An Aspect-oriented Approach for Service Adaptation

Woralak Kongdenfha[1] Hamid R. Motahari-Nezhad[1,2] Boualem Benatallah[1]
Fabio Casati[3] Régis Saint-Paul[4]

[1] University of New South Wales, Australia
{woralakk,hamidm,boualem}@cse.unsw.edu.au
[2] Hewlett Packard Labs, Palo Alto, CA, USA
hamid.motahari@hp.com
[3] University of Trento, Italy
casati@dit.unitn.it
[4] CREATE-NET International Research Center, Italy
regis.saint-paul@create-net.org

Technical Report
UNSW-CSE-TR-0902
February 2009

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 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 standalone 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. We explain how it simplifies adapter development through a case study.

1 Introduction

The push toward business process automation, motivated by opportunities in terms of cost savings, higher quality, and more reliable executions has generated the need for integrating the different enterprise applications involved in such processes. Application integration has been one of the main drivers in the software market during the late nineties and into the new millennium. The typical approach to integration and to process automation is based on the use of adapters and of message brokers [55]. 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 [15]. The main benefit they bring is that of standardization, in terms of data format (XML), interface definition language (WSDL), transport mechanism (SOAP) and many other interoperability aspects [15, 47]. 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* [19]. A business protocol specifies message exchange sequences that are supported by the service, for example expressed in terms of constraints on the order in which service operations should be invoked.

While standardization simplifies interoperability, it does not remove the need for adapters [47]. 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 the *languages* (e.g., WSDL and BPEL) for their definition, not the specific interfaces or protocols [15, 47]. 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. This is in fact what happens in practice [50], and implies the need for adaptation at both interface and business protocol levels.

The need for adaptation is intrinsic in the philosophy of Web services. Services are meant to be loosely-coupled and to target (mainly) B2B interactions. This implies that services should not be designed for interoperability with a particular client in mind. They are designed to be open and possibly without knowledge, at development time, about the type of clients that will access them.

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 *standalone 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 extends our previous work [18, 44] 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 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 standalone 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 semi-automatic 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 standalone adapters.

With respect to our previous work [18, 44], this paper makes the following extensions and contributions: (i) we discus standalone versus aspect-oriented approach for adaptation and provide guidelines to help developers to decide on situations in which each of the approaches is preferable, (ii) we provide a more comprehensive description of aspect-oriented service adaptation, (iii) we characterize mismatch patterns and adaptation templates using aspect-oriented approach for adaptation, (iv) we present the usage and the implementation of this approach, which is only sketched in previous work with an earlier version of the tool [44].

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 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 Adapter Development for Web Services

In this section, we define the adaptation requirements in Web services. Then, we propose a methodology for adapter development by providing a classification of common mismatches between service interfaces and business protocols, and introducing mismatch patterns.

2.1 Adaptation for Compatibility and Replaceability

We classify the need for adaptation in Web services into two basic categories: adaptation for compatibility and adaptation for replaceability. The first category refers to wrapping a Web service S so that it can interact with another service C. For example, consider a service S allowing companies to order office supplies. If the provider of this service wants to be able to do business with a certain retailer C (say, *Wal-Mart* or *Target*), then it needs to adapt its service S so that it can interoperate with these retailers.

Adaptation for replaceability refers to modifying a Web service so that it becomes *compliant* with (i.e., can be used to replace) another service. This is important especially in those business environments where the interaction, even at the interface and business protocol level, has been standardized either de jure or de facto (e.g. due to the presence of a dominant player in the market). For example, the RosettaNet[12] consortium standardizes the external behavior of services in the IT supply chain space. In these cases, service providers may have to adapt their services so that they can follow the guidelines prescribed by the standards. Adaptation for replaceability is also needed when a new version of a service is developed, possibly with a different external behavior, but we want to preserve backward compatibility (that is, an adapter should be provided so that the service is also offered in a version that behaves like the old one).

Replaceability may be *partial* or *total* [19]. Total replaceability occurs when a service S_r behaves externally like another service S. This means that any service that interacts *correctly* (i.e., without generating runtime faults) with S will also be able to interact correctly with S_r (note that the opposite is not necessarily true). Partial replaceability occurs when a service S_r can behave like S only in certain interactions (i.e., S_r behaves like S in some but not all conversations). A particular and important case of partial replaceability is that of replaceability with respect to the interaction with another given service S'. Hence, although in general S_r may not totally support the same conversations as S, it can still support the same conversations as S when interacting with S'. We refer the reader to [19] for a detailed definition of compatibility and replaceability among services.

This paper proposes a technique for developing adapters to achieve total replaceability. Note that replaceability can be expressed in terms of adaptability, and therefore the solution developed for adaptation for replaceability can be adopted for the latter scenario. The related issues of partial replaceability and replaceability with respect to an interaction partner can be handled in an analogous manner. In the following, we introduce mismatch patterns for the characterization and resolution of mismatches at the interface and protocol levels, and for the semi-automatic adapter development.

2.2 Mismatch Patterns for service mismatch characterization and resolution

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 semi-automated 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 clients that S can interact with.

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*[7] and *Arcweb*[8], *Google Checkout*[6] , *XWebCheckout* [13], *Amazon*

Web Service[4], *Amazon Ecommerce Service*[2] , *PayPal Web Service*[11], *Payment-Express Web Service*[10], *Amazon Flexible Payments Service*[3] and *Moon Purchase Order Management Service*[9]. In the following, we characterize these common mismatches.

Interface-level Mismatches. To characterize these mismatches, we use, as a concrete example, *Mappoint* and *Arcweb* route Web services, which offer similar functionalities for finding driving routes between two points through different WSDL interfaces (operations CalculateRoute and findRoute, respectively).

- *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). For instance, suppose that element Preference (a sub-element of the parameter Specification of operation CalculateRoute) accepts “quickest”, “shortest” and “least toll” as possible values, while element RouteType (an element of parameter routeFinderOptions of operation findRoute) accepts “quickest” and “shortest” as possible values. In this case, there is no possible value of RouteType that corresponds to the value “least toll” of Preference.

Protocol-level Mismatches. We classify common mismatches at the protocol-level using a supply chain example. 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 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 this level:

- *Ordering Mismatch*: This type of mismatches occurs when protocols P and P_r support the same messages but in different orders as in the example above.

- *Extra Message Mismatch*: This mismatch occurs when protocol P_r sends a message that protocol P does not send. In the supply chain scenario, 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, where protocol P issues an acknowledgment when receiving a request for invoice, while protocol P_r does not produce it.

- *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.

We have identified common recurring mismatches at the interface and protocol levels. In the following, we propose the concept of *mismatch patterns* to formalize these differences and also to provide solutions to resolve such recurring problems, identified in Section 2.2.

Mismatch Patterns

Mismatch pattern is a similar notion to that of *design pattern* in software engineering [36]. 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 2.1 summarizes the structure of a mismatch pattern.

Table 2.1: 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

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 required parameters. This information is used to (manually or automatically) generate the adaptation logic from the template. The developer may then want or need to further customize the resulting logic skeleton to add some custom business logic, or can just directly use the generated adaptation skeleton to deploy the adapter.

The exact specification of adaptation aspect depends on the adapter development approach. In Section 3, we discuss different approaches for adapter development, and present formalisms for the specification of adapter templates. We use the proposed mismatch pattern framework and the adapter template specification to represent the common mismatches identified in Section 2.2 as a set of built-in patterns (Section 4). The developers can also add to the built-in patterns if there are specific mismatches that they would like to handle differently or if there are mismatches that are not captured in the built-in set. Patterns can be shared between adapter developers and evolved, especially with the new trend of user-centric sharing of content fostered by Web 2.0 [46]. By provision of 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 Adapter Development Approaches

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. Consider when the business logic needs to be evolved, e.g., for business reason or to interact with another partner. The developer needs to separate and clean the adaptation logic, that has been added previously, before modifying the business logic. 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 (i.e., each with a separate BPEL process). In this case, when a change at the business logic is required, it needs to be replicated to all the versions, this make evolution expensive and error-prone.

From our perspective, it is important to separate the adaptation logic from the business logic. Such separation helps to avoid the need of developing and maintaining several versions of a service implementation and isolates the adaptation logic in a single place. 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: standalone and aspect-oriented adapters.

3.1 Standalone Service Adaptation

In this approach, an adapter A (referred to as a *standalone* adapter) is placed in between a service S implementing protocol P and another service S_r implementing protocol P_r in order to support the interactions between them, as illustrated in Figure 3.1. For service S_r (resp. service S), adapter A looks like a service whose protocol is compatible with P_r (resp. P). In this approach, all messages sent from service S pass through the adapter A, which performs the adaptation logic and invokes operations of protocol P_r from S_r, and vice versa.

To illustrate how the adaptation logic can be modeled and implemented using standalone adapter approach, we present the *Ordering Constraint Pattern* (OCP) that is used to handle the ordering mismatch introduced in Section 2.2. This pattern is associated with an adapter template, shown in Table 3.1, that consists of a set of actions to

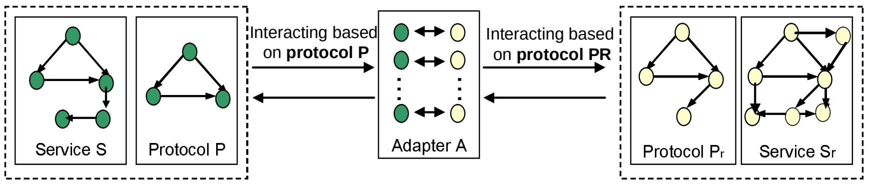

Service S
Protocol P
Interacting based on protocol P
Adapter A
Interacting based on protocol PR
Protocol Pr
Service Sr

Figure 3.1: A standalone adapter A for enabling service interoperability

resolve the ordering mismatch.

Table 3.1: Ordering Constraint mismatch Pattern (OCP)

Template Parameters	**Protocols P (of service S) and P_r (of service S_r), message $msgO^P$ to be re-ordered**
Adapter Template	- *Perform* activities as prescribed by P for parts that do not need adaptation (BPEL *receive, invoke, reply* activities); - `Receive` $msgO^p$; - `Assign` $msgO^a \longleftarrow msgO^p$; - `Assign` $msgO^{sr} \longleftarrow msgO^a$, when $msgO^{sr}$ expected; - `Invoke` O^{sr} with $msgO^{sr}$

OCP takes as its parameters the protocol specifications of two services that have a mismatch, and a message $msgO^p$ to be re-ordered. Once instantiated, OCP generates an adapter that resolves an ordering mismatch by receiving message $msgO^p$ of protocol P and storing it for later use. When message $msgO^p$ is expected, the adapter creates a message $msgO^{sr}$ required by service S_r from the value of message $msgO^p$. Then it invokes service S_r with message $msgO^{sr}$. Figure 3.2 shows the usage of OCP to resolve the ordering constraints of message sendShippingPreferencesIn. From the input parameters of the template, it is possible to determine the message ordering constraints of the two services. In this case, the adapter can temporarily store the parameter of operation sendShippingPreferences of the client protocol P and forward it to the operation of service S_r according to protocol P_r.

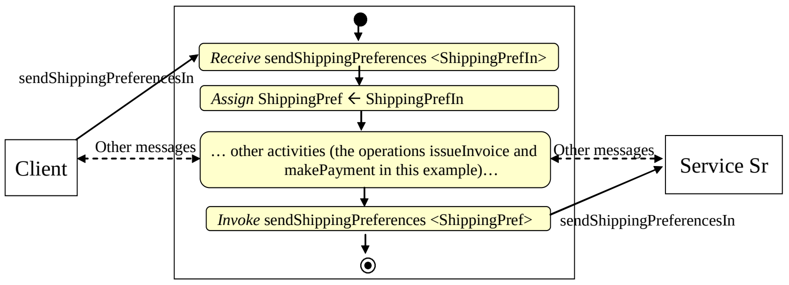

sendShippingPreferencesIn
Client
Other messages
Receive sendShippingPreferences <ShippingPrefIn>
Assign ShippingPref ← ShippingPrefIn
… other activities (the operations issueInvoice and makePayment in this example)…
Invoke sendShippingPreferences <ShippingPref>
Other messages
Service Sr
sendShippingPreferencesIn

Figure 3.2: Sample usage of the adapter template specified in the OCP

3.2 Aspect Oriented Service Adaptation

Aspect-Oriented Programming (AOP) is a technique that allows the separation of concerns in software development, making it possible to modularize cross-cutting concerns of a system [43, 35]. We consider the adaptation logic as a cross-cutting concern, which means that from the developer point of view it is transversal to the other functional concerns of the service. We therefore propose an aspect-oriented approach for Web service adaptation. In our framework, 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 Figure 3.3, 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.

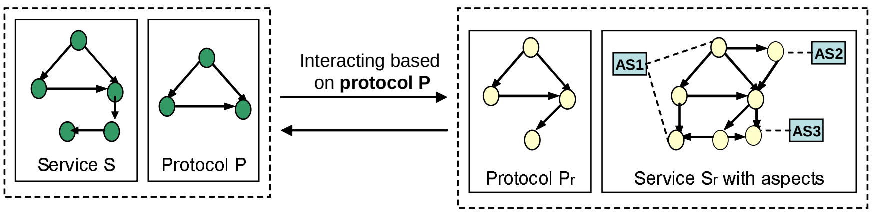


Figure 3.3: Adaptation aspects $AS1$,$AS2$, and $AS3$ for enabling service interoperability

Our framework enables both approaches for adapter code generation based on adapter templates: standalone and aspect-oriented adapters. However, in this paper, we focus on the use of the AOP approach for adapter development. The details of developing standalone adapters can be found in [18]. We defer the comparison between the two approaches for adapter development to Section 5.2.

To describe our approach for aspect template specification and adaptation aspects generation, we need to understand what is expected from an adapter. As mentioned above, the role of an adapter consists in mapping interactions with protocol P_r into interactions with protocol P, and vice versa. This requires performing 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. Therefore, we choose BPEL for defining the adapter templates. Hereafter, we detail the structure of aspect template, 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 3.2. 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 Section 3.2.

Query

A query expresses a process execution point, also known as *joinpoint* in the context of AOP [43, 35], 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 [35]. A first approach consists in the expression of joinpoints only on the service code constructs. A second approach consists in directly expressing joinpoints not only service code but also runtime execution context.

Table 3.2: 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` $msgO_j^{sr}$
	}

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 (i) operations (with or without a certain signature) to enable the resolution of interface-level mismatches, and (ii) 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 operate on BPEL code such as BPQL [17] 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 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. Figure 3.4 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: (i) our language can express conditions on service interaction paths, and (ii) our language includes keywords for specifying the relative location of the joinpoint to the BPEL activity that matched the specified conditions (i.e. the *before*, *after* or *around* keywords). As explained above, these concepts are needed to achieve a self contained query language able to express all the conditions necessary for identifying joinpoints in the service adaptation context.

<*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*

Figure 3.4: Semi-formal syntax for query language

As shown in Figure 3.4, 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 3.2 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 (i.e., unregistered) is taken, hence the query parameters of the OCPStore in this example include <operation>=sendShippingPreferences and <executionPath>=unregistered. These parameters will be evaluated, at runtime, against currently executing operation (O_i^{sr}) and execution path (S_i) of the adapting service.

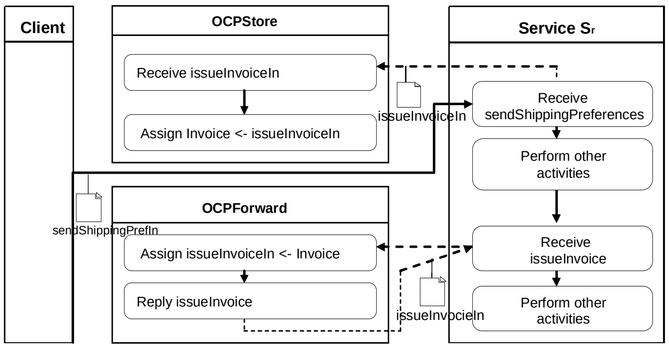


Figure 3.5: Sample usage of the aspect template specified in the OCP

Figure 3.5 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 in the advices some additional logic. This logic is not 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 to 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 2.2. We focus on the aspect-oriented approach for adapter development and use the language proposed in Section 3 for adapter template specification. However, the same set of patterns can be used to develop standalone adapters, as presented in [18].

4.1 Interface-level Patterns

Signature Mismatch Pattern (SMP)

This pattern is used to handle a signature mismatch. It consists of two parts, i.e., SMPInput and SMPOutput, as shown in Table 4.1. 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.

Table 4.1: 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` $msgO^p$; }

To instantiate the aspect template of SMP, the developer provides *inputType* (resp. *outputType*) as query parameter and XQuery/XSLT transformation functions as advice parameters to SMPInput (resp. 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.

Figure 4.1 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

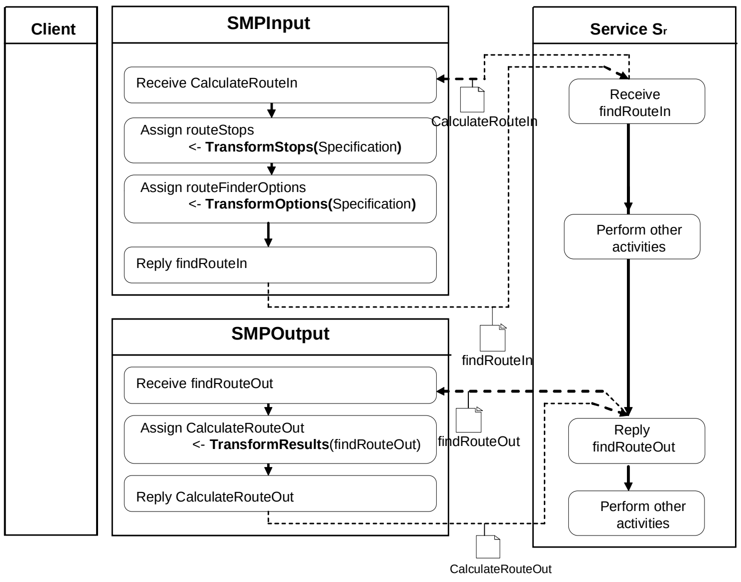


Figure 4.1: Sample usage of SMP

by protocol P_r such that service S_r can continues. Similar actions are specified in the SMPOutput to resolve mismatches on the outgoing messages of service S_r.

Parameter Constraint Pattern (PCP)

This pattern is used to handle a parameter constraint mismatch. As shown in Table 4.2, PCP checks if message $msgO^p$ of a client with protocol P verifies the constraint of operation O_r of service S_r. The constraint-checking condition is expressed by an XQuery function.

Table 4.2: Parameter Constraint Pattern (PCP)

Query	**Generic Adaptation Advice**

`query` (⟨inPara⟩,⟨operation⟩)
`executes` *around receive*
`when` P^{sr} = ⟨inPara⟩
    `AND` O^{sr} = ⟨operation⟩
	PCP(⟨T⟩) {
	  `Receive` $msgO^p$;
	  `Switch`(⟨T⟩) {
	    `Assign` $msgO^{sr} \leftarrow msgO^p$;
	    `Reply` $msgO^{sr}$; }
	  `Otherwise` {throw faultMsg;} }

Figure 4.2 shows a sample usage of PCP to resolve a parameter constraint mismatch of the two route Web services described in Section 2. In this example, the condition Verify_Specification_Constraints checks if parameter Specification of operation CalculateRoute verifies the constraints of operation findRoute. In this case, if the value of element Preference (a sub-element of parameter Specification) is in {"quickest", "shortest"}, PCP will update the value of message findRouteIn, otherwise it will raise a constraint violation exception.

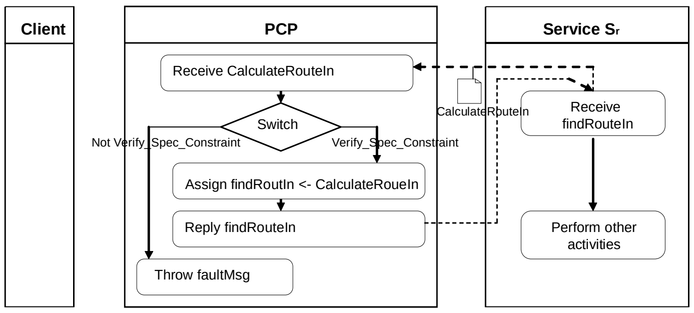


Figure 4.2: Sample usage of PCP

4.2 Protocol-level Patterns

In addition to the OCP, which was used to illustrate aspect-oriented approach in Section 3, in this section we discuss other protocol-level mismatch patterns.

Extra Message Pattern (EMP)

This pattern is used to handle the extra message mismatch. Its aspect template is shown in Table 4.3. It replaces the *reply* activity of an operation O_i^{sr} with an *empty* activity. The outgoing message of service S_r is therefore discarded.

Table 4.3: Extra Message Pattern (EMP)

Query	**Generic Adaptation Advice**

`query`(⟨message⟩,⟨operation⟩)
`executes` around reply
`when` O_i^{sr} = ⟨operation⟩
  `AND` $outputVariable^{sr}$ = ⟨message⟩
	EMP() {
	  `Empty`;
	}

It should be noted that the use of EMP makes sense only if the extra message is not an important message, i.e., containing *important* information from the S_r point of view (e.g. it contains legal disclaimer), as discarding important message would raise the problem of information loss [42]. Figure 4.3 shows a sample usage of EMP to resolve an extra message mismatch in the supply chain example described in Section 2. The *reply* activity of operation InvoiceAcknowledgement is replaced by an *empty* activity.

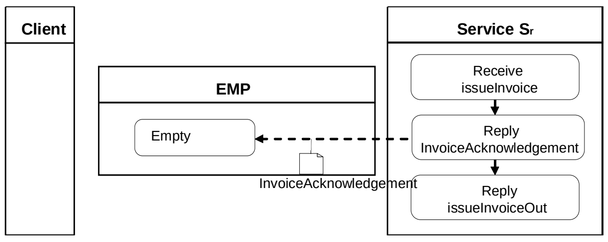


Figure 4.3: Sample usage of EMP

Missing Message Pattern (MMP)

This pattern is used to solve the missing message mismatch. As shown in Table 4.4, 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.

Table 4.4: 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` $msgO^p \leftarrow$ ⟨T⟩(⟨Var^{sr}⟩); `Reply` $msgO^p$; }

Figure 4.4 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 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 additional contextual information, specified by the user as advices' parameters) to the advices. For example, the variable POVar may not be part of message currently exchanged. Rather it is either a part of a message previously sent or received by service S_r, or an internal variable of the service S_r. In the latter case, both standalone adapter and aspect-oriented approaches are possible to generate the message InvoiceAcknowledgement since the standalone adapters can maintain messages exchanged between two processes, and the adaptation aspects are able to access the internal execution data of service S_r. 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 aspect-oriented approach.

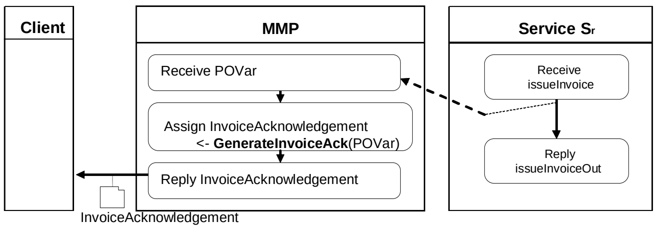


Figure 4.4: Sample usage of MMP

It should be emphasized that the possibility of generating the message InvoiceAcknowledgement depends on whether the purchase order number is an internal variable of the service S_r or it has been sent to the client before. In the latter case, both standalone adapter and aspect-oriented approaches are possible to generate the message InvoiceAcknowledgement since the standalone adapters can maintain messages exchanged between two processes, and the adaptation aspects are able to access the internal execution data of service S_r (as will be explained in Section 5). 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 aspect-oriented approach.

One to Many Pattern (OMP)

This pattern is used to resolve the one-to-many mismatch. As shown in Table 4.5, OMP receives a single message and *splits* it into a set of messages.

Table 4.5: 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` $MSGO_j^{sr}$;
	}

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 store them for later use. 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 also require information from $msgO^p$.

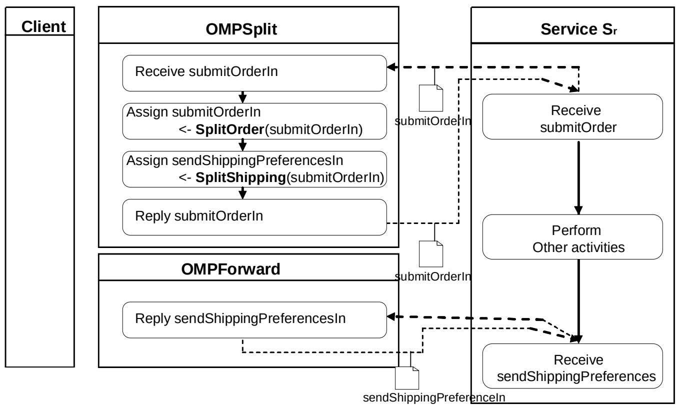

Client
OMPSplit
Service Sr
Receive submitOrderIn
Assign submitOrderIn <- SplitOrder(submitOrderIn)
Assign sendShippingPreferencesIn <- SplitShipping(submitOrderIn)
Reply submitOrderIn
submitOrderIn
Receive submitOrder
Perform Other activities
OMPForward
submitOrderIn
Reply sendShippingPreferencesIn
Receive sendShippingPreferences
sendShippingPreferenceIn

Figure 4.5: Sample usage of OMP

Figure 4.5 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.

Many to One mismatch Pattern (MOP)

This pattern resolve the many-to-one mismatch. As shown in Table 4.6, MOP receives a set of messages and *merges* them into a single message. MOP captures this type of mismatch and its resolution in a set of MOPStore and a MOPMerge. Each of the MOPStore intercepts message $msgO^p$ sent by a client of the protocol P. Since such a message is not desired at this state of service S_r, MOPStore assigns it to a temporary variable for later use. The number of MOPStore depends on the number of messages to be merged into message $msgO_j^{sr}$ of service S_r. Note that the query and advice of MOPStore will be generated from the template. The developer does not need to concern about defining the temporary variables. He only needs to specify which messages are required for the creation of the new message $msgO_j^{sr}$, which is done when the developer identifying mismatches between the two services, and write a transformation function to be used by MOPMerge to create the message $msgO_j^{sr}$ from messages that have been stored previously.

Table 4.6: Many to One mismatch Pattern (MOP)

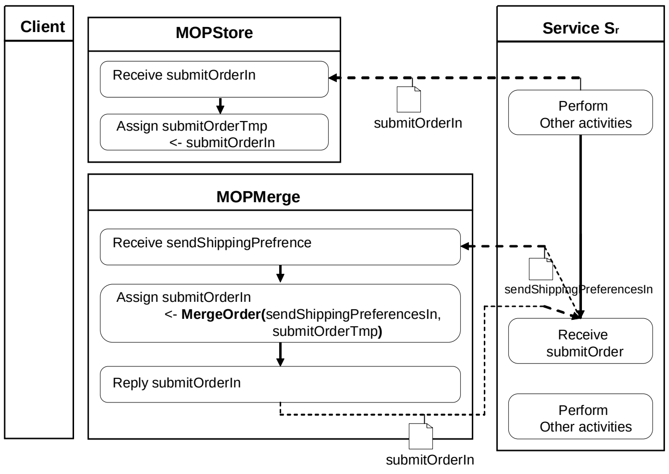
Query	**Generic Adaptation Advice**

`query`(⟨operation⟩)
`executes` *before receive*
`when` O_i^{sr}=⟨operation⟩
`AND` S_i = ⟨executionPath⟩
	MOPStore() {
	`Receive` $msgO^p$;
	`Assign` $msgO_i^{tmp} \leftarrow msgO^p$; }

`query`(⟨operation⟩) `executes` *before receive* `when` O_j^{sr}=⟨operation⟩ `AND` S_j = ⟨executionPath⟩	MOPMerge(⟨T⟩, ⟨$MSGO_i^{tmp}$⟩) { `Receive` $msgO^p$; `Assign` $msgO_j^{sr}$ $\leftarrow$ ⟨T⟩($msgO^p$, ⟨$MSGO_i^{tmp}$⟩); `Reply` $msgO_j^{sr}$; }

Figure 4.6 shows a sample usage of MOP. In this example, MOPStore intercepts message submitOrderIn and stores it for later use. When message submitOrderIn is required by service S_r, the MOPMerge generates the value of message submitOrderIn, required by S_r, by merging the data of temporary variable submitOrderTmp with the incoming message sendShippingPreferencesIn of the current state, using an XQuery function MergeOrder.

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 to develop and deploy the adaptation logic as discussed in the following section.



Client
MOPStore
Receive submitOrderIn
Assign submitOrderTmp <- submitOrderIn
submitOrderIn
Service Sr
Perform Other activities
MOPMerge
Receive sendShippingPrefrence
sendShippingPreferencesIn
Assign submitOrderIn <- MergeOrder(sendShippingPreferencesIn, submitOrderTmp)
Receive submitOrder
Reply submitOrderIn
Perform Other activities
submitOrderIn

Figure 4.6: Sample usage of MOP

5 Implementation and Evaluation

In this section, we discuss our prototype implementation and the evaluation of our approach. The prototyped implementation and a use case scenario are available at `http://www.cse.unsw.edu.au/~soc/projects/aspect-adaptation`.

5.1 Implementation

The approach for adapter development proposed in this paper has been implemented in a prototype tool that consists of two components as depicted in Figure 5.1: (i) pattern-based mismatch identification, and (ii) 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. The tool has been implemented in Java 1.5 using Eclipse 3.1. It adopts the WSDL editor of Eclipse's WTP project for handle Web service interfaces. Web service protocols are handled by a protocol editor which has been implemented as part of the prototype. The adaptation code generation component allows managing mismatch patterns and generating both standalone and aspect-oriented adaptation logic. The standalone adapter approach has been presented in our previous work [18, 48]. 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 as XQuery templates. To instantiate adaptation advices, users need to provide parameters to these templates, i.e., XQuery functions. Once instantiated, the aspect 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, i.e., joinpoints, where advices need to be executed. These 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 Figure

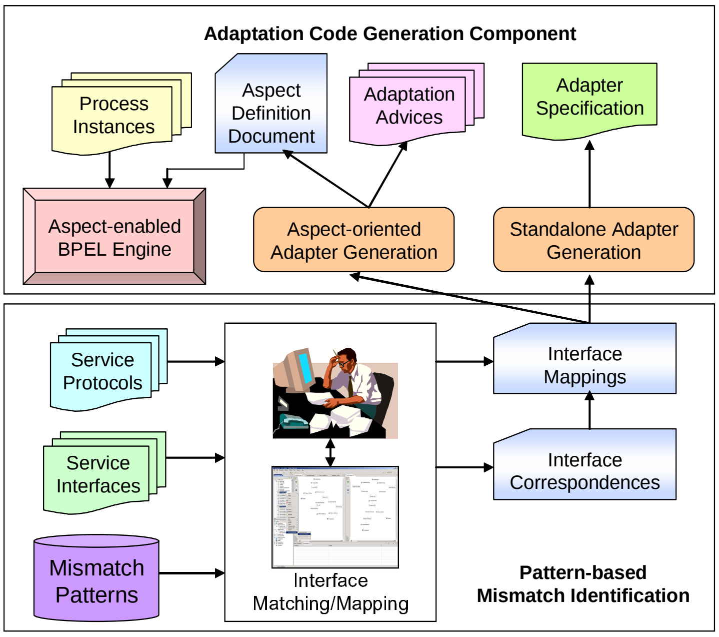

Adaptation Code Generation Component
Process Instances
Aspect Definition Document
Adaptation Advices
Adapter Specification
Aspect-enabled BPEL Engine
Aspect-oriented Adapter Generation
Standalone Adapter Generation
Service Protocols
Service Interfaces
Mismatch Patterns
Interface Matching/Mapping
Interface Mappings
Interface Correspondences
Pattern-based Mismatch Identification

Figure 5.1: The architecture of mismatch patterns based adapter development

5.2. 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 operations are matched by our query. The advice RouteReqSMPInput can be reused to solve mismatches at both joinpoints.

```
<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>
```

Figure 5.2: An Aspect Definition Document (ADD)

When several mismatches need to be addressed at the same joinpoint, the ordering of advice 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 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.

We have developed an aspect-based BPEL engine which interprets at runtime the ADD document and ensures that adaptation advices are invoked as required. Specifically, our prototype extends ActiveBPEL[1] 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[5] and itself woven with the ActiveBPEL code. This enables the aspect manager to access execution data of the business processes (step (i) in Figure 5.3). We particularly 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 context of messages sent and received by the activities. Specifically, the aspect manager collects *type* of messages, i.e., qualified names as specified in WSDL file, as well as *parameter* names and values of the messages.

The aspect manager stores the execution information in an internal data structure and uses this information to check if there is any joinpoint defined on the activity currently 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 Figure 5.3). When a match is found, the aspect manager loads the corresponding advice as specified in the ADD document (step (iii) in Figure 5.3). Consider the ADD document in Figure 5.2. When an activity is executed, the aspect manager looks at the collected contextual information and 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.

Before the BPEL engine can execute advices, the aspect manager needs to pass contextual information about the process to the advice activities. This is implemented as follows: Adaptation advices are deployed as services that have a *receive* activity, a set of activities for adaptation logic, and an optional *reply* activity. The *receive* activity has an input variable AdviceVar. The aspect manager maps content of incoming message of the joinpoint activity into this input variable AdviceVar. Consider the route web service example, when a *receive* activity that receives a message of type CalculateRouteType is matched, the aspect manager maps content of this message to input variable AdviceVar. After this input variable has been mapped, the aspect manager attaches advice activities to process execution queue. Once the advice activities have been executed, the aspect

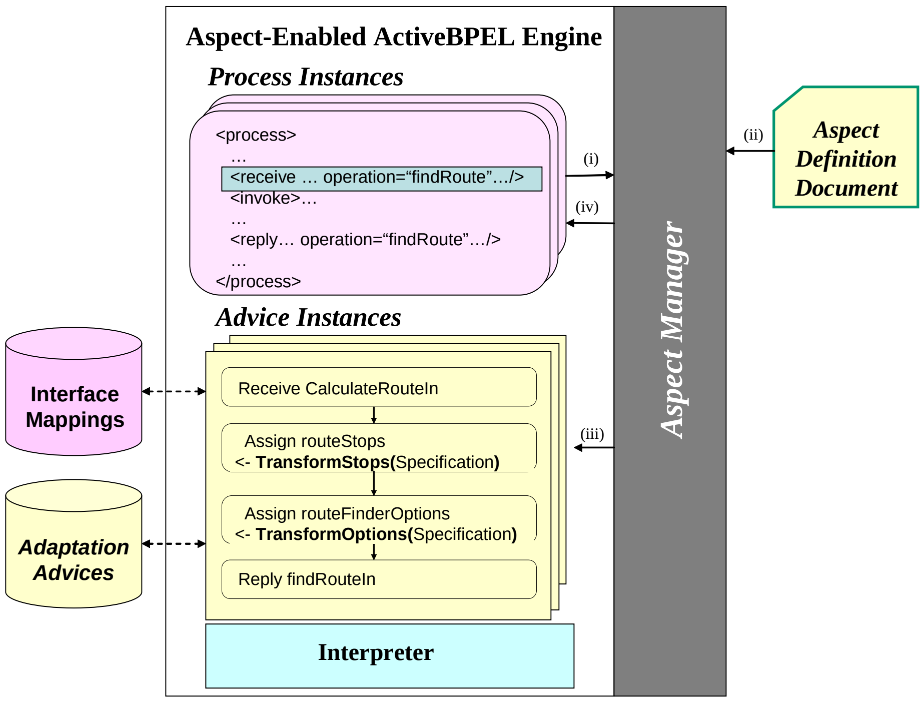

Aspect-Enabled ActiveBPEL Engine
Process Instances
<process>
…
<receive … operation="findRoute"…/>
<invoke>…
…
<reply… operation="findRoute"…/>
…
</process>
(i)
(iv)
(ii)
Aspect Definition Document
Aspect Manager
Advice Instances
Receive CalculateRouteIn
Assign routeStops <- TransformStops(Specification)
Assign routeFinderOptions <- TransformOptions(Specification)
Reply findRouteIn
(iii)
Interface Mappings
Adaptation Advices
Interpreter

Figure 5.3: The deployment of adaptation aspects at runtime

manager maps the *reply* variable of the advice into variable of the joinpoint activity. Note that by this way of implementation, the advice logic is defined based on the input variable AdviceVar of its *receive* activity rather than being specific to the variable name of a joinpoint. Therefore, we can reuse the same advice to resolve mismatches at different joinpoint activities. For example, incoming messages of both findRoute and Distance operations mentioned before, can be mapped into variable AdviceVar and then the same advice logic can be reused. However, the aspect manager only perform direct mapping between messages received by joinpoint activities and the variable AdviceVar. Therefore, the incoming messages of operations findRoute and Distance need to have the same type, i.e., parameter names and their data types as declared in WSDL, as that of variable AdviceVar. This is also required by the fact that XQuery transformation expressions in adaptation advices are defined based on specific message types.

5.2 Evaluation

To provide an evaluation of the proposed approach, we have used the prototyped implementation to develop a real-world interaction scenario. We then provide both qualitative and quantitative evaluations of the proposed approach. The qualitative evaluation is based on a comparative study between standalone and aspect-oriented adapter development approaches, while the quantitative evaluation is based on the adoption of the CK metrics [27] to compare the aspect-oriented and code modification approaches.

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 [14]). 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, Figure 5.4(a) 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.

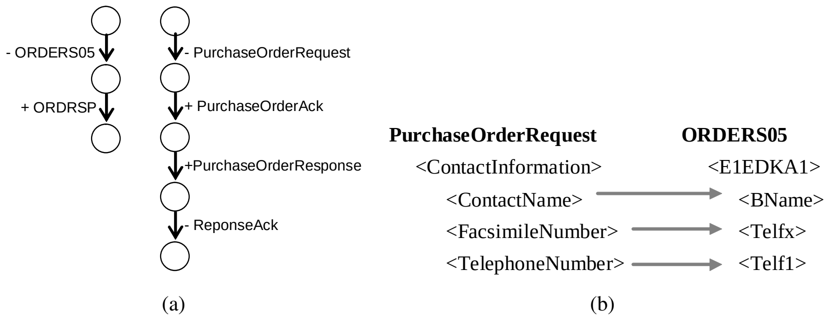


Figure 5.4: Service descriptions of SAP R/3 and RosettaNet PIP 34A: (a) business protocols and (b) message details.

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

Step1: 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 user in identifying actual mismatches. In the case study, the developer consults our pattern taxonomy and find that messages PurchaseOrderRequest and ORDERS05 are different in their element names (as shown in Figure 5.4(b)). The developer also finds that there is another signature mismatch between message PurchaseOrderResponse and ORDRSP. Finally, an extra message mismatch (PurchaseOrderAck) and a missing message mismatch (ResponseAck) are identified.

Step2: 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 limitations we cannot discuss all instantiation scenarios. Instead we discuss in details the instantiation of POReqSMPInput. From the address provided earlier, the reader can find a set of adaptation templates and their instances that we have developed as a scenario for testing our aspect-oriented adaptation approach. For now, let us consider the XQuery template for SMPInput as shown in Figure 5.5. To instantiate this template, the developer needs to provide a transformation function TransformPO to the variable *transform* of the SMPInput template. This 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. However, before this process can be deployed, the developer needs to create a WSDL file for it. This involves resolving the type declarations of messages SigRequest and SigResponse. In particular, the SigRequest needs to have the same *type* as message ORDERS05 of SAP, while SigResponse requires the same *type* declaration as message PurchaseOrderRequest of RosettaNet.

```
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>
```

Figure 5.5: Aspect Template for SMPInput

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 Figure 5.2.

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.

Qualitative Evaluation

Figure 5.6 presents a schematic comparison of the standalone and aspect-oriented approaches for adapter development, in cases where a service S_r with protocol P_r has to be adapted to n client services, with heterogeneous protocols P_1,...,P_n. In the standalone 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.

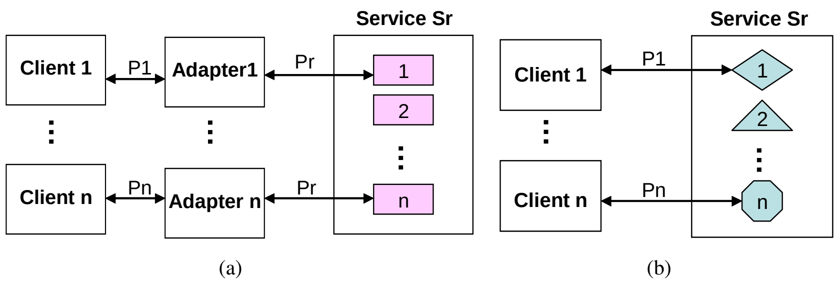


Figure 5.6: Schematic comparison of adaptation in standalone and aspect-oriented approaches: (a) service instances 1,...,n are the same, (b) service instances 1,...,n are modified with respective adaptation aspects

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 standalone 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 standalone 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 analysing 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 operations receives messages of a specific type can reuse the same aspect for message transformations (see Section 5.1). There is no need to generate individual adaptation logic for each single message as is the case of the standalone 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 standalone adapter in which gluecode appear in several places in the service code). The study in [37, 38] also shows that implementing adapters using AOP can better separate the adaptation concern from the functionality of the base programs.

Standalone Service Adaptation: A standalone adapter is implemented as a complete single business process comprising of a set of adaptation activities. The inter-dependencies between these activities are well-defined (comparing to aspect-oriented approach), and thus simplify the *understandability*.

Tradeoffs: 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 standalone adapter even if no adaptation is needed. However, aspect-oriented approach introduces overhead for every single message to check if an adaptation is required (see Step 4 in Section 5). Hence, when the number of mismatches is large relative to the total number of messages, the standalone 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 standalone 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 standalone adapters that need to be suspended and updated).

Table 5.1: Comparison of adapter code generation approaches

	Adaptation Aspects	Standalone Adapters
Possibility of mismatch resolution	$+$	$-$
Recovery	$+$	$-$
Reusability	$+$	$-$
Separation of concern	$+$	$-$
Understandability	$-$	$+$
Overhead	$+/-$	$+/-$
Maintainability	$+/-$	$+/-$

Table 5.1 provides a high-level comparison of the aspect-oriented and standalone adapter development approaches. It can be used as a guideline for an adapter developer to decide about which adapter development approach to take. It shows that 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 standalone adapters. In other case, the intended adaptation scenarios need to be taken into consideration. In cases, when the relative number of mismatches is large, the standalone 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 approach for service adaptation development.

Quantitative Evaluation

To provide a quantitative evaluation of our approach, we have created a BPEL process for a supply chain service with sixty activities. We created adaptation logic using two different approaches, i.e., code modification and aspect-oriented, to resolve mismatches occur in four different interaction scenarios between this process and its partners.

- *scenario1*: there is only one mismatch, i.e., signature mismatch, in the interaction;
- *scenario2*: this interaction consists of seven mismatches, i.e., signature, parameter constraint, ordering, extra message, missing message, split and merge mismatches;
- *scenario3*: this scenario is a case when the number of mismatches is half of the number of activities in BPEL process, i.e., thirty mismatches. In this scenario, mismatches that are relatively more complex than others occur more often, i.e., there are six mismatches of each type merge, split and parameter constraint, and three mismatches of each type signature, ordering, missing message and extra message;
- *scenario4*: this is a case when the number of mismatches is relatively large compared to the number of activities in BPEL process, i.e., sixty mismatches. This scenario is an opposite case of the scenario3 in which mismatches with less complexity occur more often than mismatches with high complexity, i.e., twelve mismatches of each type signature, ordering, missing message and extra message, and four mismatches for each of the rest.

To evaluate the impact of the code modification and aspect-oriented adapter development approaches, we made two assumptions. First, the same adaptation logic are 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 then used the CK metrics [27] to assess the applicability of our approach. The software metrics that have been calculated can be divided into four main categories: size, complexity, coupling, and cohesion metrics. Coupling metrics are the most obvious choice since the use of abstractions in AOP primarily aims to reduce the dependencies between base program and the cross-cutting concerns. AOP also aims to separate modules based on concerns, as such it is reasonable to expect that cohesion increases when AOP is used. Moreover, the introduction of AOP certainly involves the introduction of new abstractions (aspects, advices). Consequently, size metrics are expected to indicate an increase in number of modules. Finally, the use of patterns and aspects is expected to eliminate complexity of code (such as large piece of tangling code). We adapted the calculation of each metric 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 is 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.

Figure 5.7 presents the collected absolute values for all the metrics considering both code modification and aspect-oriented approaches. From these results, we calculate the percentage relative to the absolute values for scenario2 and show them in Figure 5.8.

Metrics		LOC	NOC	CC	WMC	CBO
Scenario1	Code	425	1	3.666667	64.66667	1
	Ao	415	2	3	64	0
Scenario2	Code	775	1	9.666667	99.66667	7
	Ao	705	8	5	95	0
Scenario3	Code	2307	1	35	248	30
	Ao	2007	31	15	228	0
Scenario4	Code	3272	1	51	352	60
	Ao	2672	61	11	312	0

Figure 5.7: Collected values for the CK metrics

Figure 5.8(b) summarizes the results of our evaluation in scenario2. 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 in 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%. This is because, in the code-modification approach, some additional logic needs to be included to select a behavior based on runtime context. In term of the size, the use of aspects has increased the NOC by 87.5% 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 logic 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%. This is because, in the aspect-oriented approach, adaptation logic are captured in separate modules.

In order to evaluate the impact of the number of mismatches on the adapter development approaches, we present the results of evaluation of scenario1-4 in Figure 5.8(a)-(d) respectively. The results show, in all scenarios, the aspect-oriented approach is favorable with respect to most of the metrics used. In particular, with the use of aspects, the COB metric is constantly improved by 100% in all scenarios. When the number of mismatches is small, i.e., only one mismatch in scenario1, the use of aspects slightly improves the LOC, CC and WMC metrics. These metrics are gradually

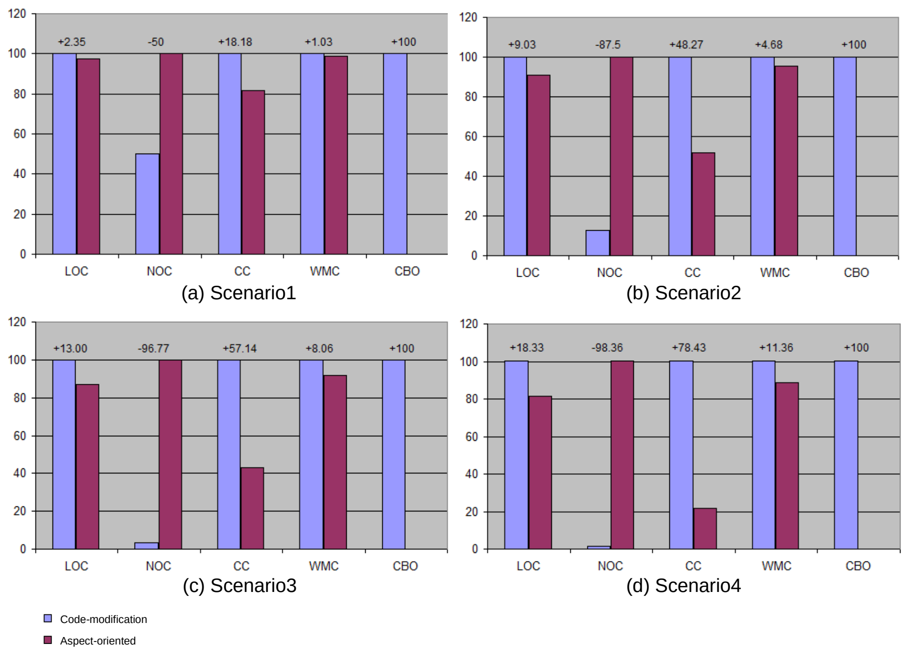


Figure 5.8: Coupling, complexity and size metrics

improved when the number of mismatches increases. On the other hand, the NOC metric is rapidly dropped when the number of mismatches is small. This metric is however gradually declined when the number of mismatches increases.

We compare the results of different scenarios in Figure 5.9. This comparison illustrates that in general the use of aspects slightly improves the LOC and WMC metrics and dramatically improves the CC and COB metrics. With respect to the number of mismatches, the aspect-oriented approach is superior when the number of mismatches is relatively large. In summary, the results show that the proposed patterns and aspect-oriented framework for adapter development have reduced complexity and coupling, and in that sense is easier to understand and maintain.

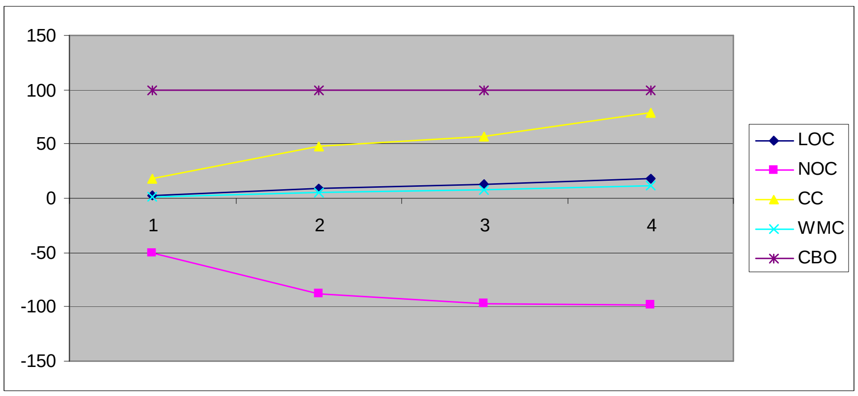


Figure 5.9: Comparison of CK metrics in different scenarios

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., [55, 20, 40, 41, 39]), and also recently for Web services (e.g., [18, 50, 21, 33, 44]). In addition, AOP has received a significant attention in software components [28, 31, 52, 23] and in Web services for the implementation of cross-cutting concerns [35, 49, 30, 24, 54]. 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 [55, 20, 40]. These approaches focus on standalone adapter development and assume that there are no mismatches at the interface-level or that the mapping between component interfaces is provided. However, interface and protocol specifications in Web services are much richer than component specifications. In addition, by nature, they are open to heterogeneities and two services are likely to present mismatches at both interface and protocol levels. This is because services are typically developed by separate teams possibly in different companies although using the same languages (e.g., WSDL and BPEL).

Becker et al. [16] identify 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 non-functional differences of software components. However, the component mismatches and patterns are presented at an abstract level. We focus on Web service interfaces and protocols, and present concrete specification of mismatch patterns, and present a semi-automated 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 [56, 57, 34, 26]. Often 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 characterization of mismatch patterns complements them and helps the adapter developer to identify and capture most of possible differences that are not detected by automated approaches by providing a taxonomy of mismatch patterns and solutions to resolve them.

Web Services Adaptation. The problem of Web services adaptation has received a significant attention [18, 33, 21, 44, 48]. To the best of our knowledge, our work [18] 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 [18], Dumas et al. [33] have identified two other mismatch patterns and proposed operators to handle mismatches. These operators can be composed when developing standalone adapters. Li et al. [45] adopt the mismatch patterns framework to identify five extra mismatch patterns at the interface- and protocol-level in the context of heterogeneous services composition.

In our work [48], we have proposed a semi-automated approach for identifying service mismatches at the interface- (by building on top of approaches in XML schema matching [51]) and 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 presented in this paper is to complement automated approaches (e.g., [48]) by: (i) providing a framework to maintain a taxonomy of mismatch patterns that not all of them can be detected by automated approaches, so to simplify the job of adapter developments, (ii) extending the adapter code development from the standalone approach (which is the focus of [48]) to aspect-oriented approach.

In this paper, we have extended the framework of mismatch patterns [18] by proposing an aspect-oriented adapter development approach. This approach complements standalone adaptation by offering a greater flexibility for managing the lifecycle of business processes. The idea of aspect-oriented approach for adaptation was first introduced in our earlier work [44]. 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 [21] proposes an automated approach for protocol-level (i.e., assuming compatible interfaces) standalone adapter development. We complement their work in that 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 [21, 48] for automatically generating the code for standalone adapters. It should be noted that adopting semantic Web services approaches also does not remove the need for adaptation [22]. 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 also have investigated matching of Web service interfaces, e.g., [32, 53]. 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. Another related work is that of Ponnekanti and Fox [50], in which a framework for handling differences among service interfaces is proposed. 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. A large amount of work has been done in integrating AOP in software components [28, 31, 52, 23]. In many of these work (e.g., [28, 31, 52]), aspects are used to adapt the component to a changing environment at the configuration-level and in the case of component evolution. Camara et al [23], a later work compared to our initial work on aspect-based adaptation [44], present 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, non-functional properties of services find a natural appeal in AOP programming [35]. In [49], Nicoara and Alonso present an aspect-oriented (Java-based) platform that aims to keep services aligned with changes in the environment. AOP has been also used at the process definition level, e.g. in [30, 24]. In [30], 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 [24], Charfi and Mezini propose to use AOP to modularize non-functional concerns, e.g., logging, security, of BPEL processes. This work has been extended to address dynamic changes in service composition in [25]. Unlike their work, we focus on the identification of common mismatches between services and enabling resolutions in either standalone or aspect-oriented adapters. T. Cottenier, et.al [29] extend Axis engine to intercept message and apply AOP to resolve mismatches between messages exchanged between services. Similarly, E. Wohlstadter, et.al [54] 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. We would like to emphasize that, in addition, we provide a classification of common mismatches at each level, capture them as patterns accompanied by aspect-oriented adaptation templates, and an implementation framework.

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 (i) proposing a taxonomy of common mismatches at the service interfaces and business protocols, (ii) a structured approach to the identification of mismatches between services and their resolutions, by introducing mismatch patterns, and (iii) proposing methods and tools for instantiating patterns with two different architectural approaches, standalone adapters and aspect-oriented adaptation. We have shown that aspect-oriented adaptation is a novel and viable approach to resolving the service mismatch problems. Specifically, it is in fact preferable to “traditional” standalone adapters whenever we have access to the service implementation and runtime environment. Moreover, our evaluation results have shown that, in comparison to code-modification, the use of aspects leads to more comprehensible and maintainable code.

The combination of mismatch patterns and aspect-oriented adaptation presents the foundation for rapid adaptation of Web services. Future work in this area will 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 lifecycle.

Bibliography

[1] ActiveBPEL Engine 2.0. http://www.activebpel.org/.

[2] Amazon Ecommerce Service. http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl.

[3] Amazon Flexible Payments Service. https://fps.amazonaws.com/doc/2007-01-08/AmazonFPS.wsdl.

[4] Amazon Web Services. http://soap.amazon.com/schemas2/Amazon-WebServices.wsdl.

[5] AspectJ. www.eclipse.org/aspectj.

[6] Google Checkout. http://code.google.com/apis/checkout.

[7] Mappoint. www.microsoft.com/mappoint/.

[8] Mappoint. www.esri.com/software/arcwebservices/.

[9] Moon Purchase Order Management Service. http://sws-challenge.org/wiki/index.php/Scenario:_Purchase_Order_Mediation.

[10] PaymentExpress Web Service. https://www.paymentexpress.com/WS/-PXWS.asmx?WSDL.

[11] PayPal Web Service. https://www.paypal.com/wsdl/PayPalSvc.wsdl.

[12] RosettaNet. www.rosettanet.org.

[13] XWebCheckout. http://www.xwebservices.com/Web_Services/XWeb-CheckOut.

[14] P. Ajalin and et al. SAP R/3 integration to RosettaNet processes using Web Service interfaces. Technical report, SoberIT, T-86.301, 2004.

[15] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services - Concepts, Architectures and Application*. Springer-Verlag, 2004.

[16] S. Becker and et al. Towards an engineering approach to component adaptation. In *Proc. of Architecting Systems with Trustworthy Components 2004, LNCS 3938*, pages 193–215. Springer, 2006.

[17] C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying Business Processes with BP-QL. In *Proc. VLDB'05*.

[18] B. Benatallah, F. Casati, D. Grigori, H. Nezhad, and F. Toumani. Developing adapters for Web services integration. In *Proc. CAiSE'05*, 2005.

[19] B. Benatallah, F. Casati, and F. Toumani. Representing, analysing and managing Web service protocols. *DKE J.*, 58(3):327–357, 2006.

[20] A. Bracciali, A. Brogi, and C. Canal. A Formal Approach to Component Adaptation. *J. System and Software*, 74(1):45–54, 2005.

[21] A. Brogi and R. Popescu. Automated generation of BPEL adapters. In *Proc. ICSOC'06*, pages 27–39, 2006.

[22] C. Bussler, D. Fensel, and A. Maedche. A conceptual architecture for semantic web enabled Web services. *SIGMOD Rec.*, 31(4):24–29, 2002.

[23] J. Cámara, C. Canal, J. Cubo, and J. M. Murillo. An aspect-oriented adaptation framework for dynamic component evolution. *Electronic Notes Theor. Comput. Sci.*, 189:21–34, 2007.

[24] A. Charfi and M. Mezini. Aspect-Oriented Web Service Composition with AO4BPEL. In *Proc. ECOWS'04*, pages 168–182.

[25] A. Charfi and M. Mezini. AO4BPEL: An Aspect-oriented Extension to BPEL. *The World Wide Web J.*, 10(3):309–344, 2007.

[26] P. Chen, M. Critchlow, A. Garg, C. van der Westhuizen, and A. van der Hoek. Differencing and Merging within an Evolving Product Line Architecture. In *Proc. PFE'03*, pages 269–281.

[27] S. Chidamber and C. Kemerer. A metrics suite for object-oriented design. *IEEE TSE*, 20(6):476–493, 1994.

[28] A. Colyer, A. Clement, R. Bodkin, and J. Hugunin. Using aspectj for component integration in middleware. In *OOPSLA'03*, pages 339–344.

[29] T. Cottenier and T. Elrad. Executable choreography processes with aspect-sensitive services. In *Proc. AERO'06*, pages 13–30.

[30] C. Courbis and A. Finkelstein. Towards Aspect Weaving Applications. In *Proc. ICSE'05*, pages 69–77.

[31] A. Dantas, J. W. Yoder, P. Borba, and R. Johnson. Using aspects to make adaptive object-models adaptable. In *Proc. RAM-SE'04*, pages 9–19, 2004.

[32] X. Dong, A. Y. Halevy, J. Madhavan, E. Nemes, and J. Zhang. Simlarity search for Web services. In *Proc. VLDB'04*, pages 372–383, 2004.

[33] M. Dumas, M. Spork, and K. Wang. Adapt or perish: Algebra and visual notation for service interface adaptation. In *Proc. BPM'06*, pages 65–80, 2006.

[34] M. A.-A. et.al. Differencing and Merging of Architectural Views. Technical report, Carnegie Mellon University, ISRI-05-128R, 2005.

[35] N. L. et.al. Survey of aspect-oriented middleware research. Technical report, Lancaster University, AOSD-Europe-ULANC-10, June 2005.

[36] E. Gamma, R.Helm, R.Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[37] A. Garcia and et al. Modularizing design patterns with aspects: a quantitative study. In *Proc. AOSD '05*, pages 3–14, 2005.

[38] J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. *SIGPLAN Not.*, 37(11):161–173, 2002.

[39] D. Hemer. A formal approach to component adaptation and composition. In *Australasian conference on Computer Science*, pages 259–266, 2005.

[40] P. Inverardi, L. Mostarda, M. Tivoli, and M. Autili. Synthesis of correct and distributed adaptors for component-based systems: an automatic approach. In *Proc. ASE'05*, 2005.

[41] W. Jiao and H. Mei. Automating integration of heterogeneous cots components. In *Proc. ICSR'06*, 2006.

[42] R. Kazhamiakin and M. Pistore. Choreography Conformance Analysis: Asynchronous Communications and Information Alignment. In *Proc. WS-FM'06*, pages 227–241, 2006.

[43] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ. In *Proc. ECOOP'01*, pages 327–353.

[44] W. Kongdenfha, R. Saint-Paul, B. Benatallah, and F. Casati. An aspect-oriented framework for service adaptation. In *Proc. ICSOC'06*, 2006.

[45] X. Li, Y. Fan, and F. Jiang. A classification of service composition mismatches to support service mediation. In *Proc. GCC'07*, pages 315–321, 2007.

[46] S. Murugesan. Understanding web 2.0. *IEEE IT Professional*, 9(4):34–41, 2007.

[47] H. R. M. Nezhad, B. Benatallah, F. Casati, and F. Toumani. Web services interoperability specifications. *IEEE Computer*, 39(5):24–32, 2006.

[48] H. R. M. Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-automated adaptation of service interactions. In *Proc. WWW'07*, pages 993–1002, 2007.

[49] A. Nicoara and G. Alonso. Dynamic AOP with PROSE. In *Proc. CAiSE'05*, pages 125–138.

[50] S. Ponnekanti and A. Fox. Interoperability among independently evolving Web services. In *Middleware'04*, pages 331–351, 2004.

[51] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4):334–350, 2001.

[52] C. C. Soria, J. Pérez, and J. A. Carsí. Dynamic adaptation of aspect-oriented components. In *Proc. CBSE'07*, pages 49–65, 2007.

[53] Y. Wang and E. Stroulia. Flexible interface matching for web-service discovery. In *Proc. WISE '03*, 2003.

[54] E. Wohlstadter and K. Volder. Doxpects: aspects supporting XML transformation interfaces. In *Proc. AOSD'06*, pages 99–108.

[55] D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM TOPLAS*, 19(2):292–333, 1997.

[56] A. M. Zaremski and J. M. Wing. Signature matching: a tool for using software libraries. *ACM TOSEM*, 4(2):146–170, 1995.

[57] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM TOSEM*, 6(4):333–369, 1997.