchanged between two processes, and the adaptation advices can access the list of exchanged messages maintained by our system. However, if this purchase order number is an internal information of service S_r, the message InvoiceAcknowledgement can only be generated when the pattern is implemented using the aspect-oriented approach.

4.3 One-to-Many Pattern

This pattern is used to resolve the one-to-many mismatch. As shown in Table 5, OMP receives a single message and *splits* it into a set of messages.

OMP consists of a OMPSplit and a set of OMPForward. OMPSplit intercepts an incoming message $msgO^p$ from a client with protocol P, then uses it to generate message $msgO_i^{sr}$ required by service S_r. OMPSplit also splits message $msgO^p$ into a set of messages $MSGO^{sr}$ and stores them. The generation of message $msgO_i^{sr}$ and $MSGO^{sr}$ are expressed by XQuery functions. Afterwards, the generated message $msgO_i^{sr}$ is sent back to process S_r, while messages $MSGO^{sr}$ will be individually used, when needed, by OMPForward to create messages required by a set of operations O_j^{sr}. The number of OMPForward, to be instantiated, depends on the number of operations that require information from $msgO^p$.

Fig. 6 shows a sample usage of OMP at runtime. OMPSplit first intercepts message submitOrderIn, then generates input messages of the operations submitOrder and sendShippingPreferences of service S_r from the submitOrderIn message, using XQuery transformation functions provided by the user, namely SplitShipping and SplitOrder. The message submitOrderIn is sent back to the service S_r, while the message sendShippingPreferencesIn is stored in the advice. When the message sendShippingPreferences is required by the service S_r, i.e., during the *receive* activity of the operation sendShippingPreferences, OMPForward sends the message sendShippingPreferencesIn to service S_r.

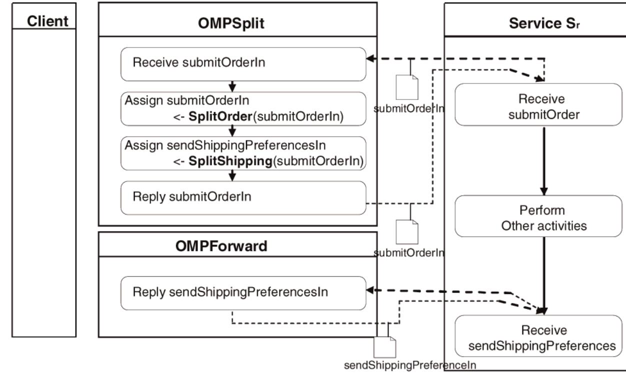

Fig. 6. Sample usage of OMP.

In this section, we have presented a repository of mismatch patterns and described how they can be instantiated to handle the differences that each of them capture. We also provide a prototype tool to support the developer during the development and deployment of the adaptation logic. The prototype is discussed in more details in the next section.

5 Implementation and Evaluation

In this section, we discuss the prototype implementation and the evaluation results.

5.1 Implementation

The approach for adapter development proposed in this paper has been implemented in a prototype tool that consists of two components depicted in Fig. 7: 1) pattern-based mismatch identification, and 2) adaptation code generation. The pattern-based mismatch identification component incorporates a tool for managing a collection of mismatch patterns (i.e., taxonomy of mismatches and their adaptation). Users can add, modify, or remove mismatch patterns to evolve the library. Web service protocols are handled by a

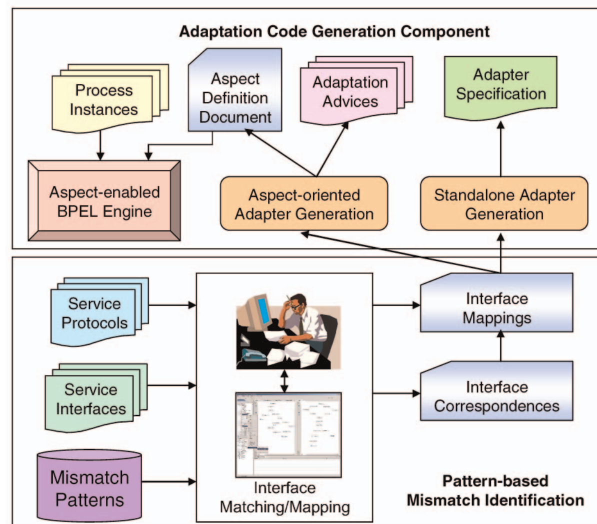

Fig. 7. The architecture of mismatch patterns-based adapter development.

```
<aspect>
 <mismatch template="SMPInput">
   <query parameter= "inputType" value="CalculateRouteType">
      <executes location="before" activity="receive"/>
   </query>
   <advice name="RouteReqSMPInput"/></mismatch>
 <mismatch template="SMPOutput">
   <query parameter="outputType" value="RouteType"
      <executes location="before" activity="reply"/>
   </query>
   <advice name="RouteResSMPOutput"/></mismatch> ... </aspect>
```

Fig. 8. An Aspect Definition Document (ADD).

protocol editor which has been implemented as part of the prototype. The adaptation code generation component allows managing mismatch patterns and generating both stand-alone and aspect-oriented adaptation logic. The stand-alone adapter approach has been presented in our previous work [5], [32]. In this paper, we focus on the implementation of the aspect-oriented approach.

The aspect-oriented adapter code generation component relies on a set of advice templates which are implemented using XQuery templates. To instantiate adaptation advices, users need to provide parameters to the advice templates, i.e., XQuery functions. Once instantiated, advice templates are used to generate the adaptation advices in terms of BPEL files which are deployed as Web services. While instantiating each adaptation advice, users also specify the process execution point (joinpoints) where the advice needs to be executed. Joinpoints are specified in terms of queries on the process code. Both the joinpoint queries and their corresponding advice for each of the mismatch are described in a single file called the Aspect Definition Document (ADD). An example of ADD document for the route web service example is shown in Fig. 8. It consists of a set of mismatch elements, SMPInput and SMPOutput. The mismatch element SMPInput specifies that a joinpoint is matched *before* a *receive* activity if the incoming message has *input type* CalculateRouteType. At this joinpoint, an advice named RouteReqSMPInput is executed. Note that in this example, it is possible that different operations in the process receive messages of the same *type*. For example, operations findRoute and Distance, which expect message of type findRouteType, receive messages CalculateRouteType. In this case, *receive* activity of both the operations are matched by our query. The advice RouteReqSMPInput can be reused to solve mismatches at both joinpoints.

When several mismatches need to be addressed at the same joinpoint, the order by which the advices are executed corresponds to the order specified by the user. For example, suppose that two distinct adaptation advices from SMPInput template, namely SMPStop and SMPFinder, are used to transform the parameter Specification of message CalculateRouteIn into the parameters routeStop and routeFinderOption of message findRouteIn. These two advices need to be executed at the same joinpoint, i.e., when a message of type CalculateRouteType arrives. The order in which these two advices are executed is important since if the advice SMPFinder is applied before advice SMPStop, the intermediate message resulted from applying a transformation specified in SMPFinder on the Specification may not have the right structure to be transformed by SMPStop. In fact, the ordering of advices yields naturally from the way

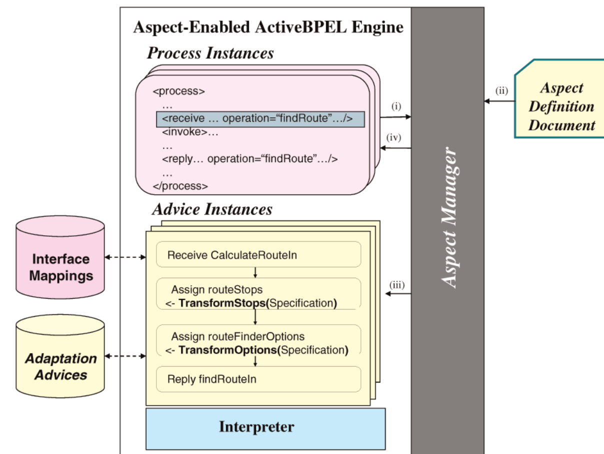

Fig. 9. The deployment of adaptation aspects at runtime.

mismatch patterns are used at design time: In this example, the user would first apply the SMPStop transformation and then, on the basis of the transformed message, apply the second transformation SMPFinder. The ADD document preserves this ordering by the order in which advices are specified in the document. At runtime, our aspect-enabled BPEL engine interprets the ADD document to decide if advices need to be executed at a given joinpoint and, if so, in which order.

In our prototype, the aspect-enabled BPEL engine has been developed by extending the ActiveBPEL[9] engine with an *aspect manager*. The aspect manager is responsible to check before and after the execution of each activity of a business process if an adaptation advice needs to be executed. The aspect manager is implemented using AspectJ[10] and itself woven with the ActiveBPEL code. This enables the aspect manager to access execution data of the business processes (step (i) in Fig. 9). We collect contextual information of each activity executed by the engine such as *activity name*, *activity type*, *partner links*, *port types*, *operation names*, and *variable names*. For messaging activities, i.e., *receive*, *reply*, and *invoke*, the aspect manager also collects the context of messages sent and received by the activities. Specifically, the aspect manager collects the *type* of messages, which corresponds to their qualified names as specified in WSDL file, as well as the name and value of *parameters* within the messages.

The aspect manager stores the execution information in an internal data structure and uses this information to check if there is a joinpoint defined on the activity currently being executed by the engine. To this end, the aspect manager matches the execution information against the query definitions presented in the ADD document (step (ii) in Fig. 9). When a match is found, the aspect manager loads the corresponding advice as specified in the ADD document (step (iii) in Fig. 9). Finally, after the completion of the adaptation advice, the BPEL engine resumes the normal process execution (step (iv) in Fig. 9). Consider the ADD document in Fig. 8. When an activity is executed, the aspect manager looks at the collected contextual information and

9. www.activebpel.org.
10. www.eclipse.org/aspectj.

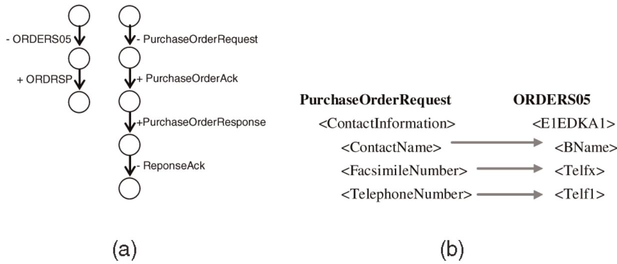

Fig. 10. Service descriptions of SAP R/3 and RosettaNet PIP 34A: (a) business protocols and (b) message details.

```
declare variable $transform external;
<process ... ">
   <variables>
      <variable messageType="ns1:SigRequest" name="SigReq"/>
      <variable messageType="ns1:SigResponse" name="SigRes"/>
   </variables>
   <sequence>
      <receive name="SMPInputReceive" variable="SigReq"
          operation="Signature" partnerLink="adapterPL"
          portType="ns1:adapterPT" createInstance="yes"/>
      <assign name="SMPInputLogic"> {$transform} </assign>
      <reply name="SMPInputReply" operation="Signature"
             partnerLink="adapterPL" portType="ns1:adapterPT"
             variable="SigRes"/> </sequence> </process>
```

Fig. 11. Aspect template for SMPInput.

checks against the specified queries to identify if the activity type is *receive*, and the incoming message has *type* CalculateRouteType. If it is the case, this activity is matched and thus a corresponding advice is executed. The execution of advices is performed by the BPEL engine. To do so, the aspect manager adds advice activities to the process execution queue, thus they will be executed as if they were regular activities of the process itself.

5.2 Evaluation

To illustrate the usage of the proposed approach, we explain a case study in which we have used our prototyped implementation to develop adapters in a real-world interaction scenario. We then provide both qualitative and quantitative evaluations of our approach. The qualitative evaluation is based on a comparative study between stand-alone and aspect-oriented adapter development approaches, while the quantitative evaluation is based on the adoption of the CK metrics [14] to compare the aspect-oriented and code-modification approaches.

5.2.1 Use Case

We evaluated the proposed approach using a real-world scenario. Consider a service S_r implemented following RosettaNet PIP 3A4 specification and another service S that has been implemented following SAP R/3 specification (scenario taken from [1]). These two services provide similar APIs for purchase order management. However, there are differences in the interface definition (message names, parameter numbers, and types) and in how they exchange messages to fulfill a functionality. For example, Fig. 10a shows the protocols of the two services for placing an order. RosettaNet protocol specifies that: the service expects to receive a message PurchaseOrderRequest (as shown by a -PurchaseOrderRequest), then sends a message PurchaseOrderAck (as shown by +PurchaseOrderAck) as an acknowledgement to its client. Upon completion of the purchase order operation, the customer receives a message PurchaseOrderResponse and then sends a message ResponseAck as its acknowledgement to the supplier. On the other hand, the SAP protocol specifies that the service expects to receive a message ORDERS05, and then sends a message ORDRSP as its response.

The following details how a developer can use our tool to develop an adapter for the aforementioned two services:

Step 1: Mismatches identification. The mismatch pattern taxonomy acts as a knowledge base, suggesting possible mismatches between the interfaces and protocols of the two services to adapt and helping the developer in identifying actual mismatches. In the case study, the developer consults our pattern taxonomy and finds that there are two signature mismatches between messages PurchaseOrderRequest and ORDERS05 (Fig. 10b) as well as messages PurchaseOrderResponse and ORDRSP. The developer also finds that there is an extra message (PurchaseOrderAck) and a missing message mismatches (ResponseAck) between the two services.

Step 2: Instantiation of Adaptation Templates. In the case study, the developer adopts the aspect-oriented adaptation approach and instantiates four templates (i.e., SMPInput, SMPOutput, EMP, and MMP) to resolve the mismatches mentioned above. Due to space limitation, we cannot discuss all instantiation scenarios. Instead, we discuss in detail the instantiation of POReqSMPInput. The reader can refer to [28] for further information. For now, let us consider the XQuery template for SMPInput as shown in Fig. 11. To instantiate this template, the developer needs to provide a transformation function TransformPO to the variable *transform* of the SMPInput template. Transformation functions can be authored using third party software (e.g., Microsoft Biztalk, IBM Websphere Integration Developer). These tools provide effective schema mapping functionalities that can be used for this purpose. In our case study, we use the IBM Websphere Integration Developer. The result of an instantiation is a BPEL process POReqSMPInput that can be deployed to solve the mismatch.

After instantiation of the adaptation advices, the developer needs to create a deployment logic (i.e., ADD document specifying how the advices are integrated with the existing service). The ADD document for this case study can be specified similarly to that shown in Fig. 8.

When all the necessary documents have been created, the developer can deploy them to solve mismatches. By the notion of adaptation template, the developers' task is reduced from the creation of adaptation logic from scratch to that of instantiating predefined patterns. In conclusion, our prototype tool supports the adapter developer to rapidly develop Web service adapters. The main task of the developer is to identify the mismatches and instantiate their corresponding templates. The tool then automatically generates the adaptation logic and deploy it to mediate the differences.

5.2.2 Qualitative Evaluation

Fig. 12 presents a schematic comparison of the stand-alone and aspect-oriented approaches for adapter development, in cases where a service S_r with protocol P_r has to be adapted

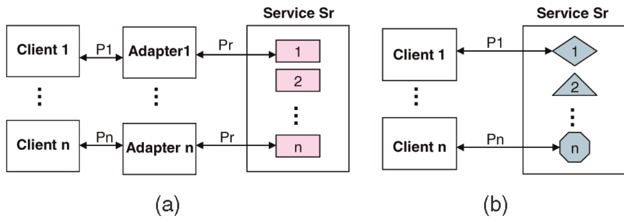

Fig. 12. Schematic comparison of adaptation in stand-alone and aspect-oriented approaches: (a) service instances $1, \ldots, n$ are the same; (b) service instances $1, \ldots, n$ are modified with respective adaptation aspects.

to n client services, with heterogeneous protocols $P_1, \ldots, P_n$. In the stand-alone adapter approach, n adapters, one per each client, have to be developed to make the interactions possible according to protocol P_r. On the other hand, in aspect-oriented approach, the runtime instance that is formed for interacting with each client has to be modified with respective adaptation aspects. Each of these two approaches has characteristics that make them suitable for certain situations. In the following, we review each of them.

Aspect-oriented service adaptation. An aspect-oriented approach to adaptation presents several characteristics that make it preferable for the development of adaptation code compared to stand-alone adapters. In particular, using aspect-oriented approach to realize adapter templates for mismatch patterns further expedites rapid adapter development in our approach. This is because there is no need for a new service (as is the case in separate process and stand-alone adapter) to be developed, rather instances of the existing service are updated at runtime. Other characteristics are discussed as follows:

Context-aware service adaptation: The intertwining of adaptation aspects inside a service allows the aspects to access internal state and variables of the service. This increases the possibility of service adaptations that require contextual information of the service, e.g., a message generation that requires internal variables of the service as discussed in Section 4.2. Note that the patterns are generic and reusable, while the instantiation of patterns is specific and allows to incorporate contextual information as the input parameters of the pattern.

Recovery: The adaptation aspects share execution context of the adapting service. When an error occurs, the recovery can be performed by analyzing the internal state of the service. This is easier than handling exceptions of two separate processes (i.e., adapter and service), which would require correlation of log entries.

Reusability: The aspect-oriented approach promotes reusability of adaptation code when many execution points require the same adaptation logic, e.g., any operation that receives messages of a specific type can reuse the same aspect for message transformations. There is no need to generate individual adaptation logic for each single message as is the case of the stand-alone adapter. Consequently, the number of adaptation logic needed to be generated is reduced.

Separation of concerns: The aspect-oriented approach cleanly separates the adaptation concern from the service functionality. The service developers are oblivious to the adaptation concern since they do not need to write gluecode between adaptation logic and service implementation (as is the case in the stand-alone adapter in which gluecode appear in several places in the service code). The study in [24], [25] also shows that implementing adapters using AOP can better separate the adaptation concern from the functionality of the base programs.

Stand-alone service adaptation. A stand-alone adapter is implemented as a complete single business process comprising a set of adaptation activities. The interdependencies between these activities are well-defined (comparing to aspect-oriented approach), and thus simplify the *understandability*.

Trade-offs. In some cases, the intended adaptation scenarios need to be taken into account when selecting the adapter development approaches. These characteristics and situations are discussed as follows:

Overhead: We consider overhead as the time spent by the adapters in performing activities that are not part of the adaptation logic. This characteristic depends on the intended adaptation scenarios, specifically the number of messages that requires adaptation. When such a number is small, the aspect-oriented approach is preferable. This is because the adaptation aspects will be invoked only for those messages that require adaptation, while all messages need to pass through the stand-alone adapter even if no adaptation is needed. However, aspect-oriented approach introduces overhead for every single message to check if an adaptation is required. Hence, when the number of mismatches is large relative to the total number of messages, the stand-alone adapter approach might be reasonable.

Maintainability: In the context of service adaptation, we consider maintainability as the impact of changes in the service implementation on the adaptation logic. The impact of changes is spread over multiple aspects comprising the adaptation logic, while it is in one place in the case of stand-alone adapters. However, in the aspect-oriented approach, the developer can update the adaptation logic by dynamically plug/unplug the aspects, without interrupting the service interactions (as is the case of stand-alone adapters that need to be suspended and updated).

To conclude, the aspect-oriented adaptation is preferable when developers consider the importance of reusability, relative possible number of mismatches to be resolved, recovery, and separation of concerns. On the other hand, when considering the understandability of adaptation logic, the developers may consider the use of stand-alone adapters. In the other case, the intended adaptation scenarios need to be taken into consideration. In cases, when the relative number of mismatches is large, the stand-alone adapter approach is reasonable. However, when we require access to service implementation and runtime environment, and the relative number of mismatches is small, the aspect-oriented approach is preferable for service adaptation development.

5.2.3 Quantitative Evaluation

To provide a quantitative evaluation of our approach, we have created a BPEL process for a supply chain service with 60 activities. We considered four different interaction scenarios between this process and its partners:

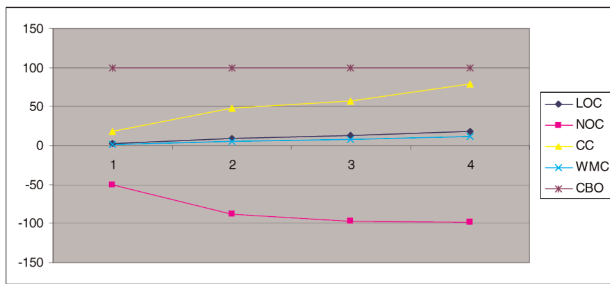

Fig. 13. Comparison of CK metrics in different scenarios.

1. scenario 1: there is only one mismatch, i.e., signature mismatch, in the interaction,
2. scenario 2: this interaction consists of seven mismatches, i.e., signature, parameter constraint, ordering, extra message, missing message, split, and merge mismatches,
3. scenario 3: there are 30 mismatches in this scenario, and finally
4. scenario 4: this interaction consists of 60 mismatches.

We then created adaptation logic to resolve mismatches in these scenarios using two different approaches, i.e., code modification and aspect oriented. To evaluate the impact of these two adaptation development approaches, we made two assumptions in our evaluation. First, the same adaptation logic is written when developing using either the aspect-oriented or the code-modification approach. Second, in code modification, the business logic is directly modified to accommodate the adaptation logic. We have adapted the CK metrics [14] in our evaluation as follows:

The size of the development is measured by the Line Of Code (LOC) and Number Of Classes (NOC). LOC is the number of lines of code in the BPEL process and aspects. NOC refers to the number of classes in the system, which are the number of process and aspects in our context. The Cyclomatic Complexity (CC) and the Weighted Method per Class are metrics that have been used to measure the complexity. CC represents the number of conditional and loop statements in the BPEL process and adaptation advices. The WMC is a metric that measures complexity by the number of activities in the BPEL process and the number of advices associated to the process. In the latter case, we weighted advices based on an assumption that an advice with more activities than another is likely to be more complex. Finally, the Coupling Between Objects (CBO) metric measures the coupling between business and adaptation logic.

Fig. 13 summarizes the results of our evaluation. The Y-axis in the graph illustrates the percentage values that represent the differences between the aspect oriented and code modification. A positive percentage means that the aspect-oriented approach was superior, while a negative percentage means that it was inferior. The results show that the aspect-oriented approach is favorable with respect to the complexity as described by the CC and WMC metrics. The most significant value proving this fact is the CC metric, which is decreasing by 48.27 percent in scenario 2. This is because, in the code-modification approach, some additional logic needs to be included to select behavior based on runtime context. In terms of the size, the use of aspects has increased the NOC by 87.5 percent according to the number of aspects introduced by our approach. Although this increment cannot be neglected, the use of aspects contributes to the decrease of the LOC in which developers need to write. This is because in our approach, adaptation logics are partially generated from patterns; hence, the actual LOC that developers need to develop is less than what is shown in the result. The aspect-oriented approach also reduces the coupling between adaptation and business logic as shown by the increasing of the CBO metric by 100 percent. This is because, in the aspect-oriented approach, adaptation logics are captured in separate modules. In summary, the results show that the proposed patterns and aspect-oriented framework for adapter development has reduced complexity and coupling, and in that sense is easier to understand and maintain. Reader can also find detailed discussion of metrics, evaluation methodology, and results in [28].

6 Related Work

The problem of adapting interaction models in software has been extensively studied in different contexts, more notably in the area of software components (e.g., [39], [7]), and Web services (e.g., [5], [34], [8], [20], [29]). In addition, AOP has received a significant attention in software components [15], [18], [36], [10] and in Web services for the implementation of cross-cutting concerns [22], [33], [17], [11], [38]. In the following, we position our work with respect to the above-mentioned efforts.

Software components adaptation. Several approaches have been proposed for automatic generation of protocol-level component adapters [39], [7], [26]. These approaches focus on stand-alone adapter development and assume that there are no mismatches at the interface-level or that the mapping between component interfaces is provided. Becker et al. [3] identify the most common mismatches between software components at the interface, protocol, and quality of service levels. They also explore the application of software patterns such as adapters, decorators, etc., to adapt functional and nonfunctional differences of software components. However, the component mismatches and patterns are presented at an abstract level. We focus on Web service interfaces and protocols, present concrete specification of mismatch patterns, and present a semiautomated approach for adapter code generation including an aspect-oriented approach for service adaptation.

In the area of software engineering, there are approaches for automatic identification of mismatches between software components based on their interface and protocol specifications [40], [41], [21], [13]. These approaches provide a measure of similarity or differences of software components, but do not aim at their adaptation. Nevertheless, automated approaches for identification of mismatches are limited, and the approach proposed in this paper based on the characterization of mismatch patterns complements them and helps the adapter developer to identify and capture most of the possible differences that are not detected by automated approaches.

Web services adaptation. The problem of Web services adaptation has received a significant attention [5], [20], [8], [29], [32]. To the best of our knowledge, our work [5] was

the first to characterize the problem of Web services adaptation and to propose the concept of mismatch patterns for adapter development. This is a pioneer work that has built the foundation for other recent work in this area. In particular, in addition to the proposed patterns presented in [5], Dumas et al. [20] have identified two other mismatch patterns and proposed operators to handle mismatches. These operators can be composed when developing stand-alone adapters. Li et al. [30] adopt the mismatch patterns framework to identify five extra mismatch patterns at the interface level and the protocol level.

In our work [32], we have proposed a semiautomated approach for identifying service mismatches at the interface level (by building on top of approaches in XML schema matching [35]) and the protocol level. The proposed approach allows to identify mismatches between service interfaces and protocols, and provides suggestions on how to resolve them whenever possible. As mentioned before, automated approaches are limited in the type of mismatches that they can detect. The focus of the work in this paper is to complement automated approaches (e.g., [32]) by: 1) providing a framework to maintain a taxonomy of mismatch patterns that not all of them can be detected by automated approaches and 2) extending the adapter code development from the stand-alone approach to aspect-oriented approach.

In this paper, we have extended the framework of mismatch patterns [5] by proposing an aspect-oriented adapter development approach. This approach complements stand-alone adaptation by offering a greater flexibility for managing the life cycle of business processes. The idea of aspect-oriented approach for adaptation was first introduced in our earlier work [29]. Here, we have extended the aspect-oriented language to represent all common mismatch patterns and have performed a comparative study to identify situations in which each adapter development approach is preferable.

Another recent work [8] proposes an automated approach for protocol-level (i.e., assuming compatible interfaces) stand-alone adapter development. We complement their work in that the adapter developers can use our approach to identify possible mismatches between service interfaces and protocols, and then use the automated approaches such as those in [8], [32] for automatically generating the code for stand-alone adapters. It should be noted that adopting semantic Web services approaches also does not remove the need for adaptation [9]. The mismatch pattern framework presented in this paper can be extended to capture possible differences between semantic-enabled services, as well.

Other related work in this area have also investigated matching of Web service interfaces, e.g., [19], [37]. However, they aim at computing a measure of similarity between service interfaces. In addition, service protocols are not considered in those work and they do not investigate service adaptation. Ponnekanti and Fox [34] present a framework for handling differences among service interfaces. In their approach, it is assumed that distinct service interfaces are derived from a common base using a limited number of modification operations. Their approach is, therefore, limited to handling mismatches at the interface level and in the context of service evolution.

AOP in software components and Web services. Many work (e.g., [15], [18], [36]) have adopted aspects to adapt the component to a changing environment at the configuration level and in the case of component evolution. Cámara et al. [10], a later work compared to our initial work on aspect-based adaptation [29], presents early results on using aspect-oriented programming to design software component adapters. In our work, in addition to presenting a systematic approach to capture service differences in terms of mismatch patterns, we provide advice templates for adaptation logic resolving each mismatch pattern and also an implementation framework for the aspect-oriented adaptation.

The use of AOP in Web services has also been extensively explored. In particular, nonfunctional properties of services find a natural appeal in AOP programming [22]. In [33], Nicoara and Alonso present an aspect-oriented (Java-based) platform that aims to keep services aligned with changes in the environment. AOP has also been used at the process definition level, e.g., in [17], [11]. In [17], aspects are used to adapt services to changing environments. In that approach, aspect weaving is done at compile time by modifying the process tree. The advantage of such an approach is that no specific extensions are needed on the execution engine to handle the aspects since these are embedded in a usual BPEL source. However, that approach requires the engine (or running process instances) to be restarted for reflecting changes. By contrast, our extension to aspect-enable ActiveBPEL engine allows dynamic weaving of aspects at runtime.

In [11], Charfi and Mezini propose to use AOP to modularize nonfunctional concerns, e.g., logging and security of BPEL processes. This work has been extended to address dynamic changes in service composition in [12]. Unlike their work, we focus on the identification of common mismatches between services and enabling resolutions in either stand-alone or aspect-oriented adapters. Cottenier and Elrad [16] extend Axis engine to intercept message and apply AOP to resolve mismatches between messages exchanged between services. Similarly, Wohlstadter and Volder [38] intercept and parse the content of SOAP messages to identify pointcuts in order to apply advices that handle document-oriented concerns, e.g., encryption or schema transformation. Both of these work focus on message-level processing. We present a holistic approach for adaptation to address both the interface- and protocol-level mismatches.

7 Conclusion and Future Work

This paper has tackled a key problem in middleware and specifically in service-oriented architectures, i.e., adapting loosely coupled services so that they can interact. The main contributions of this work consist in 1) proposing a taxonomy of common mismatches at the service interfaces and business protocols, 2) a structured approach to the identification of mismatches between services and their resolutions, by introducing mismatch patterns, and 3) proposing methods and tools for instantiating patterns with two different architectural approaches, stand-alone adapters, and aspect-oriented adaptation.

The combination of mismatch patterns and aspect-oriented adaptation presents the foundation for rapid adaptation of Web services. Future work in this area consists in using the proposed framework to identify possible mismatches at other high-level specifications of services, e.g., service policies, along with the development of tools to support detection of all common mismatches between services, which would provide the remaining missing piece to the support for the adapter development life cycle.

ACKNOWLEDGMENTS

The work has been done while Hamid Reza Motahari-Nezhad was a research fellow at UNSW, Australia.

REFERENCES

[1] P. Ajalin et al., "SAP R/3 Integration to RosettaNet Processes Using Web Service interfaces," Technical Report, SoberIT, T-86.301, 2004.
[2] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services—Concepts, Architectures and Application.* Springer-Verlag, 2004.
[3] S. Becker et al., "Towards an Engineering Approach to Component Adaptation," *Proc. Architecting Systems with Trustworthy Components 2004,* pp. 193-215, 2006.
[4] C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo, "Querying Business Processes with BP-QL," *Proc. Very Large Data Bases (VLDB '05),* 2005.
[5] B. Benatallah, F. Casati, D. Grigori, H. Nezhad, and F. Toumani, "Developing Adapters for Web Services Integration," *Proc. Center for Advancement of Informal Science Education (CAiSE '05),* 2005.
[6] B. Benatallah, F. Casati, and F. Toumani, "Representing, Analysing and Managing Web Service Protocols," *Data and Knowledge Eng. J.,* vol. 58, no. 3, pp. 327-357, 2006.
[7] A. Bracciali, A. Brogi, and C. Canal, "A Formal Approach to Component Adaptation," *J. System and Software,* vol. 74, no. 1, pp. 45-54, 2005.
[8] A. Brogi and R. Popescu, "Automated Generation of BPEL Adapters," *Proc. Int'l Conf. Service-Oriented Computing (ICSOC '06),* pp. 27-39, 2006.
[9] C. Bussler, D. Fensel, and A. Maedche, "A Conceptual Architecture for Semantic Web Enabled Web Services," *SIGMOD Record,* vol. 31, no. 4, pp. 24-29, 2002.
[10] J. Cámara, C. Canal, J. Cubo, and J.M. Murillo, "An Aspect-Oriented Adaptation Framework for Dynamic Component Evolution," *Electronic Notes in Theoretical Computer Science,* vol. 189, pp. 21-34, 2007.
[11] A. Charfi and M. Mezini, "Aspect-Oriented Web Service Composition with AO4BPEL," *Proc. European Conf. Web Services (ECOWS '04),* pp. 168-182, 2004.
[12] A. Charfi and M. Mezini, "AO4BPEL: An Aspect-Oriented Extension to BPEL," *World Wide Web J.,* vol. 10, no. 3, pp. 309-344, 2007.
[13] P. Chen, M. Critchlow, A. Garg, C. van der Westhuizen, and A. van der Hoek, "Differencing and Merging within an Evolving Product Line Architecture," *Proc. Product Families Eng. (PFE '03),* pp. 269-281, 2003.
[14] S. Chidamber and C. Kemerer, "A Metrics Suite for Object-Oriented Design," *IEEE Trans. Software Eng.,* vol. 20, no. 6, pp. 476-493, June 1994.
[15] A. Colyer, A. Clement, R. Bodkin, and J. Hugunin, "Using AspectJ for Component Integration in Middleware," *Proc. Object-Oriented Programming, Systems, Language, and Applications (OOPSLA '03),* 2003.
[16] F. Akkawi, D.P. Fletcher, T. Cottenier, D.P. Duncavage, R.L. Alena, and T. Elrad, "An Executable Choreography Framework for Dynamic Service-Oriented Architectures," *Proc. IEEE Aerospace Conf. (AERO '06),* pp. 13-30, 2006.
[17] C. Courbis and A. Finkelstein, "Towards Aspect Weaving Applications," *Proc. Int'l Conf. Software Eng. (ICSE '05),* pp. 69-77, 2005.
[18] A. Dantas, J.W. Yoder, P. Borba, and R. Johnson, "Using Aspects to Make Adaptive Object-Models Adaptable," *Proc. Reflection, AOP and Meta-Data for Software Evolution (RAM-SE '04),* pp. 9-19, 2004.
[19] X. Dong, A.Y. Halevy, J. Madhavan, E. Nemes, and J. Zhang, "Similarity Search for Web Services," *Proc. Very Large Data Bases (VLDB '04),* pp. 372-383, 2004.
[20] M. Dumas, M. Spork, and K. Wang, "Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation," *Proc. Business Process Management (BPM '06),* pp. 65-80, 2006.
[21] M. Abi-Antoun et al., "Differencing and Merging of Architectural Views," Technical Report, ISRI-05-128R, Carnegie Mellon Univ., 2005.
[22] N. Loughran et al., "Survey of Aspect-Oriented Middleware Research," Technical Report, AOSD-Europe-ULANC-10, Lancaster Univ., June 2005.
[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns.* Addison-Wesley, 1995.
[24] A. Garcia et al., "Modularizing Design Patterns with Aspects: A Quantitative Study," *Proc. Aspect-Oriented Software Development (AOSD '05),* pp. 3-14, 2005.
[25] J. Hannemann and G. Kiczales, "Design Pattern Implementation in Java and AspectJ," *SIGPLAN Notices,* vol. 37, no. 11, pp. 161-173, 2002.
[26] P. Inverardi, L. Mostarda, M. Tivoli, and M. Autili, "Synthesis of Correct and Distributed Adaptors for Component-Based Systems: An Automatic Approach," *Proc. Conf. Automated Software Eng. (ASE '05),* 2005.
[27] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, "An Overview of AspectJ," *Proc. European Conf. Object-Oriented Programming (ECOOP '01),* pp. 327-353, 2001.
[28] W. Kongdenfha, H. Motahari, R. Saint-Paul, B. Benatallah, and F. Casati, "An Aspect-Oriented Approach for Service Adaptation," Technical Report, UNSW-CSE-TR-0920, Univ. of New South Wales, 2009.
[29] W. Kongdenfha, R. Saint-Paul, B. Benatallah, and F. Casati, "An Aspect-Oriented Framework for Service Adaptation," *Proc. Int'l Conf. Service-Oriented Computing (ICSOC '06),* 2006.
[30] X. Li, Y. Fan, and F. Jiang, "A Classification of Service Composition Mismatches to Support Service Mediation," *Proc. Grid and Cooperative Computing (GCC '07),* pp. 315-321, 2007.
[31] H.R.M. Nezhad, B. Benatallah, F. Casati, and F. Toumani, "Web Services Interoperability Specifications," *Computer,* vol. 39, no. 5, pp. 24-32, May 2006.
[32] H.R.M. Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati, "Semi-Automated Adaptation of Service Interactions," *Proc. World Wide Web Conf. (WWW '07),* pp. 993-1002, 2007.
[33] A. Nicoara and G. Alonso, "Dynamic AOP with PROSE," *Proc. Center for Advancement of Informal Science Education (CAiSE '05),* pp. 125-138, 2005.
[34] S. Ponnekanti and A. Fox, "Interoperability among Independently Evolving Web Services," *Proc. Conf. Middleware,* pp. 331-351, 2004.
[35] E. Rahm and P.A. Bernstein, "A Survey of Approaches to Automatic Schema Matching," *Int'l J. Very Large Data Bases,* vol. 10, no. 4, pp. 334-350, 2001.
[36] C.C. Soria, J. Pérez, and J.A. Carsí, "Dynamic Adaptation of Aspect-Oriented Components," *Proc. Component-Based Software Eng. (CBSE '07),* pp. 49-65, 2007.
[37] Y. Wang and E. Stroulia, "Flexible Interface Matching for Web-Service Discovery," *Proc. Web Information Systems Eng. (WISE '03),* 2003.
[38] E. Wohlstadter and K. Volder, "Doxpects: Aspects Supporting XML Transformation Interfaces," *Proc. Aspect-Oriented Software Development (AOSD '06),* pp. 99-108, 2006.
[39] D.M. Yellin and R.E. Strom, "Protocol Specifications and Component Adaptors," *ACM Trans. Programming Languages and Systems,* vol. 19, no. 2, pp. 292-333, 1997.
[40] A.M. Zaremski and J.M. Wing, "Signature Matching: A Tool for Using Software Libraries," *ACM Trans. Software Eng. and Methodology,* vol. 4, no. 2, pp. 146-170, 1995.
[41] A.M. Zaremski and J.M. Wing, "Specification Matching of Software Components," *ACM Trans. Software Eng. and Methodology,* vol. 6, no. 4, pp. 333-369, 1997.

Woralak Kongdenfha received the master's degree in computer science from the Asian Institute of Technology, Bangkok, Thailand. She is a PhD student in computer science at the University of New South Wales, Sydney, Australia. Her research interests include service-oriented computing, Web mashups, and data integration.

Hamid Reza Motahari-Nezhad received the PhD degree in computer science and engineering from the University of New South Wales (UNSW), Sydney, Australia. He is a researcher at Hewlett-Packard Labs, Palo Alto, California. Previously, he was a research fellow at UNSW. His research interests include service-oriented computing, cloud computing, and business process management. He is a member of the IEEE.

Boualem Benatallah is a professor at the University of New South Wales, Sydney, Australia. His research interests include Web service protocols analysis and management, enterprise services integration, process modeling, and service-oriented architectures for pervasive computing. He has published widely in international journals and conferences including *IEEE TKDE*, *IEEE TSE*, *IEEE IC*, *IEEE IS*, *VLDB Journal* and ICDE, ICDS, WWW, and ER conferences.

Fabio Casati is a professor at the University of Trento, Italy. Previously, he was a senior researcher at Hewlett-Packard Labs, Palo Alto, California. His research interests have three main directions: The first is on middleware for integration, the second is about bringing and extending traditional integration technologies to all enterprise data and to the Web, and the third is related to improving how scientists produce, disseminate, evaluate, and consume scientific knowledge.

Régis Saint-Paul received the MSc and PhD degrees in computer science from the University of Nantes, France. He is a researcher at CREATE-NET, Trento, Italy. He spent two years as a research associate at the University of New South Wales, Sydney, Australia. His research interests include service-oriented architectures, database systems, data mining, data summarization, and end-user programming. He is a member of IEEE Computer Society and the ACM.