Developing Adapters for Web Services Integration

Boualem Benatallah[1], Fabio Casati[2], Daniela Grigori[3],
Hamid R. Motahari Nezhad[1,4], and Farouk Toumani[5]

[1] SCSE, University of New South Wales, Sydney NSW 2052, Australia
{boualem, hamidm}@cse.unsw.edu.au
[2] HP Labs, Palo Alto, CA, 94304 USA
fabio.casati@hp.com
[3] PriSM, Université de Versailles, 45 avenue des Etats-Unis,
78035 Versailles Cedex, France,
daniela.grigori@prism.uvsq.fr
[4] NICTA, Australian Technology Park, Bay 15 Locomotive Workshop,
Sydney NSW 1430, Australia,
[5] LIMOS, ISIMA, Campus des Cezeaux, BP 125, 63173 Aubière Cedex, France
ftoumani@isima.fr

Abstract. The push toward business process automation has generated the need for integrating different enterprise applications involved in such processes. The typical approach to integration and to process automation is based on the use of adapters and message brokers. The need for adapters in Web services mainly comes from two sources: one is the heterogeneity at the higher levels of the interoperability stack, and the other is the high number of clients, each of which can support different interfaces and protocols, thereby generating the need for providing multiple interfaces to the same service. In this paper, we characterize the problem of adaptation of web services by identifying and classifying different kinds of adaptation requirements. Then, we focus on business protocol adapters, and we classify the different ways in which two protocols may differ. Next, we propose a methodology for developing adapters in Web services, based on the use of mismatch patterns and service composition technologies.

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* [YeSt97, CFPT03]. Adaptors 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. Message brokers, and message-oriented middleware in general, provide an efficient and reliable way to transport messages (typically corresponding to operation invocations or

their replies) among the adapters, which in turn interact with the wrapped application. While very effective and relatively successful, this approach presents several limitations. In particular, process integration efforts require a high number of different adapters, as the level of heterogeneity in IT infrastructures is typically very high. Furthermore, whenever new versions of the wrapped applications are deployed, adapters need to be modified to account for the differences in protocols and formats supported by these new versions. This is also why enterprise application integration (EAI) platforms are very expensive.

Web services were born as a solution to (or at least as a simplification of) the integration problem [ACKM04]. 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. Standardization reduces heterogeneity and makes it therefore easier to develop business logic that integrates different (Web service-based) applications. Web services also represent the most promising technologies for the realization of service-oriented architectures (SOAs), not only within but also outside companies' boundaries, as they are designed to enable loosely-coupled, distributed interaction [BeCT04].

While standardization makes interoperability easier, it does not remove the need for adapters. In fact, although the lower levels of the interaction stacks are standardized (as discussed later), different Web services may still support different interfaces and protocols. In addition, the novel opportunities enabled by Web services have an implication in terms of adaptation needs. In fact, having loosely-coupled and B2B interactions imply that services are not designed having interoperability with a particular client in mind (as it was often the case with CORBA-style integration) [CFPT03]. They are designed to be open and possibly without knowledge, at development time, about the type and number of clients that will access them, which can be very large. The possible interactions that a Web service can support are specified at design time, using what is called a *business protocol* or *conversation protocol* [BeCT04]. 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. This is important, as it rarely happens that service operations can be invoked at will independently from one another. Hence, adaptation should not be limited to handling heterogeneity at the operation level, but also at the business protocol level.

This paper presents a framework for developing Web service adapters. First, we characterize the problem of adaptation by identifying and classifying different kinds of adaptation needs (Section 2). Then, we focus on interface and business protocol adapters and we classify the different ways in which two interfaces and protocols may differ (Section 3). These differences are captured using *mismatch patterns*. Patterns help users in analyzing differences and in resolving them. In fact, among other information, patterns include a template of business logic that can be used to semi-automatically develop adapters to handle the mismatch captured by each pattern. We provide a number of built-in patterns corresponding to the possible mismatch we have identified at the interface and protocol levels. Finally, we discuss related work, conclusions and future directions in Sections 4 and 5.

2 Toward a Methodology for Web Service Adapters

This section presents an overview of the proposed approach to semi-automated development of service adapters. We first characterize the interoperability problem in general, then we define the focus of our work, and finally we describe at a high level the approach we adopt.

2.1 Interoperability at Business-Level Interfaces and Protocols

Interoperability among Web services, just like interoperability in any distributed system, requires that services use the same (or compatible) protocols, data formats, and semantics. In our work, we focus on interoperability issues at business-level interfaces and protocols. To interact, services must have compatible:

- Interfaces (i.e., the set of operations supported by services),
- Business protocols (i.e., the allowed message exchange sequences). These can be expressed for example using BPEL abstract processes, WSCI, or other protocol languages (see, e.g., [BeCT04]).

More precisely, we classify the need for adaptation in Web services in 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, offered by provider SP, allowing companies to order office supplies. If SP wants to be able to do business with certain retailers (say, *Wal-Mart* or *Target*), then it needs to adapt its service S so that it can interoperate with these retailers. In general, many adapters can be defined depending on the number of different client protocols that SP must interact with. Hence, in this case, adaptation is performed by considering the client's protocol. Note that adaptation may be required for one or more of the interoperability layers identified above, since for two services to interact, compatibility must be achieved at all layers.

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 consortium standardizes the external behaviour 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 behaviour, 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* [BeCT04a]. Total replaceability occurs when a service SR 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 SR (note that the opposite is not necessarily true). Partial replaceability occurs when a service SR can behave like S only in certain interactions (that is, SR behaves like S in some but not all conversations). For example, an ordering service SR may need or be

able to replace ordering service S only for orders of certain products but not other, or may able to process all orders but does not allow cancellations, while S does. We refer the reader to [BeCT04a] for a detailed definition of compatibility and replaceability among services, as well as other important relations among different elements of service descriptions.

This paper proposes a technique for developing adapters to achieve total replaceability. As mentioned above, this is a very important and relevant problem. The related issues of partial replaceability and compatibility can be handled in an analogous manner. We also decided to initially focus on developing adapters to resolve differences at the interface and business protocol level. Replaceability at the lower layers has been either addressed by standardization (e.g., messaging) or has been the subject of excellent research work by other groups [RyWo], and hence research on protocol replaceability constitutes the next level up the interoperability stack in supporting interaction among services. Incidentally, although we discuss the problem of protocol replaceability in the context of business protocols, analogous techniques can be used for other service aspects characterized by protocols (e.g., trust negotiation protocols or basic coordination).

2.2   Developing Service Adapters Using Mismatch Patterns

The intended benefit of this work is to help programmers develop adapters through a methodology and semi-automated code development, starting from the protocol definitions. The adapters have the goal of making a service SR, characterized by protocol PR, "look like" (interact as) another service S that has protocol P, so that SR can then interact with any client that can interact with S (see Figure 1).
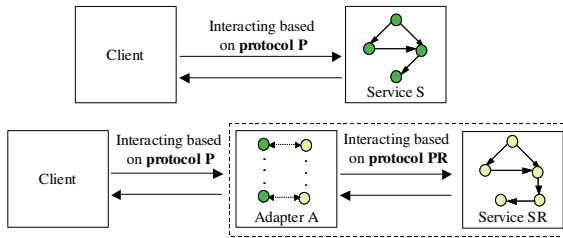

Fig. 1. Adapters allows achieving protocol replaceability

Hence, the adapter A for SR is a Web service that, to clients, behaves like S from an interaction perspective. In particular, if S supports protocol P, then adapter A also supports protocol P when interacting with clients. Adapter A will implement protocol P by invoking methods of SR. From the perspective of SR, A looks like a service whose protocol is *compatible* with PR (Figure 1).

The approach proposed in this paper to adapter development is based on mismatch patterns, which are design patterns that can be used to capture the possible differences among services (and specifically among interfaces and protocols). We have analyzed interfaces and protocols to identify common differences and for those we have specified the corresponding mismatch patterns. Indeed we believe that the identification of

the various kinds of differences among interfaces and, most of all, protocols, is a contribution in itself. Developers can, however, add to the set of 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.

Besides capturing differences, patterns can be used both as guidelines for designer in developing adapters and as input to a tool that automatically generates the adapter code. In fact, mismatch patterns contain both formal and informal descriptions of the type of adapter (called *adapter template*) used to resolve that type of mismatch. The table below summarizes the structure of an adapter pattern. In the following of this section we discuss and exemplify in more detail the part related to adapter templates and their instantiation.

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

To describe the approach to adapter template specification and adapter generation and to motivate our choices, we begin by discussing what is expected of an adapter and how they can be modelled and implemented. As mentioned above, the job of an adapter consists in mapping interactions with protocol P into interactions with protocol PR. This requires performing activities such as receiving messages, storing messages, transforming message data, and invoking service operations. These tasks can be very well modelled by process-centric service composition languages such as BPEL (http://www-128.ibm.com/developerworks/library/ws-bpel/). Hence, our aim is to leverage patterns to manually or automatically generate process skeletons that map interactions according to protocol P into interactions according to protocol PR. Analogous solutions can be identified if third-generation programming languages like Java or C# or if other process languages instead of BPEL are used. In any case, the generated specifications can be then enacted by the corresponding execution engine (e.g., a BPEL engine, or a Java virtual machine). We chose a process-based notation because it is well-suited to model business logic and because it is easy to derive the protocol specifications of a service when its implementation is specified as a business process [BBCT04] (although, as we will see, other aspects such message transformations need to be modelled). A high-level process-based notation is also appropriate to compose complex adapters from primitive adapter templates, possibly leveraging one of the many process management tools available on the market. In addition, this notation can be mapped to others (e.g., state machines or state-charts), which are endowed with formal semantics. Using a high level notation allows, e.g., using formal analysis techniques to verify the correctness of adaptors.

Given that we aim at generating and customizing a process definition, it was natural to select a process language for defining the adapter templates as well, as using a

similar modelling framework simplifies adapter generation, especially when it is per-
formed manually. Indeed, we borrow BPEL notation, concepts, and terminology for
this purpose, endowed with additional annotations to specify *adaptation abstractions*.
In particular, the additional annotations may include XQuery
(www.w3.org/TR/xquery/) functions to specify message transformations that are
commonly needed in adapters, directives to help developers understand how to instan-
tiate certain elements of the adapter template.

Example. As an example, consider the MapPoint (www.microsoft.com/mappoint/)
and Arcweb (www.esri.com/software/arcwebservices/) route Web services, which of-
fer similar functionalities for finding driving routes between two points using differ-
ent WSDL interfaces (operations `CalculateRoute` and `findRoute`, respectively).
Suppose that Arcweb corresponds to service SR and MapPoint to service S according
to the architecture presented in Figure 1[1]. The names, number, and types of the in-
put/output parameters of the operations `CalulateRoute` and `findRoute` differ.
The operation `CalculateRoute` requires one input parameter called Specification
whose type is `SegmentSpecification`. The operation `findRoute` requires two
parameters: `routeStops` and `routeFinderOptions` whose types are `RouteStops`
and `RouteFinderOptions`, respectively. The values of both parameters `routeS-
tops` and `routeFinderOptions` can be computed from the value of the parameter
`Specification`.

This type of difference is handled by a mismatch pattern called *SMP (Signatures
Mismatch Pattern)*. This pattern concerns differences that occur when two services S
and SR have operations that have the same functionality but differ in operation name,
number, order or type of input/output parameters. In general, adapters that resolve this
kind of differences need to perform the actions described below, which therefore con-
stitute our adapter template for this pattern (written here in pseudo-code for ease of
presentation):

Template Parameters	**Signatures of operations O of service S and OR of service SR, XQuery functions for message transformations**
Adapter Template	*Receive* the input message OI of operation O from client (*BPEL receive activity*)
	Transform OI into a format that is compliant with the type of input message ORI of operation OP, using XQuery transformation functions (one or more *BPEL assign activity*, depending on the parameters to be transformed)
	Invoke operation OR (*BPEL invoke activity*)
	Transform output message ORO of operation OR into a format that is compliant with the type of the output message OO of operation O, using XQuery transformation functions (one or more *BPEL assign activity*, depending on the parameters to be transformed)
	Send reply of operation O to client (*BPEL reply activity*)

Note that the template is parametric: to instantiate it and generate an executable
BPEL process, the user needs to provide several parameters. In this case, the parame-
ters are the signatures of the operations that have a mismatch and the XQuery trans-

---

[1] The idea for using the arcweb vs MapPoint example is taken from [PoFo04].

formation functions. The parameters that the user needs to specify are part of the *template parameters* field of the pattern. This information is used to (manually or automatically) generate a process skeleton from the template. The developer may then want or need to further customize the resulting process skeleton to add some custom business logic, or can just directly use the generated process skeleton to deploy the adapter. For built-in patterns, we have automated code that actually generates adapters given the pattern name and template parameters. Note that complex adapters, i.e., those resolving several mismatch types, can be constructed by composing primitive templates.

Figure 2 shows an adapter that resolves the signature mismatch among operation CalculateRoute of S and operation findRoute of SR according to the adapter template of SMP pattern.
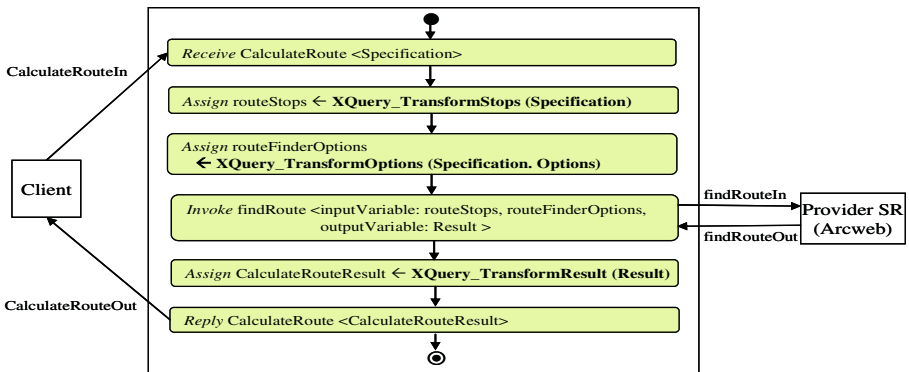

Fig. 2. Sample usage of SMP

In the process skeleton, the parts in bold indicate the parameters that are provided by the adapter developer. The symbol "<>" is used to denote parameters of operations. We also identify the input and output messages of operations by adding "In" and "Out" to the end of operation name in the examples.

In Figure 2, the adapter first receives a message that contains the value of the input parameter Specification of operation CaculateRoute (hence behaving like S). Then it 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. These functions are specified by the developer, possibly by using one of the many XQuery tools being developed by major software vendors. After performing message transformations, the adapter invokes the operation findRoute of SR (Arcweb), and performs symmetric actions on the reply. In this example, the configuration of the template activities consists of specifying XQuery functions, namely XQuery_TransformStops, XQuery_TransformOptions and XQuery_TransformResults.

We conclude the section by pointing out aspects that are outside the scope of this work. The work in this paper does not address the problem of automatically identifying differences between two actual protocols. For example, a type of difference occurs when protocol P requires two messages, m_a and m_b to be in sequence, while PR allows

them to be in any order. Referring to this example, we do not present here a mechanism for finding out that the difference between P and PR consists in the different ordering constraints on m_a and m_b. The problem of identifying the actual differences constitutes a separate research thread in itself and is extremely complex. This is, however, an orthogonal issue. In this paper, we assume that an analyst will identify the differences and the corresponding pattern. For example, an analyst (or, in the future, a tool) will look at P and PR and identify that there is a difference of type ordering constraint involving m_a and m_b. From there, we derive the corresponding mismatch pattern that resolves the difference so that the adapter can appear as supporting protocol P and is implemented by invoking operations of PR.

3 Characterizing and Resolving Differences Between Business Protocols

This section describes our approach for developing Web service adapters at the interface and business protocol levels. For each level, we present a taxonomy of possible mismatches and propose a solution to tackle each kind of mismatch. The rationale behind the proposed taxonomy is to characterize differences based on how we can approach/solve them. For each pattern we also provide an example (which corresponds to a sample usage for that pattern).

3.1 Differences at the Operation Level

In addition to SMP pattern that we described in section 2.3 for resolving operation signatures mismatch, in this section we describe a mismatch pattern called *PCP (Parameter Constraints Pattern)* that handles parameter constraints mismatch as described below. This type of mismatch occurs when the operation O of S imposes constraints on input parameters, which are less restrictive than those of OR input parameters in SR (e.g., differences in value ranges).

Template Parameters	Signatures of operations O of service S and OR of service SR, XQuery functions for checking parameter constraints
Adapter Template	*Receive* the input message OI of operation O from client (*BPEL receive activity*)
	If OI *verifies* OR constraints (*BPEL switch activity*):
	Then *Invoke* operation OR (*BPEL invoke activity*)
	Send reply of operation O to client (*BPEL reply activity*)
	Else *Raise* a constraint-violation exception and terminate conversation (*BPEL reply activity*)

In this adapter template, input messages of operation O are first checked to verify if they are compliant with OR constraints. 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. But, 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 `Pref-`

erence. If the value of `RouteType` is in {"quickest", "shortest"}, the adapter will forward the invocation message to Arcweb. Otherwise, the adapter will raise a constraint violation exception. Figure 3 shows an adapter resolving this constraint mismatch. We observe again that in the case of built-in patterns we have pattern-specific code that generates the adapter given the user-defined parameters. However, the same can be done manually by the developer by looking at the adapter template field of the pattern and at the sample usage. Note that constraint-checking conditions is expressed using XQuery queries. For example, the condition `Verify_Specifica tion_Constraints` checks if Specification verifies the constraints of `RouteType`.
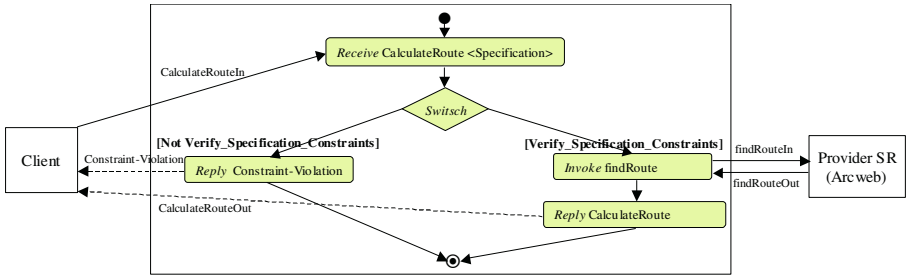

Fig. 3. Sample usage of PCP

3.2 Differences at the Protocol Level

We now consider the problem of developing adapters to resolve service mismatches that occur at the protocol level. We build our approach using the extend business protocol model presented in [BeCT04]. That protocol model allows a richer description of the external behavior of a service by providing specific abstractions that enable, for example, to model temporal and transactional properties of service conversations.

In this section we use a supply chain example to illustrate adapter templates. For instance, protocol P may expect 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 PR allows the following sequence of operations: `login`, `getCatalogue`, `submitOrder`, `issueInvoice`, `makePayment` and `sendShippingPreferences`. This is possible, e.g., because provider SR does not charge differently according to the shipping preferences. Clients are allowed to specify their shipping preferences at a final step. Note that for the sake of clarity, we omitted the acknowledgements from the message sequences.

Message Ordering Mismatch
This type of difference is concerned with the order in which protocols expect to receive certain message. Mismatch occurs when protocols P and PR support the same message but in different orders. This type of difference is handled by a mismatch pattern called *OCP (Ordering Constraint Pattern)* described below:

Template Parameters	**Protocols P and PR, message m to be re-ordered**
Adapter Template	*Perform* activities as prescribed by P for parts that do need adaptation (*BPEL receive, invoke, reply activities*) *Receive* message m according to protocol P (*BPEL receive activity*) *Store* m in the adapter (*BPEL assign activity*) *Send* m to SR when it is expected (*BPEL invoke activity*)

Figure 4 shows an adapter that resolves the ordering constraints for the message `sendShippingPreferencesIn`. From the input parameters of the template, it is possible to determine the message ordering constraints of protocols P and PR. In this case, the adapter can temporarily store the parameter of operation `sendShipping-Preferences` of protocol P and forward the operation to service SR according to the messages choreography of protocol PR. Note that the adaptation will be more complex if protocol P issues a reply or acknowledgement for the `sendShippingPreferencesIn` message and the client expect receiving such message. We will discuss such a scenario in the following when we discuss the need for generating missing messages.
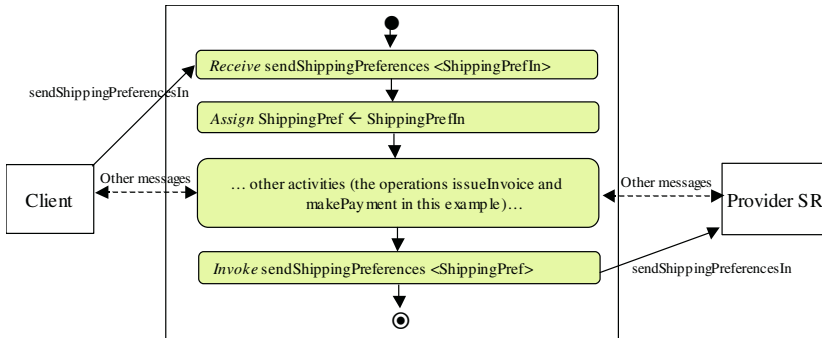

Fig. 4. Sample usage of OCP

Extra Message Mismatch

This type of differences occurs in situations where protocol PR issues an extra message that protocol P does not issue. This type of difference is handled by a mismatch pattern called *EDP (Extra Message Pattern)*. The adapter template of EDP allows intercepting and discarding the extra message in order to make PR look like P. It should be noted that such an adaptation makes sense only if the extra message of PR does not affect the semantics of the target protocol (i.e., does not change the functionality of PR).

Template Parameters	**Protocols P of S and PR of SR, message m of PR to be discarded**
Adapter Template	*Perform* activities as prescribed by P for parts that do need adaptation (*BPEL receive, invoke, reply activities*) *Discard* m when received (*BPEL receive activity*)

In the supply chain scenario, assume that protocol PR sends an acknowledgement after receiving message `issueInvoiceIn` but protocol P does not. Figure 5 shows an adapter that when receives the message `InvoiceAck`, it discards it (does not perform any action).
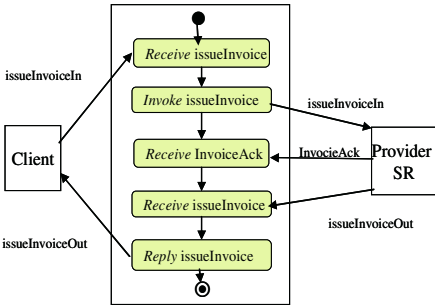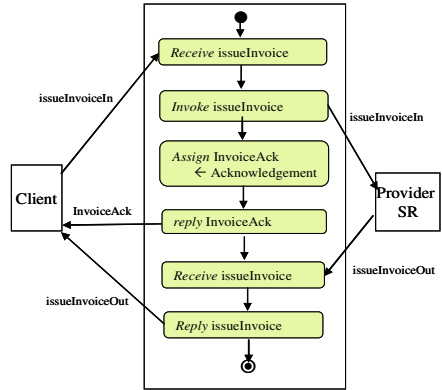

Fig. 5. Sample usage of EMP

Fig. 6. Sample usage of MMP

Missing Message Mismatch

This type of differences occurs when protocol P issues an extra message that protocol PR does not issue. It should be noted that this extra message does not affect the semantics of the protocol PR. This type of difference is handled by a mismatch pattern called *MMP (Missing Message Pattern).* The adapter template of MMP generates a new message to make PR look like P.

Template Parameters	**Protocols P of S and PR of SR, message m of P to be generated**
Adapter Template	*Perform* activities as prescribed by P for parts that do need adaptation (*BPEL receive, invoke, reply activities*) *Generate* m when expected by P (*BPEL assign activity*) *Reply* m according to P (*BPEL reply activity*)

Consider the opposite case of the previous example, where protocol P issues an acknowledgement when receiving a request for invoice (i.e., the message `issueInvoiceIn`), while protocol PR does not. Figure 6 shows an adapter that generates the message `InvoiceAck` and sends it to the client after receiving the message `issueInvoiceIn` and invoking the operation `issueInvoice` of SR.

Message Split Mismatch

This type of differences occurs when the protocol P requires a single message to achieve certain functionality, while in protocol PR the same behavior is achieved by receiving several messages. This type of difference is handled by a mismatch pattern called *OMP (One to Many messages Pattern)* described below:

Template Parameters	**Protocols P and PR, message m of P to be split and messages mr_1, ..., mr_n of PR to be extracted from m, XQuery functions for parameters extraction**
Adapter Template	*Perform* activities as prescribed by P for parts that do need adaptation (*BPEL receive, invoke, reply activities*)
	Generate mr_1, ..., mr_n from m when m is received, send mr_1, ..., mr_n as prescribed by PR (*BPEL assign, invoke activities*)

Suppose that protocol P requires to receive the purchase order as well as shipping preferences in one message called submitOrderIn, while protocol PR needs two separate messages for this purpose, namely, sendShippingPreferencesIn and submitOrderIn. Figure 7 shows an adapter that resolves this mismatch. In this case, when the adapter receives submitOrderIn, it generates the parameters of the operations sendShippingPreferences and submitOrder of PR from the parameter of the operation submitOrder of P, using XQuery transformation functions, namely XQuery_SplitShipping and XQuery_SplitOrder. Following that messages submitOrderIn and sendShippingPreferencesIn are forwarded to SR. Note that in other cases, since messages are stored in the adapter, the adapter can forward them to SR in the order prescribed by PR.
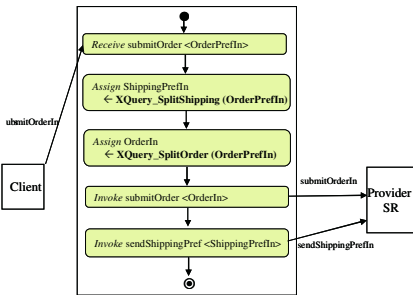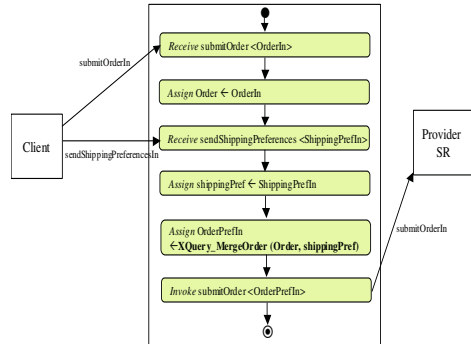

Fig. 7. Sample Usage of OMP   **Fig. 8.** Sample usage of MOP

Message Merge Mismatch

This type of differences occurs when protocol P needs to receive several messages for achieving certain functionality while protocol PR requires one message to achieve the same functionality. This type of difference is handled by a mismatch pattern called *MOP (Many to One message Pattern)* described below.

Template Parameters	**Protocols P and PR, messages m_1, ..., m_n of P to be merged into message mr of PR, XQuery function for parameter computation**
Adapter Template	*Perform* activities as prescribed by P for parts that do need adaptation (*BPEL receive, invoke, reply activities*)
	Receive m_1, ..., m_n according to P (*BPEL receive activities*) and store them until mr is generated (*BPEL assign activities*)
	Generate mr by merging m_1, ..., m_n when mr is expected by PR (*BPEL assign, invoke activities*)

Suppose that protocol P requires messages `submitOrderIn` and `sendShipping-PreferecesIn` separately, but protocol PR needs all of this information included in the `submitOrderIn` message. Figure 8 shows an adapter that resolves this mismatch. In this template, when the adapter receives the messages `submitOrderIn` and `send-ShippingPreferencesIn`, it generates the parameter of operation `submitOrder` of PR using an XQuery function, namely `XQuery_MergeOrder` that merges the parameters of the operations `submitOrder` and `sendShippingPreferences` of P. The adapter knows the order of messages of protocols P and PR from the input, so it is able to generate receive and storage activities for messages and to invoke operations of SR according to PR.

It should be noted that differences can be complex (i.e., cannot be reduced to one of the above primitive patterns). Adapters resolving several mismatch types can be constructed by composing primitive templates. We plan to extend the set of protocols management operators provided in our framework and reported in [BeCT04a] to cater for adapter templates composition.

4 Related Work

Although a lot of work and progress has already been done in the area of web services in the last few years, efforts have been mostly focused on service description models and languages, and on automated service discovery and composition [ACKM04].

Very recently, authors in the academia have published papers that discuss similarity and compatibility at different levels of abstractions of a service description (e.g., [BeCT04a, Bordeaux04, DHMN+04, PoFo04, WMFN04]). In terms of protocols specification and analysis, existing approaches provide models (e.g., based on pi-calculus or state machines) and mechanisms to compare specifications (e.g., protocols compatibility and replaceability checking). In particular, the work that is more related to ours, also in terms of overall line of research, is that of Lenezirini and Mecella's group. Specifically, in [Bordeaux04] a protocol analysis framework based on concepts analogous to those of replaceability and compatibility is presented. In [PoFo04], the authors proposed a framework for handling differences among service interfaces, but protocols are not discussed.

The framework we propose in this paper builds upon this previous work as well as on work we did over the past three years in the area of service protocols modeling, analysis, and management to classify differences among service protocols providing similar functionalities and bridge these differences via adapters (see [BeCT04, BeCT04a] for representative examples of our line of research).

Our aim is to provide a comprehensive framework for managing differences at various abstraction layers including interface, protocol, and policy aspects. In this paper, we focus on both interface and protocol levels. To the best of our knowledge, there is no existing work that considers the classification and management of differences between service protocols and the development of adapters to resolve them.

In the software engineering area, few approaches exist for analyzing software components mismatch. In [ZaWi97], the authors focus on analyzing differences related to data types and to pre- and post-conditions in component interfaces. In [YeSt97, CFPT03] the focus is on specifying interface mappings and using these

mappings to build component adaptors. These efforts provide mechanisms that can be leveraged for Web service protocols adaptation, but are not sufficient. In fact, service protocols require richer description models than component interfaces. This is because clients and services are typically developed by separate teams, possibly even by different companies, and service descriptions are all client developers have to understand to know how the service behaves.

5 Discussion and Ongoing Work

We argue that, while standardization is crucial in making service oriented computing a reality, the effective use and widespread adoption of service technologies and standards requires high-level frameworks and methodologies for supporting automated development and interoperability (e.g., mechanisms for analyzing protocol compatibility, replaceability and compliance, semi-automated generation of adaptors). We see the evolution of the work in Web services interoperability as mirroring in a way, at least conceptually, the work done in databases over the last thirty years and leading to standard models and languages, algebras, theoretical foundations, and transformation techniques. We believe that Web services protocols require the same kind of background work in terms of simple models, operators, algebras, and support for manipulation/transformation. The work we have been doing goes in this direction, and the research presented in this paper is a key part of it. Indeed. The framework presented in this paper is one of the components of a broader CASE tool, partially implemented, that manages the entire service development lifecycle.

In this paper, we focused on identifying differences among heterogeneous business protocols in Web services and on semi-automatically resolving such differences when possible via adapters. We have identified mismatch patterns as a convenient mean to capture the differences among interface/protocols of two services and to encapsulate the solution to such differences. This solution is in the form of process fragments that can be manually or automatically instantiated to generate actual adapters. Other components of our framework focus on modeling and specifying service abstractions: composition logic, business protocols, and trust negotiation policies [BeCT04, SBC03]. Based on these models, service lifecycle can be automated, from development through analysis, management, and enforcement [BBCT04, BeCT04a]. We believe that this work will result in a comprehensive methodology and platform that can support large-scale interoperation of Web services and facilitate service lifecycle.

As the reader will have noticed, there are several aspects that we have not discussed in this paper, some due to lack of space, others because we have not developed a solid, validated solution as yet. An issue is that of automated code generation in relation to the adapter template. In the current framework, the code for generating the adapter is pattern-specific, that is, each pattern comes with an associated function that takes actual values for the pattern parameters as input and generates the adapter code. If a user wants to develop a new pattern, then the functions must also be provided. In reality, in the current version of the framework these functions do not access the adapter templates, which can therefore be specified semi-formally (its main purpose right now is to help the developer understand the pattern and its solution, especially in case a manual adapter generation is needed). In the future we plan to define a formal

language (an extension of BPEL) that can be used to formally specify adapter templates. Then, generic (as opposed to pattern-specific) code will generate adapters by reading the adapter template. As mentioned, we use annotated BPEL for this purpose, but at the time of writing we have not finalized the language design nor the generation code. We believe this will be an improvement as pattern developers can use this BPEL-like language to specify the template and do not need to write low-level code for adapter generation.

References

[ACKM04] Alonso, G., Casati, F., Kuno, H., Machiraju, V.: *Web Services: Concepts, Archtectures, and Applications*. Springer Verlag (2004).

[BBCT04] Baina, K., Benatallah, B., Casati, F., Toumani, F.: *Model-Driven Web Service Development.* Procs of CAiSE'04, Riga, Latvia (2004).

[BeCT04] Benatallah, B., Casati, F., Toumani, F.: *Web services conversation modeling: A Cornerstone for E-Business Automation.* IEEE Internet Computing, vol. 8, no. 1 (2004).

[BeCT04a] Benatallah, B., Casati, F., Toumani, F.: *Analysis and Management of Web Services Protocols.* ER'04, Shanghai, China (2004).

[Bordeaux04] Bordeaux, L., et al: *When are two Web Services Compatible?.* VLDB TES'04. Toronto, Canada (2004).

[CFPT03] Canal, C., Fuentes, L., Pimentel, E., Troya, J., Vallecillo, A.: *Adding Roles to CORBA Objects.* IEEE TSE, vol. 29, no. 3 (2003).

[DHMN+04] Dong, X., A., Halevy, Y., Madhavan, J., Nemes, E., Zhang, J.: *Similarity Search for Web Services.* VLDB'04. Toronto, Canada (2004).

[PoFo04] Ponnekanti, S. R., Fox, A.: *Interoperability among Independently Evolving Web Services.* Middleware'04. Toronto, Canada (2004).

[RyWo] Ryan, N. D., Wolf, A. L.: *Using Event-Based Translation to Support Dynamic Protocol Evolution.* ICSE'04. Edinburgh, Scotland, United Kingdom (2004).

[SBC03] Skogsrud, H., Benatallah, B., Casati, F.: *Model-Driven Trust Negotiation for Web Services.* IEEE Internet Computing, vol. 7, no. 6 (2003) 45-52.

[YeSt97] Yellin, D. M., Strom, R. E.: *Protocol specification and Component adaptors*. ACM TOPLAS, vol. 19, no. 2 (1997).

[WMFN04] Wombacher, A., Mahleko, B., Fankhauser, P., Neuhold, E.: *Matchmaking for Business Processes based on Choreographies*. EEE'04, Taipei, Taiwan (2004).

[ZaWi97] Zaremski, A. M., Wing, J. M.: *Specification Matching of Software Components*. ACM TOSEM, vol. 6, no. 4 (1997).