Adaptation of Web Service Interactions Using Complex Event Processing Patterns

Yéhia Taher, Michael Parkin, Mike P. Papazoglou, and Willem-Jan van den Heuvel

European Research Institute for Service Science, Tilburg University, The Netherlands
{y.taher,m.s.parkin,mikep,wjheuvel}@uvt.nl

Abstract. Differences in Web Service interfaces can be classified as signature or protocol incompatibilities, and techniques exist to resolve one or the other of these issues but rarely both. This paper describes *complex event processing* approach to resolving both signature and protocol incompatibilities existing between Web Service interfaces. The solution uses a small set of operators that can be applied to incoming messages individually or in combination to modify the structure, type and number of messages sent to the destination. The paper describes how CEP-based adapters, deployable in CEP engines, can be generated from automata representations of the operators through a standard process and presents a proof-of-concept implementation.

1 Introduction

Web services allow the integration of distributed software through standard interface definition languages, transport mechanisms and aspects such as security and quality of service. Web Service interfaces (i.e., WSDL, BPEL, etc.) define the messages and protocol that should be used to communicate with the service [7]. However, if two services wish to interact successfully, they must both support the same messages and protocol through the implementation of compatible WSDL and/or BPEL documents. Unfortunately, this is difficult to achieve in practice; Web Services are often developed independently and follow different standards or approaches in constructing their interfaces and Web Service compositions will often use them in ways that were not foreseen in their original design and construction [3, 2]. Therefore, it is likely that most Web Services will be incompatible as services will not support the same interface.

This is a short paper describing a general approach to resolving differences between Web Services protocols through the use of Complex Event Processing (CEP [4]) technology. Specifically, we extend our previous work [10, 11] to show how a small set of general operators can be used to match the messages from one service with those of another. By using a *continuous query engine* running within a CEP platform, we demonstrate signature and protocol adaptation between Web Services in a proof-of-concept implementation.

This paper is structured as follows: Section 2 describes our CEP-based approach to signature and protocol adaptation; Section 3 introduces the operators used to resolve differences in Web Service protocols; Section 4 presents the

G. Kappel, Z. Maamar, H.R. Motahari-Nezhad (Eds.): ICSOC 2011, LNCS 7084, pp. 601–609, 2011.

CEP solution and a proof-of-concept implementation; Section 5 compares related work; Section 6 contains conclusions and our plans for future work.

2 Approach

Incompatibilities between Web Service protocols can be classified as either [2,3]:

1. Signature Incompatibilities arise due to the differences between services in expected message structure, content and semantics. In Web Services, XML schema provides defines a set of 'built-in' types to allow the construction of complex input and output message types from these primitives. This flexibility in constructing message types in XML often means that a message from one Web Service will not be recognized by another and, therefore, there is a requirement to provide some function that maps the schema of one message to another [6].

2. Protocol Incompatibilities are found when Web Services wish to interact but are incompatible because they support of different message exchange sequences. For example, if two services perform the same function, e.g., accept purchase orders, but Service A requires a single order containing one or more items while Service B expects an order message for each item, there is a mismatch in their communication protocols that must be resolved in order for them to interoperate. To solve these incompatibilities, there are two approaches: a) to force one of the parties to support the other's interface, or b) to build an adapter that receives messages, converts them to the correct sequence and/or maps them into a desired format and sends them to their destination. However, both of these solutions are unsatisfactory; imposing the development of an interface for each target service can lead to an organization having to maintain a different client for each service it uses, and the implementation of bespoke ad-hoc point-to-point adapters is costly and not-scalable.

Our solution is to automate the generation of adapters so the process is repeatable and scalable and remove the necessity to build costly bespoke adapters. Our approach for generating adapters is described in [9], which presents an algorithm for detecting signature and protocol incompatibilities between two Web Service protocol descriptions (i.e., interfaces) and a CEP-based mediation framework to perform protocol adaptation practically. This paper completes the mediation framework by showing how the incompatibilities found between two Web Service protocols, classified according to a set of basic transformation patterns by the algorithm in [9], can be transformed into configurable automata *operators* which are used to generate *adapters*.[1] In Section 3 we describe the operators required for each transformation pattern then in Section 4 show how they are used to generate CEP adapters and deployed to a CEP engine.

Complex Event Processing technology can discover relationships between events through the analysis and correlation of multiple events and triggers and

[1] Adapters are therefore the components that resolve sets of incompatibilities found between two services and are aggregations of predefined *operators* who's purpose is to resolve individual, specific incompatibilities.

take actions (e.g., generate new events) from these observations. CEP does this by, for example, modeling event hierarchies, detecting causality, membership and/or timing relationships between events and abstracting event-driven processes into higher-level concepts [4]. CEP platforms allow streams of data to run through them to detect conditions that match the *continuous computational queries* (CCQs, written in a Continuous Computation language, or CCL) as they occur. As a result, CEP has an advantage in performance and capacity compared to traditional approaches: CEP platforms typically handle many more events than databases and can process throughputs of between 1,000 to 100k messages per second with low latency. These features make a CEP platform an excellent foundation for situations that have real-time business implications.

In the context of Web Services, *events* occur when SOAP messages are sent and received. Therefore, CEP adaptation requires the platform to consume incoming messages, process them and send the result to its destination. However, a CCQ written for a particular adaptation problem is similar to the bespoke adapter solution described earlier. To offer a universal solution and a scalable method for Web Service protocol adaptation, we automate the generation and deployment of CCQs to transform incoming message(s) into the required output message format(s) using the predefined set of transformation operators.

3 Operators

[3] describes five basic transformation patterns that can reconcile protocol mismatches. We have developed an operator for each of these patterns that can be applied individually or in combination to incoming messages to achieve a transformation in both the structure, type and number of messages sent to the destination — i.e., to resolve both signature and protocol incompatibilities.

The operators developed for each of the transformation patterns are: *Matchmake*, which translates one message type to another, solving the *one-to-one* transformation; *Split*, a solution for the *one-to-many* pattern, which separates one message sent by the source into two or more messages to be received separately; the *Merge* operator is the opposite of the *Split* operator (i.e., it performs a *many-to-one* transformation) and combines two or more messages into a single message; the *Aggregate* operator is used when two or more of the same message from the source service interface correspond to one message at the target service and is a solution for the *one*$^{+}$*-to-one* transformation; finally, *Disaggregate* performs the opposite function to *Aggregate* operator.

Following [9], the operators are represented as *configurable automata*. Transitions between states represent both observable and non-observable actions. *Observable actions* describe the behavior of the operator vis-à-vis the service consumer and provider, i.e., an action is observable if it is a message consumption or transmission event. *Unobservable actions* describe the internal transitions of the operator, such as the transformation of a messages contents, and are performed transparently to the source and target services.

Transitions caused by observable actions are denoted as $<a,?/!m,a'>$, where a is the starting state and a' the end state following the consumption *(?)* or

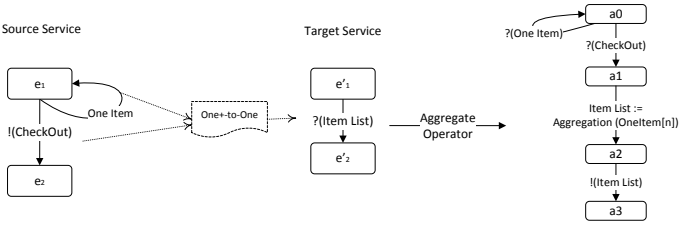

Fig. 1. The Aggregate Operator

transmission *(!)* of message *m*. An unobservable action is denoted as $<a, \psi, a'>$, where *a* and *a'* are the start and end states following internal action ψ.

For reasons of space it is not possible to describe all five operators in detail and we have chosen the *Aggregate* operator to illustrate how they work. Figure 1 shows the operator to resolve *one*$^+$*-to-one* incompatibilities between services, e.g., when a customer submits a purchase order for each item but the retailer expects a list of all items together. To resolve this incompatibility the aggregate operator consumes and stores *?OneItem* messages until it receives the message *(?CheckOut)* indicating all messages have been sent. The operator aggregates the stored messages into a list of items message using *ItemList=Aggregation(OneItem[n])* and forwards the new message using *!(ItemList)*.

4 CEP-Based Adaptation

4.1 General Principles

The adaptation of interactions between source and target services is specified using automata, therefore deploying them as CEP adapters requires their translation into continuous queries. To do this, we modeled message consumption and transmission actions as events. For each message type consumed or transmitted we create an input or output stream. A continuous query subscribes to the input stream of messages it wants to adapt and publishes the adapted message(s) to the corresponding output stream(s). For convenience, we name the input/output stream the same name as the message it consumes or transmits. This method allows a CEP engine to intercept messages exchanged between two services, to detect patterns of incompatibilities and implement corresponding adaptation solution, i.e., combinations of the operators encoded as continuous queries.

4.2 Conceptual Architecture

Figure 2 illustrates the conceptual architecture of the CEP implementation that translates the adapter specified in automata into continuous queries, i.e., via `Automata` → `Continuous Queries`. This includes the creation of input and output streams for the continuously running queries in the CEP engine, waiting for messages arriving through input streams.

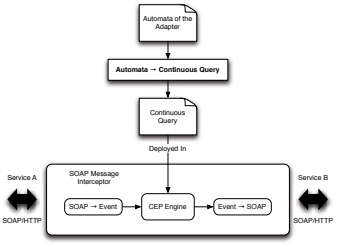

Fig. 2. Conceptual CEP Adaptation Architecture

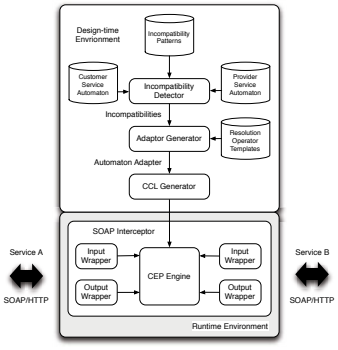

Fig. 3. Architectural Framework for the CEP Solution

In the second step, the `SOAP Message Interceptor`'s role is to control the exchange of messages between the two services. Upon receipt of a message, the Interceptor sends it to the input stream with the same name (through `SOAP` → `Event`). The message received is published as an event and is consumed by the query that subscribes to the input stream. The message(s) produced as a result of applying the operators is published to the corresponding output stream. Once on the output stream, the message is consumed by the SOAP message interceptor (through `Event` → `SOAP`) and sent to the target.

Figure 4 shows the transformation of the *Aggregation* automata to a continuous query. First, an input stream and a window to store messages arriving on the input stream are created for action *?(m1)* and an input stream is created for action *?(m2)*. The aggregation query is then specified: it subscribes to the window where messages from action *?(m1)* are stored and to the input stream for action *!(m2)* actions. After an *!(m2)* action, messages in the window with the same correlation criteria as new message are aggregated into a single message, the input messages are removed and the result is published to the output stream.

Figure 5 shows a concrete example where messages arriving through the input stream *Order_In* are stored in the window *Order_Win*. When the message arrives through *CheckOut_In* indicating order number #03203 is complete, messages in *Order_Win* with the same order number (#03203, the correlation criteria) are aggregated into a single message. The final message, containing *Item1* and *Item2*, is published to *Order_Out*.

4.3 Proof of Concept

This section illustrates the practical generation and deployment of CEP-based adapters using *model transformation*. It has two stages: the *design phase* models the adapter using operator automata through the use of an incompatibility detection process to produce a platform independent model, whilst the *transformation phase* takes the platform independent model to produce the adapter as a CCQ for a CEP engine, i.e, a platform specific model.

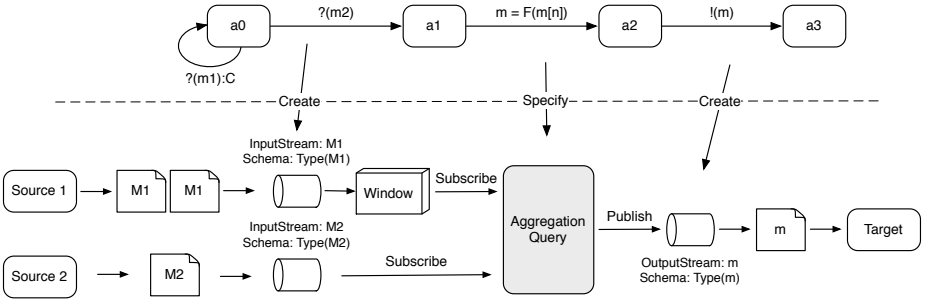

Fig. 4. Aggregate Translation

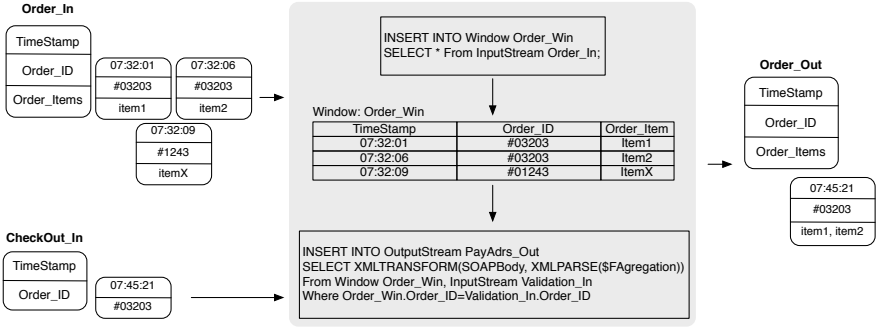

Fig. 5. Aggregate Flow

Figure 3 shows the framework for the automatic generation of adapters. If two incompatible services, A and B, wish to communicate, at design-time an adaptor can be generated for the runtime CEP engine by classifying the incompatibilities between two service interfaces (using the method described in [9]) and using them to construct the adapter using *resolution operator templates* (described in Section 3). The resulting adapter is converted into the CEP engine's continuous computation language (CCL) that is deployed at run-time within a SOAP message interceptor to provide a message serialization/deserialization capability.

The **Design Time Environment** is used to instantiate the template operators described in Section 3 so they can be used in a specific Web Service protocol adaptation. The design-time tools can be used to develop strategies for dealing with complex adaptation situations by allowing the composition of the template operators. In these cases, the designers of the adaptation must identify what adaptations are required between two services and use a graphical user interface (the *Design Tool* shown) to wrap the corresponding composition of operators in a map. Maps are exported to the CCQ code generation tool that includes a compiler to produce a CEP execution-time module (i.e., the CCQ) which is then loaded into the CEP execution engine.

The **Run Time Environment** contains a CEP platform with a continuous query engine and a set of SOAP message integration layers to allow it to send and receive messages to and from Web Services. The continuous query engine provides the capability for the system to receive, process, correlate and analyze SOAP messages against a CCQ. However, since Web services communicate through the use of SOAP messages, intermediate adapters are required to provide entry and exit points to the engine. These intermediate adapters are of two types: input and output wrappers. An input wrapper receives SOAP messages from the source's service interface and transforms it to the representation appropriate for the CEP engine and then sends it to the engine. Similarly, an output wrapper receives events produced by the engine and transforms it to a SOAP message before forwarding the message to the target service.

4.4 Demonstration and Experimentation

A demonstration of our prototype can be seen at: `http://www.youtube.com/watch?v=g05ciEPZ_Zc`.

5 Related Work

As [3] describes, there are many commercial tools to achieve Web Service *signature mediation* and solve signature incompatibilities, including: Microsoft's Biztalk mapper[2], Stylus Studio's XML Mapping tools[3], SAP's Exchange Infrastructure (XI) Mapping Editor[4] and Altova's MapForce[5]. Academic research also exists in resolving signature incompatibilities through the use of semantic web technology (i.e., OWL), such as that described in [5] that presents a "context-based mediation approach to [...] the semantic heterogeneities between composed Web services", and the Web Service Modeling Ontology (WSMO) specification [8] that provides a foundation for common descriptions of Web Service behavior and operations. This research does not attempt to resolve the associated problem of protocol incompatibility, however.

Active research is also being performed into the adaptation of web service protocols, although all work we have surveyed does not tackle both problems of signature and protocol incompatibility and all use different approaches to the CEP-based technique presented. For example, although [3] presents mediation patterns together with corresponding BPEL templates, a technique and engineering approach for semi-automatically identifying and resolving identifying protocol mismatches and a prototype implementation (the *Service Mediation Toolkit*), it does not solve the signature adaptation problem. Similarly, [2] "discusses the notion of *protocol compatibility* between Web Services" and [1] again only "focusses on the protocol mismatches, leaving data mismatches apart" —

[2] `http://www.microsoft.com/biztalk/en/us/default.aspx`
[3] `http://www.stylusstudio.com/xml_mapper.html`
[4] `http://www.sdn.sap.com/irj/sdn/nw-xi`
[5] `http://www.altova.com/mapforce/web-services-mapping.html`

i.e., they present solutions to protocol mismatches and do not tackle the associated problem of signature incompatibility. Our chosen approach solves both signature and protocol incompatibilities.

6 Conclusion

Web service incompatibilities are found in either their message signatures or protocols. This paper presents an CEP approach to adapt Web Service interactions and resolve these conflicts. Using predefined operators represented as configurable automata allows us to automatically CEP generate adapters capable of intercepting incoming messages sent between services and adapting their structure, type and number into the desired output message(s). Our future work will be in two areas: (i) performing extensive testing on real services, and (ii) developing tools to assist service designers to generate adapters.

Acknowledgment. The research leading to these results has received funding from the European Community's Seventh Framework Program [FP7/2007–2013] under grant agreement 215482 (S-CUBE). We thank Marie-Christine Fauvet, Djamal Benslimane and Marlon Dumas for their comments and contributions on earlier stages of this work.

References

1. Ardissono, L., Furnari, R., Petrone, G., Segnan, M.: Interaction Protocol Mediation in Web Service Composition. International Journal of Web Engineering and Technology 6(1), 4–32 (2010)
2. Dumas, M., Benatallah, B., Nezhad, H.R.M.: Web Service Protocols: Compatibility and Adaptation. IEEE Data Engineering Bulletin 31, 40–44 (2008)
3. Li, X., Fan, Y., Madnick, S., Sheng, Q.Z.: A Pattern-Based Approach to Protocol Mediation for Web Services Composition. Information & Software Technology 52(3), 304–323 (2010)
4. Luckham, D.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley Longman (2001)
5. Mrissa, M., Ghedira, C., Benslimane, D., Maamar, Z., Rosenberg, F., Dustdar, S.: A Context-Based Mediation Approach to Compose Semantic Web Services. ACM Transactions on Internet Technology (TOIT) 8(1), 1–23 (2008)
6. Nezhad, H.R.M., Benatallah, B., Martens, A., Curbera, F.: Semi-Automated Adaptation of Service Interactions. In: Proceedings of the 16th International Conference on World Wide Web, pp. 993–1002 (2007)
7. Papazoglou, M.: Web Services: Principles & Technology. Pearson Education (2008)
8. Roman, D., Keller, U., Lausen, H., de Bruijn, J., Lara, R., Stollberg, M., Polleres, A., Feier, C., Bussler, C., Fensel, D.: Web Service Modeling Ontology. Applied Ontology 1(1), 77–106 (2005)
9. Taher, Y., Aït-Bachir, A., Fauvet, M.C., Benslimane, D.: Diagnosing Incompatibilities in Web Service Interactions for Automatic Generation of Adapters. In: Proceedings of the 23rd International Conference on Advanced Information Networking and Applications (AINA 2009), pp. 652–659 (2009)

10. Taher, Y., Marie-Christine, F., Dumas, M., Benslimane, D.: Using CEP TEchnology to Adapt Messages Exchanged by Web Services. In: Proceedings of the 17th International Conference on the World Wide Web (WWW 2008), Beijing, China, pp. 1231–1232 (April 2008)
11. Taher, Y., Nguyen, D.K., van den Heuvel, W.J., Ait-Bachir, A.: Enabling Interoperability for SOA-Based SaaS Applications Using Continuous Computational Language. In: Proceedings of the 3rd European ServiceWave Conference, Ghent, Belgium, pp. 222–224 (December 2010)