An Aspect-Oriented Framework for Service Adaptation

Woralak Kongdenfha[1], Régis Saint-Paul[1],
Boualem Benatallah[1], and Fabio Casati[2]

[1] SCSE, University of New South Wales, Sydney, NSW, 2052, Australia
{woralakk, regiss, boualem}@cse.unsw.edu.au
[2] DIT, University of Trento, Via Sommarive 14, I-38050 POVO (TN), Italy
casati@dit.unitn.it

Abstract. Web services are emerging technologies for integrating heterogeneous applications. In application integration, the internal services are interconnected with other external resources to form a virtual enterprise. This puts new requirements on the standardization in terms of external specification, i.e., a combination of service interfaces and business protocols, that interconnected services have to obey. However, previously developed service implementations do not always conform to the standard and require adjustment.

In this paper, we characterize the problem of aligning internal service implementation to a standardized external specification. We propose an Aspect oriented framework as a solution to provide support for service adaptation. In particular, the framework consists of i) a taxonomy of the different possible types of mismatch between external specification and service implementation, ii) a repository of aspect-based templates to automate the task of handling mismatches, and iii) a tool to support template instantiation and their execution together with the service implementation.

1 Introduction

The essence of service-oriented computing (SOC) lies in the creation of loosely coupled, reusable components that can be invoked and composed by clients. In a successful SOC environment, a service may invoke several other services as part of its execution, and may in turn be invoked by several clients. The creation of such modular components and of the infrastructure for their secure and reliable invocation is a challenging endeavor that is being tackled by scores of companies and researchers through novel technologies, methodologies, and standards.

The realization of SOC raises the need for methodologies and tools to manage *service adaptation*. Service adaptation refers to the problem of modifying a service so that it can correctly interact with another service, overcoming functional and non-functional mismatches and incompatibilities.

There are two main situations in which adaptation is needed, both likely to be fairly common in SOC. In a first scenario, the ACME company offers

A. Dan and W. Lamersdorf (Eds.): ICSOC 2006, LNCS 4294, pp. 15–26, 2006.

a service that implements some business process (e.g., a quotation and ordering process). As part of exposing a service to clients, the company also provides the *external specifications* of the service, that is, the description of the service interface (typically in WSDL), the business protocol supported by the service (i.e., the order in which the interface methods can be invoked), and possibly other non-functional attributes. The external specifications are used at development time to write clients that can correctly interact with a service, and at run time by the middleware that supports service selection and interoperability. In some cases, external specifications are mandated by standardization consortia (such as RosettaNet), that define how services in a certain industry sector should behave. Such standardization is important as it simplifies interoperability and promotes competition (if many services support the same external specifications, it is technically easy for clients to switch between providers based on economical convenience). If ACME wants many customers to use its Web service, then it has to make its Web service as interoperable as possible. This implies the need of being compliant with a variety of different external specification requirements, provided by different standardization consortia or different customers. Hence, while the service functionality remains to a large extent the same, a service needs to adapt to different external specifications. As the number of services provided by ACME grows, and as the number of customers grows, managing adaptation quickly becomes a daunting effort.

In a second scenario, ACME runs (composite) services which are themselves implemented by invoking other services, possibly offered by third parties. It is not infrequent to have many ACME services invoke a same third-party service, e.g., a payment service offered by a financial institution. Hence, a version change of the external service would have a deep impact on the collection of composite services, possibly preventing some or all of them from performing their task. In this case, some or all of the composite services need to be adapted to interact with the new version of the invoked service. Again, as scale increases, so does the complexity of the adaptation management effort.

This paper presents a framework and a tool for managing service adaptation. We argue that, to simplify adaptation, 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 further 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. This is particularly evident in the second scenario above: if the invoked service changes the interface or protocol, then all the composite services invoking it will have to undergo analogous changes to interact with the new version of the invoked service. Consistently with this vision, we propose the use of an aspect-oriented programming (AOP) approach to weave adaptation solutions into the different composite services that need to be modified. To the best of our knowledge, this is the first work to identify service adaptation as a cross-cutting concern and to propose an aspect oriented approach to tackle it.

In a nutshell, the proposed approach works as follows. First, we provide a taxonomy of mismatches that can occur between two services. We specifically focus on service interfaces and protocols, the two most commonly used parts of external specifications. The reasoning behind having a taxonomy of mismatches is because we argue that similar mismatches can be addressed with similar modifications to the service implementation. Then, for each mismatch, we provide a template that embodies the AOP approach to adaptation. Specifically, the template contains a set of <pointcut, advice> pairs that define where the adaptation logic is to be applied, and what this logic is. As a very simple example, if the base service changed the signature of an operation, the pointcuts will be the activities in the composite services where the operation is invoked, and the adaptation logic consists in modifying the invoked message so that it can be made compatible with the new interface. In our approach, pointcuts are specified as queries over business process execution, that is, over the execution of composite services. In this paper, and in the tool we developed, we assume that services are implemented in BPEL, though the concepts are independent of the specific language adopted. The advices are therefore also specified in BPEL, as snippets that modify the behavior of the services at the specified pointcuts. In fact, since, as we will see, adaptation needs arise from the conjunction of a given service composition and a particular client interaction, weaving adaptation code at runtime is more suited than static weaving done at the code level as it allows for query expressed on particular execution contexts.

Finally, we present the development and runtime tools we have implemented to support aspect-oriented adaptation. All of the above ingredients of the solution correspond to contributions of this paper, with the exception of the identification of the mismatch taxonomy, which is part of our earlier work [3].

2 Service Mismatches

We illustrate an instance of service adaptation problem as it occurs in the first scenario discussed in the introduction through a supply chain example. Figure 1 shows a model of the interactions that take place in a supply chain process. This model is expressed using the Business Process Modeling Notation [15], which is a high-level equivalent of BPEL [11]. In this supply chain process, the client follows a standardized External Specification (ES) which specifies a protocol that allows the client to performs operations in one of the two sequences, namely S_1 (top part of the client flow) or S_2 (bottom part of the client flow).

In this example, the implementation of the Business Process (BP) (bottom part of the figure), differs from the target ES, and therefore is incompatible with the client, in several respects:

(a) Signature mismatch: The BP allows a client to order products through an operation named `OrderProduct` that requires an input parameter named `order` whose type is `ProductOrderInfo`. The ES specifies the same functionality via the `SubmitOrder` operation with the same input parameter but the data type is `OrderDetail`.

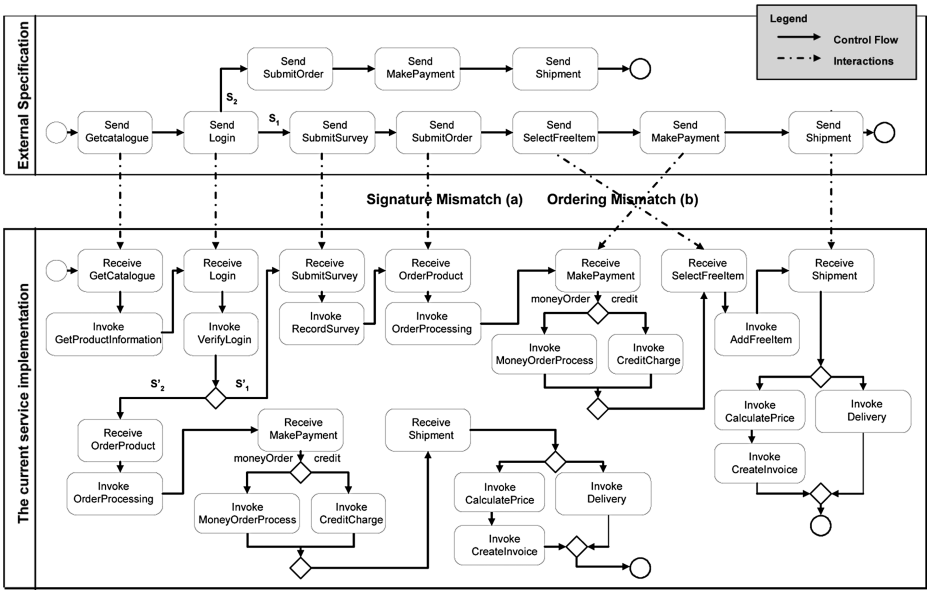

Fig. 1. A Supply Chain Example showing the Differences between an ES and a BP

(b) Ordering Mismatch: After an order has been submitted, the BP requires the client to send a `makePayment` message, while the ES specifies that the client has also the possibility of using `selectFreeItem`. This possibility can create an ordering mismatch when the client choses to follow the execution path S_2 of the ES.

To be able to interact with clients of this supply chain community, the external behavior of this BP has to be modified so that it complies with the community's ES. At the same time, since the BP might participate in some other companies' workflow and interact with various internal partners, we have to make sure that any modification will not prejudice those interactions.

Mismatch Types. In our previous work [3], we identified a taxonomy of possible mismatches at the interface and protocol levels. To make this paper self-contained, we briefly introduce the mismatch types, in addition to the signature and ordering mismatches above, as follows:

- Parameter constraint: Two services have different constraints on an input parameter, where the value range of the ES parameter is not a subset of the BP parameter, therefore values sent by the client are not accepted by the BP. For output parameters, mismatch occurs when value range of the BP parameter is not a subset of the ES parameter.
- Extra message: The BP issues a message that is not specified in the ES.
- Missing message: The BP does not issue a message specified in the ES.

– Message split: The ES specifies a single message to achieve a functionality, while the BP requires several messages for the same functionality.
– Message merge: The ES specifies several messages to achieve a functionality, while the BP requires only one message for the same functionality.

3 Aspect Oriented Service Adaptation

To address mismatches such as the ones mentioned above, we introduce adaptation templates. An example of template for the ordering mismatch is presented in Figure 2. In the following we detail this template structure: we first introduce joinpoint queries and discuss the alternatives and rationale for their design. We then present the advices and, finally, we show how the ordering template, as well as an other example of template corresponding to the signature mismatch, are applied to perform adaptation.

Ordering Template	
Query	**Generic Adaptation Advice**
`query`(<operation>,<sequence>) `executes` *before receive* `when` O_j^{bp} = <operation> `AND` S_i = <sequence>	OrderingPart1() { `Receive` $msgO_i^{bp}$; `Assign` $msgO_i^{tmp} \longleftarrow msgO_i^{bp}$; }
`query`(<operation>,<sequence>) `executes` *before receive* `when` O_i^{bp} = <operation> `AND` S_j = <sequence>	OrderingPart2() { `Assign` $msgO_i^{bp} \longleftarrow msgO_i^{tmp}$; `Reply` $msgO_i^{bp}$; }

Fig. 2. Template corresponding to the ordering mismatch

3.1 Joinpoints

The key part in the aspect-oriented approach to adaptation lies in understanding the requirements for the joinpoint query language. To this end, we first observe that the need of adaptation advice is determined not only by the BPEL code, but also by the actual messages received from the client, and in general by runtime service execution data. For instance, the ordering mismatch (b) of Figure 1 only happens when the interaction path follows sequence $S_2^{'}$. In this situation, it is the client choice of using one particular interaction pattern among the possible ones (i.e. sending `SubmitSurvey` after sending the `login` message) that triggers the adaptation need.

In general, aspect oriented programming can be done using various approaches for query language. A first approach consists in tailoring the query language for the identification, within the BPEL code, of locations where advices should be inserted. This limits the query expressiveness to conditions on the BPEL code only. As observed above, adaptation advice execution is also conditioned by runtime context, i.e. by how the service is actually used by a client or how it executes. Using a query language that focuses on the identification of code location would force us

to include, as part of the advice, some code to evaluate those runtime conditions. A second approach consists in directly expressing, in the query language, not only code location but also runtime conditions. This approach has been preferred since it groups together all advice execution conditions in the query and frees the advice code from any runtime condition evaluations. The net result is a more readable code and advices that are more generic.

Note that we are discussing here the query language syntax, not the actual deployment of the solution. Choosing a query language that incorporates runtime conditions still allows for aspect weaving done either at compile-time or at runtime. At compile-time, a new BPEL code would be generated with advices weaved preceded by runtime conditions. In a runtime deployment model, a specially modified query engine evaluates execution conditions based on the execution context it maintains, leaving the original code unmodified. While both deployment models are viable, the first one (compile-time) imposes to incorporate in the BPEL code some additional logic, not part of advices, that is needed to maintain execution context informations (e.g. the interaction pattern used by the client). In this paper, we therefore chose the second (runtime) deployment model which, in addition to its greater simplicity, also allows to dynamically plug and unplug adaptation aspects. The special runtime environment needed for this deployment model is presented in section 4.

Intuitively, we expect the query language to be able to perform i) identification of operations with (or without) a certain signature (this is to handle interface-level mismatches), and ii) identification of paths that are or are not present in a protocol, that is the query language must be able to discriminate between the various execution paths that lead to or follow this activity (this is to handle protocol-level mismatches). Due to space limitations we only present here the intuition rather than the detailed analysis which is based on the mismatch types discussed earlier. 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. In addition, we need the language to be able to define the location of the joinpoint, i.e. whether the advice is to be performed *before*, *after* or *around* (i.e. in place of) the BPEL activity.

In addition, the query needs to be able to handle runtime conditions, and to this end it can take parameters that are matched against execution context at the time of query evaluation. Parameters are given by the user and correspond to BPEL construct or operations sequences. For example, in the first query of Figure 2, parameters corresponding to the ordering mismatch above would be <operation>=`makePayment` and <sequence>=$S_2^{'}$. This query will be evaluated by the runtime environment before each receive activity, as indicated by the `executes` statement, and the two variables O_j^{bp} and S_i are valued according to the current operation and the sequence of operations that lead to the receive activity under consideration.

Figure 3 presents, in a semi-formal way, the syntax for a query specification language that satisfies the above requirements. This query language shares some

common characteristics with query languages that operate at the code level such as BPQL[2]. The main differences are that i) conditions on BP executions can be expressed and ii) the language also incorporates the location of the advice relative to the joinpoint (i.e. the *before*, *after* or *around* keywords). As explained above, those modifications are needed to achieve a self contained query language able to express all the conditions for advice execution. Examples of queries are given in Figure 2 and in Figure 5.

```
<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|inputParameter
                       |outputParameter|type|executionPath
```

Fig. 3. Semi-formal syntax for query language

Finally, we observe that mismatches of different types may occur at the same point in a process. In this case, many queries may need to be evaluated. We have prioritized the query evaluation based on the mismatch types. For example, signature mismatches need to be addressed before a message is stored and forwarded to the BP by an ordering template.

3.2 Advices

An advice corresponds to the code that is executed when its associated query conditions are satisfied. We call this code *generic* since it requires parameters that are specific to an adaptation situation. We choose BPEL as a language to express template advices for consistency with the original BPEL service implementation, although any other languages commonly used to implement web services could be a choice. Moreover, the activities required for adapting business processes, such as receiving messages, storing messages, transforming message data, and invoking service operations, are very well modeled by BPEL.

As an example, consider the ordering template (Figure 2). Figure 4 presents how this template behaves at runtime: upon receiving a message, the runtime environment triggers the execution of the OrderingPart1 if this message is not desired at this stage of the BP execution, i.e. if message `selectFreeItemIn` is received. When executed, the OrderingPart1 advice assigns the `selectFreeItemIn` into a temporary variable, i.e. `freeItemTmp` for later use. When the message `selectFreeItemIn` is required by the BP, the orderingPart2 advice copies its value from the `freeItemTmp` variable and forwards it to the BP. Note that for the sake of clarity, we omitted the acknowledgment of the `selectFreeItemIn` message in this

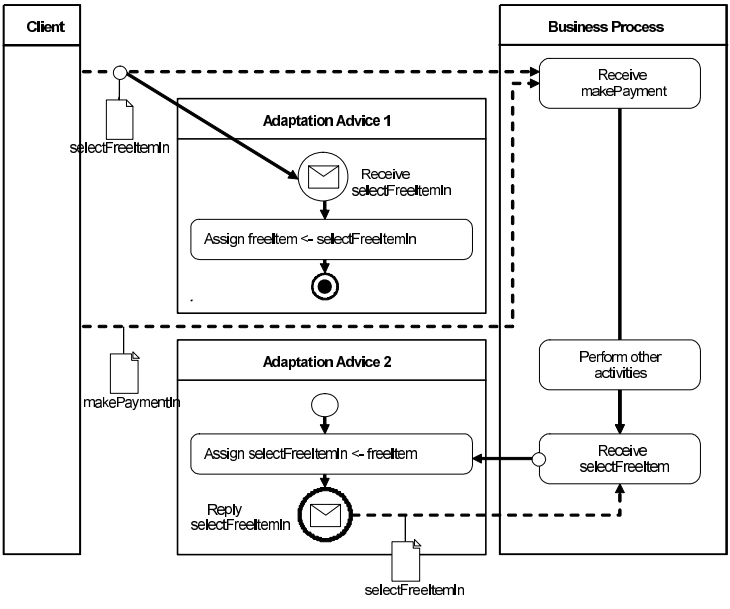

Fig. 4. Sample Usage of the Ordering Template

mismatch template. In a situation where the client requires an acknowledgment, the adaptation logic will be more complex. We refer the reader to [3] where this situation is discussed.

As another example, consider the template to address the signature mismatch, given in Figure 5. It also consists of two adaptation advices: SignaturePart1 and SignaturePart2. The SignaturePart1 first intercepts an incoming message $msgO^{es}$ of an operation O^{es} specified by the ES, then transforms the data type $type^{es}$ of a message parameter into $type^{bp}$ required by the BP, and finally sends the resulted message $msgO^{bp}$ to the BP. Similar actions are specified in the SignaturePart2 to solve mismatch for the outgoing messages of the BP.

Due to space limitations, it is not possible here to present the full set of templates, but the general method described above can easily be applied in the other mismatch situations.

4 Template Usage and Tool Support

The generic mismatch handling procedures encapsulated in templates allows for i) generation of the adaptation logic, and ii) integration of the generated adaptation logic into the business process. All the developer has left to do is to identify the mismatches and instantiate their corresponding templates. For example, once a signature mismatch has been identified between two services, the user retrieves the corresponding template and provides parameters for the queries and advices of SignturePart1 and SignaturePart2. Both queries of this template take a data

Signature Template	
Query	**Generic Adaptation Advice**
query(<inputType>) executes *before receive* when $type^{bp}$= <inputType>	SignaturePart1(<T_i>) { Receive $msgO^{es}$; Assign $msgO^{bp}.inPara^{bp}.type^{bp}$ ⟵ <T_i>($msgO^{es}.inPara^{es}.type^{es}$); Reply $msgO^{bp}$; }
query(<outputType>) executes *before reply* when $type^{bp}$ = <outputType>	SignaturePart2(<T_o>) { Receive $msgO^{bp}$; Assign $msgO^{es}.outPara^{es}.type^{es}$ ⟵ <T_o>($msgO^{bp}.outPara^{bp}.type^{bp}$); Reply $msgO^{es}$; }

Fig. 5. Template corresponding to the signature mismatch

type as input and express that their corresponding advices should be executed for each receive (resp. reply) of the BP that involves a message parameter of that type. In addition, SignaturePart1 and Signaturepart2 advices each takes a *Transformation Function* (denoted $< T_i >$ and $< T_o >$) that is responsible of actually transforming the data types of the message parameters. One of the benefits of using those precise templates is that the developer's task is limited to the identification of the mismatch (i.e. checking the compatibility of the data models as used in the BP and as specified by the ES) and, when those types do not correspond, to write the mapping between them. For mapping authoring, third party tools (e.g. Biztalk) already provide efficient schema matching functionality. In our implementation, we used XQuery[4] functions to perform those transformations, though other languages can be used.

The developer is assisted in this task by a tool that we have developed. The tool consists of a development and runtime environment.

Development Environment. The development environment assists the developer in instantiating the adaptation templates. To this end, the user has to provide the parameters for queries and advices. As discussed in section 3.1, the query parameters correspond to BPEL construct identifiers (i.e. their names as found in the BPEL source). The user has to look through the process specification which could be large in its size. We intend to provide a query support that allows the user to query over process specifications and give parameters to template queries. On the other hand, advice parameters are transformation functions that can be authored using third party softwares.

Once both the query and advice parameters are provided, the Development Environment generates two outputs: the Aspect Definition Document and a collection of adaptation advices. An example of the Aspect Definition Document is shown below. It is an XML file that consists of a set of mismatch elements, each specifying a template and its corresponding parameters. Used together, the Aspect Definition Document and Adaptation Advices (template instances with

their query and advice parameters) allow the Runtime Environment, discussed in the next section, to adapt the BP in compliance with the ES.

```
<aspect>
  <mismatch template="Signature">
    <advice name="SignaturePart1" location="before" activity="receive">
      <queryParameter name="inputType" value="ProductOrderInfo"/>
      <adviceParameter name="Ti" value"TransformProductOrderInfo"/> </advice>
    <advice name="SignaturePart2" location="before" activity="reply">
      <queryParameter name="outputType" value"OrderConfirmation"/>
      <adviceParameter name="To" value"TransformOrderConfirmation"/> </advice>
  </mismatch>  ... </aspect>
```

Runtime Environment. The Runtime is implemented on top of the ActiveBPEL engine [1], and enables the dynamic weaving of adaptation advices with the business process. Similar extensions can be considered for other types of business process implementation (e.g. J2EE). During process execution, the runtime environment uses query information in the Aspect Definition Document to identify if an adaptation advice needs to be executed, based on the current execution context. If it is the case, the adaptation advice, which is also specified in the Aspect Definition document, is loaded and executed according to its definition. After the completion of the adaptation advice execution, ActiveBPEL continues to process the BPEL instance. Interestingly, this extension has itself been implemented using an aspect weaved with the ActiveBPEL code using AspectJ.

The runtime environment supports the inclusion of multiple adaptations for the same BP. To make this possible, each adaptation aspect is associated with a specific virtual URL. When the BP is first invoked by a client, the URL is used to determine which adaptation aspect (i.e. which Aspect Definition document) should be used. Hence, the same business process can be adapted to different ESs.

5 Related Work

In the software engineering area, few approaches exist for analyzing and solving software component mismatches. [9] proposes an algorithm to identify mismatches between different versions of architectural models and generates an edit script to solve those mismatches. This approach can be extended to identify mismatches between business protocols. Several efforts recognize the importance of protocol specification in component-based models [5,12,17]. They provide models for component interface specifications (based on formal approaches e.g. process algebra) and algorithms (e.g. compatibility checking) that can be used for web service protocol specification and analysis.

In the context of web services, [13] proposed a technique called chain of adapters, that satisfies their identified requirements, to manage different versions of services. In our previous work [3], we argued that mismatches between service interfaces and protocols are recurring, hence we complemented the adapter approach by providing a taxonomy of mismatches. [8] also supports this argument and provides visual operators for adaptations. However, in our previous work, we made no contribution on the actual implementation of the adaptation logic

and how to integrate them with the service implementation. In this paper, we design, for each mismatch, an aspect based template that consists of a collection of adaptation logic expressed in BPEL, and query to support weaving of the adaptation logic with the business process.

A large amount of work has been done in the area of AOP, however they mostly address non-functional concerns of software [10]. We focus on previous work that applies AOP to the adaptation problem. [14] proposes an aspect oriented platform to support adaptation of services according to changes in the environment, while we focus on the adaptation of processes to be compatible with external specifications. In addition, their focus on services implemented in Java prompts them to identify joinpoints on methods and field accesses rather than BPEL activities as in our framework. Work of [6,7] also supports aspect oriented adaptation of BPEL processes according to changes in the environment. They use XPath to identify pointcuts which restrict to queries on individual process execution events. Our query on execution paths differentiate our framework from this previous work. Another work that also focuses on adaptation for compatibility is presented in [16]. They propose a framework for transforming XML messages where pointcuts are defined on document contents using XPath. We differ from this work in terms of the mismatch taxonomy and our focus on business protocol mismatches.

6 Conclusion

In this paper we proposed the use of AOP for service adaptation to interface and protocol mismatches. We have argued for adaptation as a cross-cutting concern and for the separation of business and adaptation logic. This modularization facilitates the maintenance of the BP when the target ES evolves since only the adaptation logic needs to be changed. A further benefit of our framework consists in the precise input parameters required from the user that can be built from third party tools or supported by a graphical interface. The notion of template also promotes reusability of adaptation logic that occurs repetitively across different locations in an implementation of a service. In this paper, we exemplified the application of our framework into the first scenario discussed in the introduction where adaptation needs to be performed at the level of individual web services. The benefits become even greater when considering situations where adaptation needs to be performed across several composite services since, in this case, considering adaptation as a cross cutting concern becomes critical.

We have developed a proof-of-concept implementation of the proposed framework. In particular, we have implemented the Runtime Environment that takes an Aspect Definition Document and Adaptation Advices as inputs to adapt a business process in accordance to an external specification. Our experience with the framework has been primarily example driven. For the Development Environment, we have provided a GUI support for the instantiation of adaptation advices.

In the current framework, users have to look at the protocol definition and the BP model to identify the mismatches and provide query parameters. When

the model grows large, this task can become significant. In the future, we plan to extend the Development Environment to offer a semi-automated identification of mismatches and a graphical interface that allows the user to create queries over process specifications and navigate through the results in order to identify query parameters.

References

1. ActiveBPEL Engine 2.0. http://www.activebpel.org/.
2. C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying Business Processes with BP-QL. In *VLDB'05*.
3. B. Benatallah, F. Casati, D. Grigori, H.R. Motahari Nezhad, and Farouk Toumani. Developing Adapters for Web Services Integration. In *CAISE'05*, pages 415–429.
4. S. Boag, D. Chamberlin, M. Fernández, D. Florescu, J. Robie, and J. Siméon. XML Query Language (XQuery 1.0), November 2005. http://www.w3.org/TR/xquery/.
5. A. Bracciali, A. Brogi, and C. Canal. A Formal Approach to Component Adaptation. *Journal of System and Software*, 74(1):45–54, 2005.
6. A. Charfi and M. Mezini. Aspect-Oriented Web Service Composition with AO4BPEL. In *ECOWS'04*, pages 168–182.
7. C. Courbis and A. Finkelstein. Towards Aspect Weaving Applications. In *ICSE'05*, pages 69–77.
8. M. Dumas, M.Spork, and K.Wang. Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation. In *accepted to BPM'06*.
9. M. Abi-Antoun et.al. Differencing and Merging of Architectural Views. Technical report, Carnegie Mellon University, CMU-ISRI-05-128R, August 2005.
10. N. Loughran et.al. Survey of aspect-oriented middleware research. Technical report, Lancaster University, June 2005.
11. T. Andrews et.al. Business Process Execution Language for Web Services 1.1. Technical Report TUV-1841-2004-16, BEA, IBM, Microsoft, SAP, Siebel, 2003.
12. P. Inverardi and M. Tivoli. Deadlock-free software architectures for COM/DCOM applications. *Journal of Systems and Software*, 65(3):173–183, 2003.
13. P. Kaminski, H. Muller, and M. Litoiu. A design for adaptive web service evolution. In *SEAMS'06*, pages 86–92.
14. A. Nicoara and G. Alonso. Dynamic AOP with PROSE. In *CAISE'05*, pages 125–138.
15. Stephen A. White. Business Process Modeling Notation (BPMN 1.0), May 2004. http://www.bpmn.org.
16. E. Wohlstadter and K. Volder. Doxpects: aspects supporting XML transformation interfaces. In *AOSD'06*, pages 99–108.
17. D. Yellin and R. Strom. Protocol Specifications and Component Adaptors. *ACM TOPLAS*, 19(2):292–333, 1997.