

Service Mosaic

A Model-Driven Framework for Web Services Life-Cycle Management

Although Web services provide abstractions for simplifying integration at lower levels of the interaction stacks, they don't yet help simplify integration at higher abstraction levels such as business-level interaction protocols. Using a model-driven framework for Web services life-cycle management, the authors help facilitate the scalable development and maintenance of service-oriented applications by analyzing and managing Web service business protocols. Instead of using simple black and white measures, they identify different classes of protocol compatibility and replaceability. They implemented this framework in a prototype platform called Service Mosaic.

Web services are becoming the technology of choice for application integration. The main benefits are support for loosely coupled and decentralized interactions and standardization, which helps reduce the costs of application integration. To a large extent, these costs result from interacting entities having different interfaces, speaking different communication protocols, and supporting different data formats and interaction models.

Although Web services provide abstractions to simplify integration at lower levels of the interaction stacks (such as data syntax and communication protocols),^{1,2} where researchers have already identified and even solved many issues, they don't yet help simplify integration at higher abstraction levels (such as data or message types and business-level interaction protocols). Having loosely coupled

interactions implies that services aren't designed to be interoperable with a particular client (as traditional application integration often assumes). At development time, designers might not even know the type and number of clients that will access the service. Even if services speak the same low-level protocols, they're likely to have differences, for example, in terms of message types and allowed message-exchange patterns. Unless everyone agrees to adopt the same standards, most interactions will require one or both parties to perform some adaptation to enable a successful interoperation.

Our research looks at how to solve these and other pressing issues that affect the Web services development life cycle, facilitating the scalable development and maintenance of service-oriented applications. After years of research and development work in this area, we developed

**Boualem Benatallah
and Hamid Reza
Motahari Nezhad**

University of New South Wales

Fabio Casati

Hewlett-Packard Labs

**Farouk Toumani
and Julien Ponge**

University Blaise Pascal, France

a model-driven framework, called Service Mosaic, for Web services life-cycle management. Here, we overview the integrated framework and its implementation architecture. (See previous papers for details about the protocol model,³ protocol management operators,⁴ protocols adaptation,⁵ and code skeleton generation from protocol models.⁶)

Basic Principles and Building Blocks

Interoperability solutions must tackle several different abstraction levels. Broadly speaking, we've identified the following service interoperability layers:

- *Messaging.* Services should support messaging protocols' connectivity, regardless of the information's syntax and semantics. In Web services, the most common protocol at this level is SOAP. A service provider might further require (or allow) messaging to have certain properties. For example, the interaction might need to be reliable and secure.²
- *Basic coordination.* This layer is concerned with requirements and properties related to a set of message exchanges among two or more partners. For instance, two or more services might need to coordinate to provide atomicity based on the two-phase commit protocol. WS-Transaction is an example of specification at this level.
- *Business-level interfaces and protocols.* The previous layers are concerned with transferring messages among services, possibly endowed with properties such as security and reliability. Interoperation of services also requires compatible interfaces (that is, the set of operations the services support) and business protocols (or constraints on the order in which operations should be invoked to achieve a successful interaction). Besides letting developers create clients that can correctly interact with a service by stating the allowed *conversations* (or set of message exchanges), protocol specifications have other important applications that can simplify Web services life-cycle management, such as providing automated support for discovery, development time analysis, evolution, exception handling, and code generation.
- *Policies and nonfunctional aspects.* The definition of a service might include policies (such as privacy policies) and other nonfunctional aspects (such as quality-of-service descriptions) that are useful for clients to understand if they can or want to interact with the service. There-

fore, we must consider them when looking at Web services interoperability.

This article focuses on the business interface and protocol layer. Incidentally, although we discuss business protocol aspects here, we could use analogous techniques for other service aspects characterized by protocols (such as basic coordination).

We believe the evolution of work in Web services interoperability mirrors, at least conceptually, the work done in databases over the past 30 years that has led to generic abstractions and techniques (such as data models, relational algebras, theoretical foundations, and declarative query and transformation techniques) for simplifying the design of complex applications and enabling high-level data manipulation. We believe that Web service protocols require similar building blocks in terms of simple and useful models and support for high-level analysis, manipulation, and transformation. This philosophy inspires our work. More precisely, we consider the following aspects:

- *Conceptual protocol modeling.* Users should have at their disposal protocol models that are easy to understand and use. The key problem here is right-sizing the model – that is, including aspects that are frequently needed but avoiding overloading the model with features that are rarely necessary, making it complex and less likely to be used. Another important aspect is that the protocol modeling language should be formal enough to allow automated analysis and manipulation.
- *An algebra for analyzing and managing protocols.* Defining a protocol algebra and operators to query, analyze, and manipulate protocols allows the assessment of compatibility among services, the understanding of similarities and differences, and the composition and evolution of services. Other benefits involve supporting developers in verifying (statically or dynamically) whether the service implementation is consistent with its specifications and in verifying consistency between different specifications (at different levels of abstractions) for the same service.
- *Model-driven development of protocol adapters.* The need for adapters in Web services comes from the potentially high number of interacting services, each of which can support different business interfaces and protocols. This creates the need for providing multiple

faces of the same service for interacting with different partners (it's unrealistic to hope that all services will adopt the same standard specifications at all levels of the interaction stack). Developing adapters using ad hoc and low-level programming techniques is hardly applicable, especially in dynamic environments. Hence, it's desirable to provide a methodology and automated support to understand the differences and (partially) generate the adaptation logic.

The next sections discuss protocol modeling, analysis, and adaptation in more detail.

High-Level Analysis and Management of Protocols

In previous research,⁴ we developed a protocol algebra and protocol management operators targeted at three main types of analysis, which we believe are essential to Web service analysis and management:

- *compatibility*, assessing if two services can interoperate correctly;
- *replaceability*, verifying whether two different protocols can support the same set of conversations; and
- *consistency*, verifying whether a service's implementation can support the declared protocol definition.

Here we focus on protocol modeling and discuss compatibility and replaceability analysis. Although the formalization for verifying each of these properties depends on the protocol model adopted, the concepts apply to any protocol modeling language.

Modeling Business Protocols

Several languages exist for describing Web service protocols, such as the Business Process Execution Language (WS-BPEL) or the Choreography Description Language (WS-CDL). These languages are concerned more with implementation aspects than specifying protocol properties. They aren't suitable for automating activities such as protocol compatibility and compliance analysis. Our framework features a simple, high-level but expressive model to represent features and abstractions that are useful and needed in practice.³ We derived our model's main features from analyzing real e-commerce portals.

Message choreography. Protocols are modeled by state machines. States represent the different phases that a service might go through during its interaction with a requestor. Transitions are triggered by messages the requestor sends to the provider or vice versa. A message corresponds to a service operation invocation or to its reply. Hence, each state identifies a set of outgoing transitions and, therefore, a set of possible messages that can be sent or received. We chose state machines as the base formalism because they're a well-established model for describing reactive components and because of their simplicity, which we believe to be an essential characteristic for successful models. In addition, they're sufficient (with few extensions) for modeling all the behaviors frequently needed in practice.⁷

Transactional implications and effects. A service can include operations that semantically cancel the effects of other operations; some operations, for example, cancel a book purchase or flight reservation. In addition, for some operations to execute, they must acquire resources for the client. For instance, flight reservation services let customers hold seats on a plane.

To account for modeling transactional implications and effects, we distinguish between several types of transitions of a protocol state machine. *Effectless transitions* have no effect from the client's perspective. *Compensatable transitions* denote transitions with effects we can cancel. *Definite transitions* denote transitions with permanent effects. *Resource-locking transitions* lock certain resources for the requester for a time (useful for operations such as seat reservations).

Time-sensitive conversations. In some cases, state transitions can occur without an explicit invocation by requesters, typically (but not only) to model constraints in which requesters can only invoke an operation within a certain time window. We refer to these transitions as *implicit transitions*.³ A protocol might need to specify that a purchase order message is accepted, for example, only if it's received within 24 hours after a *quotation* has been made. We can specify this behavior by tagging transitions with a time interval – that is, the transition is fired as the interval expires, leading the state machine to a new state from which previously invoked operations are no longer enabled.

Details about this extended protocol model and

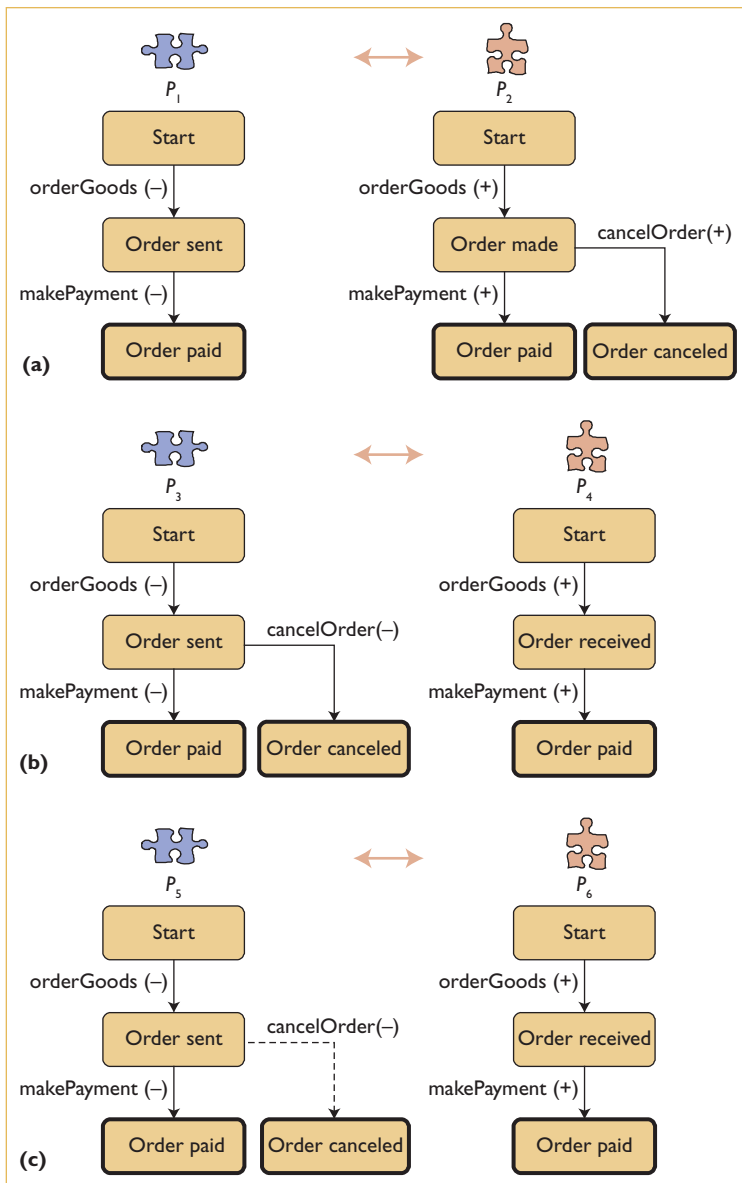


Figure 1. Protocol analysis. These examples demonstrate (a) full compatibility, (b) partial compatibility, and (c) difference analysis. We labeled input messages with the symbol (+) and output messages with the symbol (-).

information on its formal representation, based on *branching time semantics* (needed to precisely define operators and prove their properties), are available elsewhere.⁴

Compatibility

Compatibility analysis is concerned with verifying whether two services can interoperate. It's necessary for static and dynamic binding, and it also aids in evolution because it helps verify that a modified client can still interact as desired with a

certain service. More precisely, we identified two compatibility classes:

- *Partial compatibility.* A protocol P_x is partially compatible with another protocol P_y if some executions of P_x can interoperate with P_y – that is, if at least one possible conversation can take place among a service supporting P_x and one supporting P_y .
- *Full compatibility.* A protocol P_x is fully compatible with another protocol P_y if all the executions of P_x can interoperate with P_y – that is, P_y can understand any conversation that P_x can generate.

Protocol P_1 in Figure 1a can interact with protocol P_2 without generating errors, and P_2 can understand any conversation that P_1 can generate. Hence, P_1 is fully compatible with P_2 . In fact, P_2 could even engage in more complex conversations – for example, P_2 can provide order cancellation – but P_1 doesn't stress these aspects of the protocol. Protocol P_3 in Figure 1b isn't fully compatible with P_4 . Here, it's P_3 , which can send a `cancelOrder` message, that P_4 doesn't support. Hence, P_3 isn't fully compatible with P_4 . However, some conversations can occur between P_3 and P_4 – namely, all the ones in which the client doesn't cancel the order. Hence, P_3 is partially compatible with P_4 .

These notions of compatibility are useful in the context of Web services. For one thing, it doesn't make sense for incompatible services to interact because they can't hold a meaningful conversation. Furthermore, if only partial compatibility exists, the developer must be aware of this because the service won't be able to exploit its full capabilities when interacting with the partially compatible ones. As an example, Figure 1c graphically depicts the paths in protocol P_5 (solid lines) that can interact with P_6 and the paths that are related to illegal interactions (dashed lines).

This discussion shows the need for protocol analysis operators. We've only mentioned the need for two kinds of operators: Boolean operators that take two protocols as input and test whether they're partially or fully compatible, and an operator that takes two protocols as input and returns the conversations that can take place between two services supporting these protocols. These operators leverage other basic protocol management operators, such as intersection difference and projection. (More details and formal definitions are available elsewhere.⁴)

Replaceability

Replaceability analysis identifies whether two protocols are equivalent in terms of conversations that they can support, in general and when interacting with a certain client. Such an analysis also involves finding the set of conversations that both services can support if they aren't equivalent. For instance, we can use this to determine whether a new version of a service (protocol) can support the same conversations as the previous one or whether a newly defined service can support the conversations a given standard specification mandates. We identified several replaceability classes (see Figure 2), which provide basic building blocks for analyzing the commonalities and differences between service protocols.

Protocol *equivalence* occurs when two business protocols P_x and P_y can support the same set of conversations. Any conversation that doesn't result in errors – that is, it's legal – according to P_x will also be legal according to P_y , and vice versa. We can interchangeably use the two protocols in any context, and the change is transparent to clients. Protocols P_7 and P_8 in Figure 2a are equivalent.

Protocol *subsumption* occurs when protocol P_y is subsumed by another protocol P_x because P_x supports at least all the conversations that P_y supports. Hence, we can transparently use protocol P_x instead of P_y , but the opposite isn't necessarily true. Protocol P_8 in Figure 2b subsumes protocol P_9 .

The previous definitions discussed replaceability in general. However, it's important to understand whether we can use a service to replace another when it's interacting with a client – that is, to determine protocol *equivalence and subsumption, with respect to a client protocol*. Let's assume that we've developed a service to interact with another service our supplier offers. When we upgrade our service, we need to know if we can still have conversations with our supplier. This leads to a weaker definition of replaceability: a protocol P_x can replace another protocol P_y with respect to a client protocol P_c if every legal conversation between P_y and P_c is also a legal conversation between P_x and P_c . In this case, P_x can replace P_y to interact with P_c . For instance, protocol P_{12} in Figure 2c can replace P_{11} when interacting with P_c , although the two aren't equivalent, but P_{12} doesn't subsume P_{11} (and therefore can't replace P_{11} for arbitrary clients).

The last class addresses *replaceability with respect to an interaction role*. Let's assume that we have a service supporting protocol P_x that, among

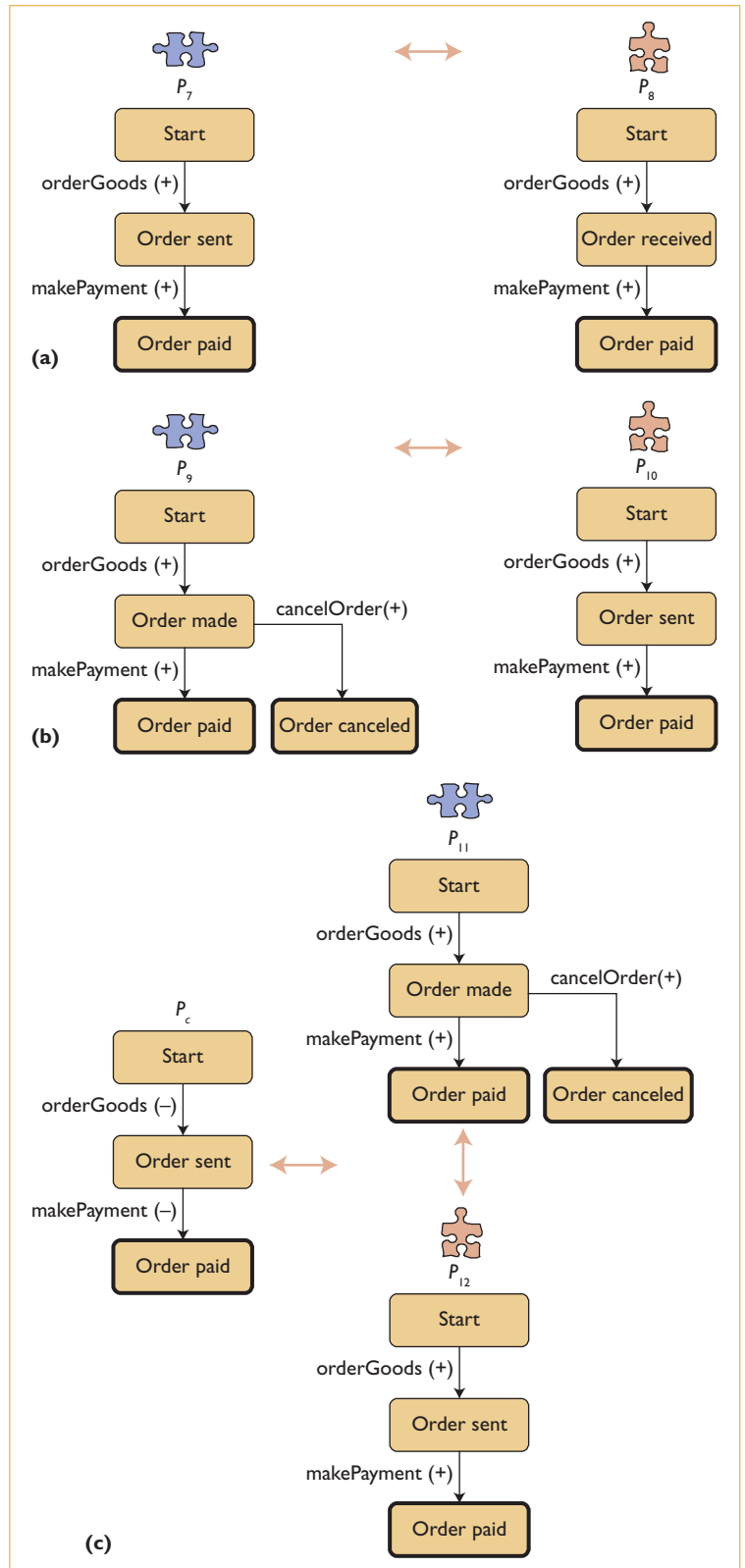


Figure 2. Protocol analysis. These examples demonstrate (a) equivalence, (b) subsumption, and (c) replaceability with respect to a client. We labeled input messages with the symbol (+) and output messages with the symbol (-).

other conversations, can also support some of the conversations mandated by a certain standard consortium and defined by a protocol called P_R . In this case, we might need to know if a new version of a protocol – say P_y – can support the same set of standard conversations. That is, can P_y replace P_x when P_x carries on conversations defined as part of P_R . We say that a protocol P_y can replace another protocol P_x with respect to a role P_R if P_y behaves as P_x when P_x behaves as P_R . This replaceability class lets us identify executions of a protocol P_x that protocol P_y can replace, even when P_x and P_y aren't comparable with respect to any of the previous replaceability classes. When we say that a protocol P_y behaves as a protocol P_x , we mean that not only can P_y support all conversations that P_x can, but also that P_y won't support all conversations that P_x doesn't support (again, with respect to an interaction role). This means that subsumption doesn't imply equivalence with respect to an interaction role.

As for compatibility, this discussion emphasizes the need for operators to analyze equivalence, subsumption, and different notions of replaceability. The need also exists for understanding, when two protocols aren't equivalent, which conversations both can or can't handle. This leads to providing operators to determine intersection and difference among protocols, among others, to identify which conversations can and can't be supported when we use a service in place of another. (More details appear elsewhere.⁴)

Model-Driven Development of Protocol Adapters

Whenever we have services that aren't fully compatible or equivalent, we might have to implement some modifications so that interoperability is possible. In many cases, we can achieve compatibility and replaceability by placing an adapter in front of the service that mediates for the differences.

However, using adapters only shifts the problem from implementing many variants of a service to developing many adapters. Hence, it's beneficial to identify a way to support adapter development and management, possibly in a semiautomated fashion. We take the view that, although concrete adapter specifications are application specific in many cases, it's possible to generically capture the type of differences among protocols and the way to resolve them into what we call *mismatch patterns*. Indeed, we've analyzed interfaces and protocols to iden-

tify the most typical differences, and for these, we've specified the corresponding mismatch patterns.⁵ This is akin to detecting structural and semantic differences in data mappings.⁷ Each mismatch pattern includes an *adapter template* to tackle the mismatch, as well as a *sample usage*. The template is useful as a guideline for developers and as input to a tool that automatically generates the adapter code.

We distinguish between operation- and protocol-level mismatches. Operation-level mismatches characterize heterogeneities related to operation definitions. Examples include differences that occur when two services S and SR have operations with the same functionality but differ in operation name, number, order, or type of I/O parameters. Protocol-level mismatches characterize heterogeneities related to message choreography as well as temporal and transaction properties. Examples include differences that occur when two services expect a messages in a different order, when one service sends messages that the other doesn't accept, when one service requires a single message to achieve certain functionality whereas the other requires several, and so on. (A comprehensive discussion on this topic appears elsewhere.⁵)

Essentially, an adapter maps interactions with protocol P (with which the client is designed to interact) into interactions with protocol PR (supported by the provider). This requires performing activities such as receiving messages, storing messages, transforming message data, and invoking service operations. We can model these tasks naturally using process-centric service composition languages such as BPEL. Our approach leverages patterns to assist in automatically generating BPEL process skeletons that map interactions according to protocol P into interactions according to protocol PR .

Table 1 shows an example of the message-ordering mismatch pattern. In this case, the same message is required in different orders by interacting services. The adapter template provides a solution for resolving the mismatch and corresponding BPEL activities. The sample usage in Figure 3 shows how to apply this mismatch pattern in a supply-chain management system. In this replaceability scenario (a client replaces service S with SR), the service SR expects the shipping-preferences message in a different order. The adapter then saves the message and sends it when the protocol of service SR requires it.

Table 1. An example of the message-ordering mismatch pattern.

| Mismatch Type | Message Ordering |
|---------------------|---|
| Template parameters | Protocols P (of service S) and PR (of service SR), message m to be reordered |
| Adapter template | Perform activities as prescribed by P for parts that don't 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) |

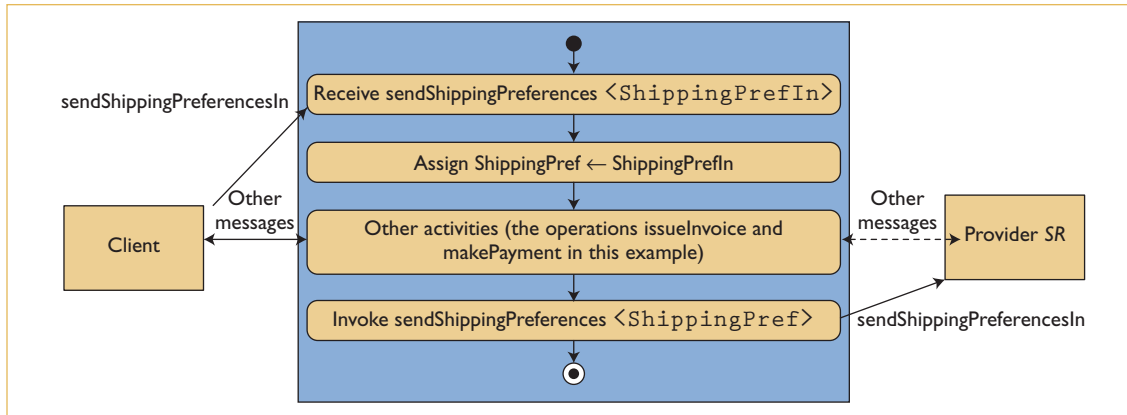


Figure 3. Mismatch pattern sample usage. The sample usage for the example of the message-ordering mismatch pattern in Table 1.

We choose a process-based notation because it's well suited to model business logic. Also, it's easy to derive a service's protocol specifications when we specify its business logic as a business process.³ Adapter processes are annotated with additional high-level directives to specify *adaptation abstractions*. In particular, the additional annotations can include XQuery functions to specify message transformations that are commonly needed in adapters as well as directives to help developers understand how to instantiate certain elements of the adapter template. A process-based notation is also appropriate for composing complex adapters from primitive adapter templates. (Due to space limitations, we don't discuss developing protocol adapters in this article.)

Service Mosaic

We've implemented our framework in a prototype platform, called Service Mosaic, as a computer-aided software engineering (CASE) tool set for modeling, analyzing, and managing service models including business protocols, orchestration, and adapters. (More details regarding the CASE tool set are available on the Service Mosaic project Web site, <http://servicemosaic.isima.fr/>.) We developed

the Service Mosaic platform using Java and Java 2 Enterprise Edition (J2EE) technologies and on top of the Eclipse platform. Figure 4 shows Service Mosaic's layered architecture. The platform's components are as follows:

- *Model representation and manipulation components* support representing, storing, and manipulating service descriptions and protocols. We provide basic manipulation operations of model elements, such as protocols, as core libraries that shield higher-level components from the details of their physical representations (XML, databases, and so on).
- *Analysis and management components* include operators for protocol compatibility and replaceability analysis,⁴ a code generator that produces BPEL skeletons from business protocol specifications,³ and a code generator that produces BPEL templates for implementing the adapters.⁵ We also provide a service-based interface for these components, and as such, they can be invoked by our GUI (in the development environment) or by external applications via SOAP clients.
- The *development environment* provides visual

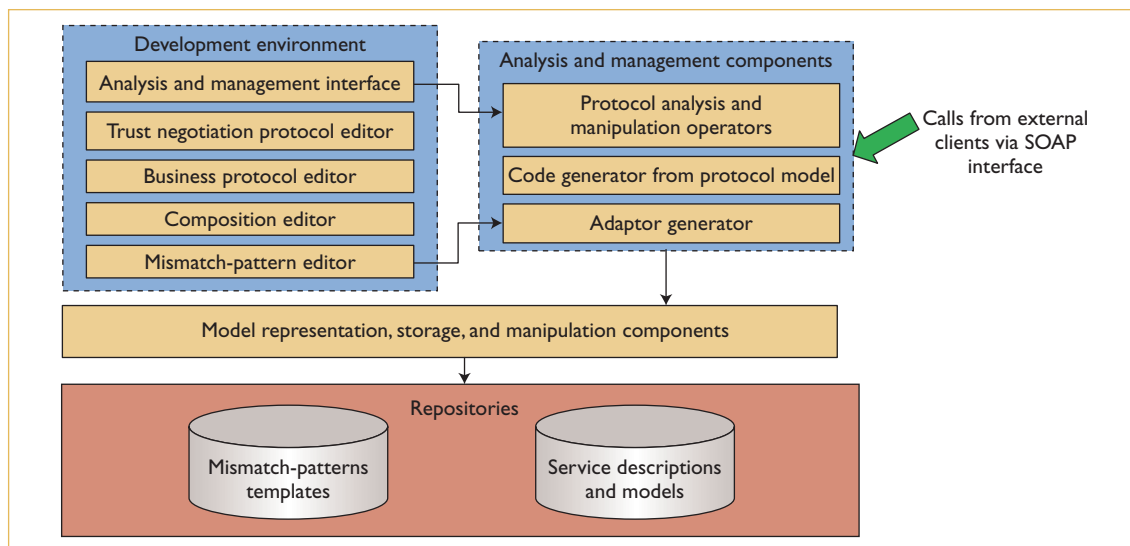


Figure 4. The Service Mosaic platform's layered architecture. It includes sets of classes for protocol definition, protocol analysis, and protocol data management.

exploration for modifying, analyzing, and managing model elements. It offers state-machines and state-chart graphical editors for business protocols, trust negotiation protocols, and orchestration models, for example.

We've validated Service Mosaic's usefulness in several applications. We applied protocol operators in the context of some supply-chain scenarios for protocol compatibility, replaceability, and protocol evolution. We also used mismatch patterns to identify mismatches of ArcWeb (www.esri.com/software/arcwebservices/) and MapPoint (www.microsoft.com/mappoint/default.msp) routing services, which provide the same functionalities using different interfaces (operations and messages). Generating adapters by instantiating adapter templates associated to each mismatch pattern remarkably reduces the time and effort needed to manually derive such adapters.

In terms of managing the Web service development life cycle, technology is still in the early stages. There has been little concern so far regarding methodologies and tools for conceptual modeling and development of services. Service development tools (such as BPEL4WJ and the Oracle BPEL Process Engine) that support emerging standards and protocols have started to appear, but these efforts and tools are mainly concerned with implementation aspects. We argue that the effective use and widespread adoption of service

technologies and standards requires high-level frameworks and methodologies for supporting automated development and interoperability. Prior research proposed a methodology for defining processes based on composing protocols.⁸ The paper also proposed notions of refinement among protocols and of commitments. Another work investigated compliance verification with respect to a commitment protocol.⁹ These approaches differ from the one we propose here. We don't aim to understand if two protocols can provide the same commitment or to verify whether a given protocol execution leads to a commitment violation. Our notions of compatibility and replaceability (as well as our analysis and operators) focus on understanding whether two services can syntactically interact, and if so, how. They aim at understanding if the protocols can exchange messages without resulting in runtime errors and which conversations are (or aren't) possible. This is a prerequisite to any other kind of semantic or conceptual similarity analysis among services.

Our current work focuses on extending analysis and management techniques for timed protocols, and we plan to concentrate on transactional aspects. We're also investigating business protocol discovery techniques to bring the benefits of protocol-based interactions to services that don't explicitly model business protocols and to interactions that involve groups of services. Finally, we plan to explore techniques for cataloging and analyzing previous adapters to improve the process of developing new adapters. □

References

1. M.P. Papazoglou and D. Georgakopoulos, "Special Issue on Service-Oriented Computing," *Comm. ACM*, vol. 46, no. 10, 2003.
2. N. Leavitt, "Are Web Services Finally Ready to Deliver?" *Computer*, vol. 37, no. 11, 2004, pp. 14–18.
3. B. Benatallah, F. Casati, and F. Toumani, "Web Service Conversation Modeling: A Cornerstone for e-Business Automation," *IEEE Internet Computing*, vol. 8, no. 1, 2004, pp. 46–54.
4. B. Benatallah, F. Casati, and F. Toumani, "Analysis and Management of Web Service Protocols," *Conceptual Modeling: ER 2004, Proc. 23rd Int'l Conf. Conceptual Modeling*, LNCS 3288, Springer-Verlag, pp. 524–541.
5. B. Benatallah et al., "Developing Adapters for Web Services Integration," *Proc. 17th Conf. Advanced Information Systems Eng. (CAiSE)*, LNCS 3520, Springer-Verlag, 2005, pp. 415–429.
6. K. Baina et al., "Model-Driven Web Service Development," *Proc. 16th Conf. Advanced Information Systems Eng. (CAiSE)*, LNCS 3084, Springer-Verlag, 2004, pp. 290–306.
7. E. Rahm and P.A. Bernstein, "A Survey of Approaches to Automatic Schema Matching" *VLDB J.*, vol. 10, no. 4, 2001, pp. 334–350.
8. M.P. Singh et al., "Protocols for Processes: Programming in the Large for Open Systems," *Oopsla Companion*, ACM Press, 2004, pp. 120–123.
9. M. Venkatraman and M.P. Singh, "Verifying Compliance with Commitment Protocols," *Autonomous Agents and Multi-Agent Systems*, vol. 2, no. 3, 1999, pp. 217–236.


Boualem Benatallah is an associate professor at the University of New South Wales, Australia. His latest work focuses on service-oriented computing and large-scale data sharing. Benatallah has a PhD in computer science from the University of Grenoble, France. He is a member of the IEEE and ACM. Contact him at boualem@cse.unsw.edu.au.

Fabio Casati is a senior researcher at Hewlett-Packard Labs, Palo Alto. His research interests include business processes, Web services, business-aware application management, and middleware intelligence. Casati has a PhD in computer science from Politecnico di Milano. Contact him at casati@hp.com.

Farouk Toumani is a senior lecturer at the School of Engineering in Computer Science, Modeling, and Applications, University Blaise Pascal, France. His research interests include Web services, the Semantic Web, and knowledge representation for databases. Toumani has a PhD in computer science from the National Institute of Applied Sciences, Lyon, France. Contact him at ftoumani@isima.fr.

Julien Ponge is a PhD student at the University Blaise Pascal, France, and under cotutelle agreements with the University of New South Wales, Australia. His research interest is in applications-integration issues in Web services. Ponge has a masters by research in computer science from the University Blaise Pascal, France. Contact him at ponge@isima.fr.

Hamid Reza Motahari Nezhad is a PhD student in computer science at the University of New South Wales, Australia, and under an agreement with National Information and Communications Technology of Australia (NICTA). His research interests are in Web service interoperability and analysis, and management of Web services business protocols. Nezhad has an MS in computer science from the Amirkabir University of Technology, Tehran, Iran. He is a student member of the IEEE and the Australian Computer Society. Contact him at hamidm@cse.unsw.edu.au.



IEEE distributed systems
Expert-authored articles and resources ONLINE

IEEE's online-only magazine offers **free access** to peer-reviewed articles, departments exploring the latest trends and news, and communities including:

- ✓ Web Systems
- ✓ Distributed Agents
- ✓ Security
- ✓ Cluster & Grid Computing
- ✓ And More!

GET IN THE KNOW WITH DSO!

<http://dsonline.computer.org>