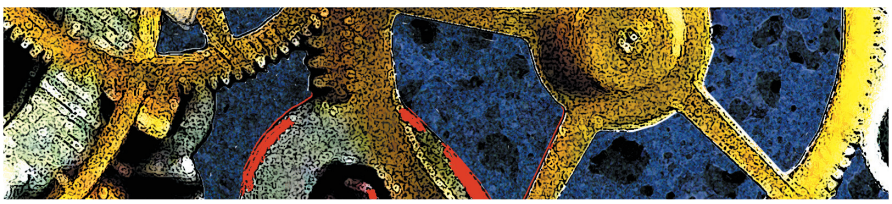
# Managing Evolving Services

**Michael P. Papazoglou and Vasilios Andrikopoulos**, Tilburg University

**Salima Benbernou**, Paris Descartes University

*// Services are subject to constant change and variation, leading to continuous redesign and improvement. However, service changes shouldn't be disruptive by requiring radical modifications or by altering the way that business is conducted. //*



**SERVICE EVOLUTION IS** "the continuous process of development of a service through a series of consistent and unambiguous changes."[1] The evolution of a service is expressed through the creation and decommissioning of different service versions during its lifetime. These versions must be aligned with each other in a nondisruptive manner and in a way that lets a service designer track modifications and their effects on the service. To control service evolution, a designer must know why a change was made, what its implications are, and whether the change is consistent. Eliminating spurious results and inconsistencies that occur because of uncontrolled changes is necessary for services to evolve gracefully, ensure stability, and handle variability in their behavior.

We can classify the nature of service changes depending on their causal effects:

- *Shallow changes*. These are small-scale incremental changes localized to a service or restricted to the service's clients.
- *Deep changes*. These are large-scale transformational changes cascading beyond a service's clients, possibly to entire value chains (end-to-end service networks).

Typical shallow changes focus on *structural-level* changes (service types, messages, interfaces, and operations) and *business protocol* changes (the conversations in which the service participates). Typical deep changes include *policy-induced* (pertaining to business agreements between service providers and consumers), *operational behavior* (for example, when the service fulfills its expected function in a timely and orderly manner), and *nonfunctional* changes (relating to quality-of-service, or QoS, issues and service-level agreement, or SLA, guarantees for individual and end-to-end services).

While both shallow and deep changes need an appropriate versioning strategy, deep changes further introduce several intricacies of their own and require the assistance of a change-oriented service life cycle to allow services to react appropriately to changes as they occur.

In this article, we discuss a causal model of service changes that addresses the effects of both shallow and deep changes. This article is largely based on concepts and definitions found in previous work.[1] The definitions used have been revised and amended on the basis of formalization and compatibility analysis, prototype implementation, comparison with functionality offered by open standards, and an empirical in-depth investigation using an industrial strength case study.

## Background

Evolution in software systems traditionally has been considered as either a part, or a synonym of software

0740-7459/11/$26.00 © 2011 IEEE

## SERVICES VERSUS COMPONENT EVOLUTION

While evolving, both components and services can offer multiple interfaces for the same functionality. This makes it possible to sustain several simultaneous versions. However, subtle differences exist between these two related technologies summarized by the following dimensions:

- *Coupling*. Services make use of abstract message definitions to mediate their binding with respect to each other. They focus on message and event definitions rather than method signatures, which typify components.[1]
- *Invocation*. Services introduce the concept of service capability, which describes the classification, functionality, and conditions under which a particular service can be discovered and invoked. This leads to reactive services (which can respond to environmental demands without compromising operational efficiency). Components typically focus on locating and invoking other components by name.
- *Binding*. An SOA application might choose a service dynamically on the basis of quality of service, using parameters such as response time, throughput, and availability. Components depend on mechanisms like glue coding, wrappers, delegation, or aggregation for binding.[1]
- *Composition*. Service composition leads naturally to the creation of higher-level services that are typically long-running, coordinated workflow service arrangements specified according to business protocols. Component composition is typically on a lower level that depends on the component model used.[2]

Managing service evolution therefore requires a systematic revisit of the techniques and theories for component evolution.

### References

1. I. Crnković et al., "A Classification Framework for Software Component Models," *IEEE Trans. Software Eng.*, vol. 99, 2010; http://doi.ieeecomputersociety.org/10.1109/TSE.2010.83.
2. A. Elfatatry, "Dealing with Change: Components versus Services," *Comm. ACM*, vol. 50, no. 8, 2007, pp. 35–39.

maintenance. The insight gained from early studies resulted in empirical laws that drive and govern software systems' evolution.[2] Evolution is particularly important in distributed systems because of a complex web of software interdependencies. As Keith Bennett and Václav Rajlich point out,[3] attempting to apply the conventional maintenance procedure (halt operation, edit source, and reexecute) in large distributed systems (such as those emerging in service-oriented environments) isn't sensible. The difficulty of identifying which software artifacts form the system itself is nontrivial, especially in the context of large service networks. In addition, the lack of ownership and access to the actual source code (if any) of third-party services (because of the service-oriented architecture, or SOA, principles of encapsulation and loose coupling), doesn't allow us to apply various maintenance techniques, such as refactoring or impact analysis.

The difficulty of approaching system evolution purely as a maintenance activity has already appeared in the study of component engineering in general, and component evolution in particular. The basic ideas and solutions for the evolution of component-based systems (CBS) are summarized elsewhere.[4] Evolving a component includes changes in both its interfaces and its implementation, with each having different evolutionary requirements. Because of their composability and emphasis on reuse, components exhibit strong dependencies with the components they consume. Changing a component might therefore have implications for other components and upgrading to a new component might require that both versions (old and new) be deployed in parallel while the transition takes place. Finally, identifying and distinguishing between different component versions requires the introduction of software configuration management (SCM) techniques, such as version identifiers incorporated into the component metadata. Because version identifiers don't explain what changes occurred between versions, checking for compatibility must be performed separately.

Historically and conceptually, we can consider CBS as a predecessor of service-oriented computing. However, we must remember that components and services differ in terms of coupling, binding, granularity, delivery, and communication mechanisms and overall architecture (see the "Services versus Component Evolution" sidebar).

## Case Study

To provide a systematic way of looking at service evolution, we use the industrial-strength "Automotive Purchase Order Processing" case study developed jointly with IBM Almaden that the S-Cube Network of Excellence (www.s-cube-network.eu) uses as a validation scenario. The case study is an example of how to realize standardized supply-chain activities using SOA-based processes for a fictitious enterprise in the automobile industry called Automobile Incorporation (AutoInc). AutoInc consists of different business units—such as sales, logistics, and manufacturing—and collaborates with external partners like suppliers, banks,

and transport carriers. The case study describes a typical automobile ordering process, where customers can place automated orders with AutoInc.

Within this context, we performed a case study on the effect of changes to different services. We developed various evolutionary scenarios, describing the required actions and proposed modifications to purchase order-processing services. We identified these scenarios either as maintenance actions (for example, optimizing the performance of some services) or as reengineering efforts (completely redesigning service interfaces).

## Dealing with Shallow Changes

Shallow changes affect both individual and end-to-end services. To deal with shallow changes, we discuss helpful practices for service compatibility and versioning derived on the basis of structural and business protocol service changes.

### Service Version Compatibility

To deal with message exchanges between a service provider and a client, they must still be able to exchange valid messages despite any interface changes that could happen to either side. To achieve this, we must rely on the notion of service version compatibility. *Service version compatibility* guarantees that we can introduce a new version of either a provider or a client of service messages without changing the other. We classify compatibility in two dimensions:[5]

- *Horizontal compatibility* or *interoperability*. This means that two services can participate successfully in an interaction, either as service producers or consumers.
- *Vertical compatibility* or *substitutability* (from the provider's perspective) or *replaceability* (from the consumer's perspective). This describes the requirements that let one service version replace another in a given context.

Traditionally, two types of changes to a service definition can guarantee version compatibility:[6]

- *Backward compatibility*. A new version of a message client is introduced and the message providers are unaffected. The client might introduce new features but should still support all the old ones.
- *Forward compatibility*. A new version of a message provider is introduced and the message clients that are only aware of the original version are unaffected. The provider might have new features but shouldn't add them in a way that breaks any old clients.

Some types of changes that are both backward and forward compatible include the addition of new service operations to an existing service description. In this case we're talking about full compatibility. Full compatibility lets you replace an existing service version with an equivalent (that is, compatible) one without affecting the correct operation and performance of its clients.

From a practical standpoint, compatible service evolution in the services description standard Web Services Description Language (WSDL) 2.0 is limited to service changes that are either backward or forward compatible, or both.[7] The types of service changes that are compatible are

- the addition of new WSDL operations to an existing WSDL document, and
- the addition of new XML schema types within a WSDL document that aren't contained within previous types.

Incompatible change types, on the other hand, include removing an operation, renaming an operation, changing an operation's parameters (in data type or order), and changing the structure of a complex data type. An alternative approach enables the compatible evolution of services.[5] Instead of restricting service changes to the short list that we mentioned, a theoretical framework allows for reasoning on the evolution of services. As a result, further compatible changes (called *T-shaped*) are allowed; for example, removing data elements from incoming message data types and adding data elements in outgoing message data types. Service version compatibility for structural changes is based on two fundamental premises of type theory:[8]

> Evolution is particularly important in distributed systems because of a complex web of software interdependencies.

- *Service argument contravariance*. If we redefine the argument of a service, the new argument types must always be an extension (generalization) of the original ones.
- *Service result covariance*. If we redefine the result of a service, the new result types must always be a restriction (specialization) of the original ones.

For example, we can enhance the message payload of services in the

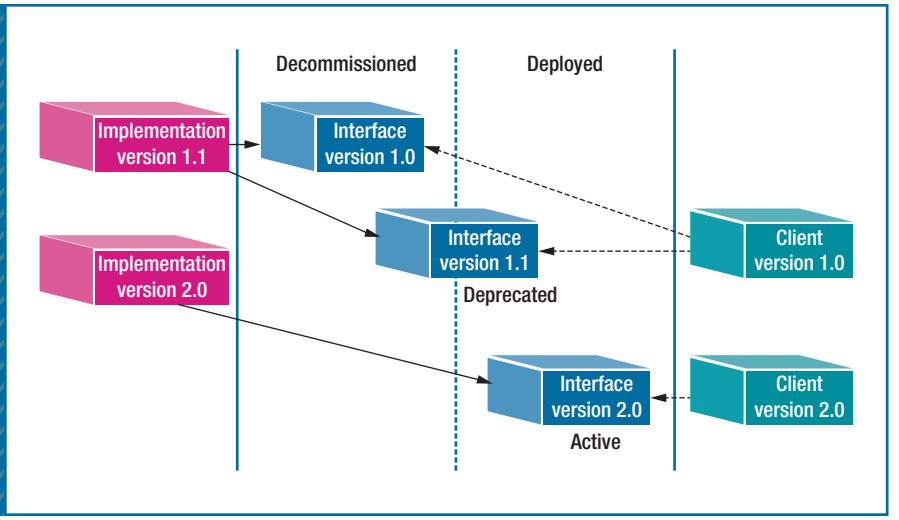



**FIGURE 1.** Service versioning. With a compatible change, clients can continue to use a previous version while adjusting to the latest service interface, or until the previous version is decommissioned. With an incompatible change, the client must identify the new version and adjust to the new interface before the old one is decommissioned.

AutoInc case study by time-stamping information if the messages are produced as output of the service, but not if they're consumed as input from the service. This reasoning is inversed when considering the evolution of the clients of AutoInc's services. To realize these capabilities, we must replace the existing model of validating incoming messages against their original (and possibly obsolete) XML schema with a direct marshalling of the message content into objects—and checking the compatibility of the message with the expected message schema separately.[5]

When evolving a business protocol, states and transitions can be added or removed from an active protocol. A new version of a protocol is created each time its internal structure or external behavior changes. We call the parts of a specific protocol that a client can observe the *protocol view*. Because the client's protocol view is restricted to the parts that directly involve the client, a client might have equivalent views on different protocols. Evolution doesn't affect clients whose views on the original and target protocols are the same. Business protocol evolution is considered by Seung Hwan Ryu and his colleagues when they distinguish between various aspects of protocol changes.[9] *Protocol compatibility* aims at assessing whether two protocols can interact—that is, if it's possible to have a conversation between two services despite changes to their protocols. Two protocols can have *complete* compatibility, meaning that both protocols can understand the conversations of the other; or *partial*, when there's at least one conversation possible between them.

Protocol compatibility is essential for redesigning service interfaces while allowing existing clients to consume them. In the case of AutoInc, it allows for the replacement of single-entry-point asynchronous interfaces with more complicated multiple-entry interfaces that provide for both synchronous and asynchronous communication on demand. This allows for the deployment of one service interface to cover both types of communication instead of having to provision for multiple service interfaces.

## Versioning Shallow Changes

A robust versioning strategy allows for service upgrades and improvements, while continuously supporting previous versions. Service versioning is therefore important for both developers and providers, building on the notion of service version compatibility. In the case of service evolution, the cost of provisioning for multiple service interfaces is nonlinear. As with component evolution, developing a new interface requires additional effort in binding the interface versions with the underlying implementation. In the case of multiple active service versions, however, each active version also requires access to resources in the supporting infrastructure. Furthermore, each version adds managerial overhead in terms of monitoring and auditing to ensure that it complies with the agreed-upon SLAs. As such, providing for multiple interfaces in services can overtax the service provider. Therefore, developers should minimize the amount of active versions by employing compatible changes.

With a compatible change, the service implementation need only support the latest version of a service interface[10]—for example, implementation version 1.1 of a Receive Purchase Order service in AutoInc supports interface version 1.1 in Figure 1. A client can continue to use a previous service version (interface 1.0) while adjusting to a new version of the interface description or until the version is decommissioned. With an incompatible change, the client receives a new version of the interface description (interface version 2.0 in Figure 1) and must adjust to the new interface before the old interface is decommissioned. The service must either continue to support both versions during the handover period (having one active and one deprecated version as in Figure 1), or the service and the clients must change at the same time. Alternatively, the client can continue until it encounters an error, at which point it switches to the new version. While this is common practice, it's prone to errors and inconsistencies in provider and consumer interaction and thus should be avoided.

## Dealing with Deep Changes

Deep changes characterize complex services and require that such services be redefined and possibly realigned within an entire end-to-end service. Deep service changes require a *change-oriented service life-cycle* methodology to provide a sound foundation for spreading changes in an orderly fashion so that impacted services are appropriately (re)configured, aligned, and controlled as the changes occur.

The change-oriented service life cycle ensures that standardized methods and procedures are used for the prompt, efficient handling of all service changes to minimize the impact on service operation and quality. This means that in addition to functional (structural and behavioral) changes, a change-oriented service life cycle must deal with policy-induced, operational behavior, and nonfunctional changes. The objective is to provide end-to-end QoS capabilities by ensuring that services are performing as desired and that service designers are able to anticipate out-of-control or out-of-specification conditions, and responds to them appropriately. This includes traditional QoS capabilities, such as security, availability, accessibility, integrity, and transactionality, as well as *service volumes* (number of service events, number of items consumed, and service revenue) and *velocities* (performance characteristics). These measurements show how an enterprise is performing its services.

Figure 2 illustrates a deep change-oriented service life cycle that comprises a set of interrelated phases, activities, and tasks that define the change process from the beginning to end. Each phase produces a major deliverable that contributes to the change objectives. Logical breaks in the change process are associated with key decision points.

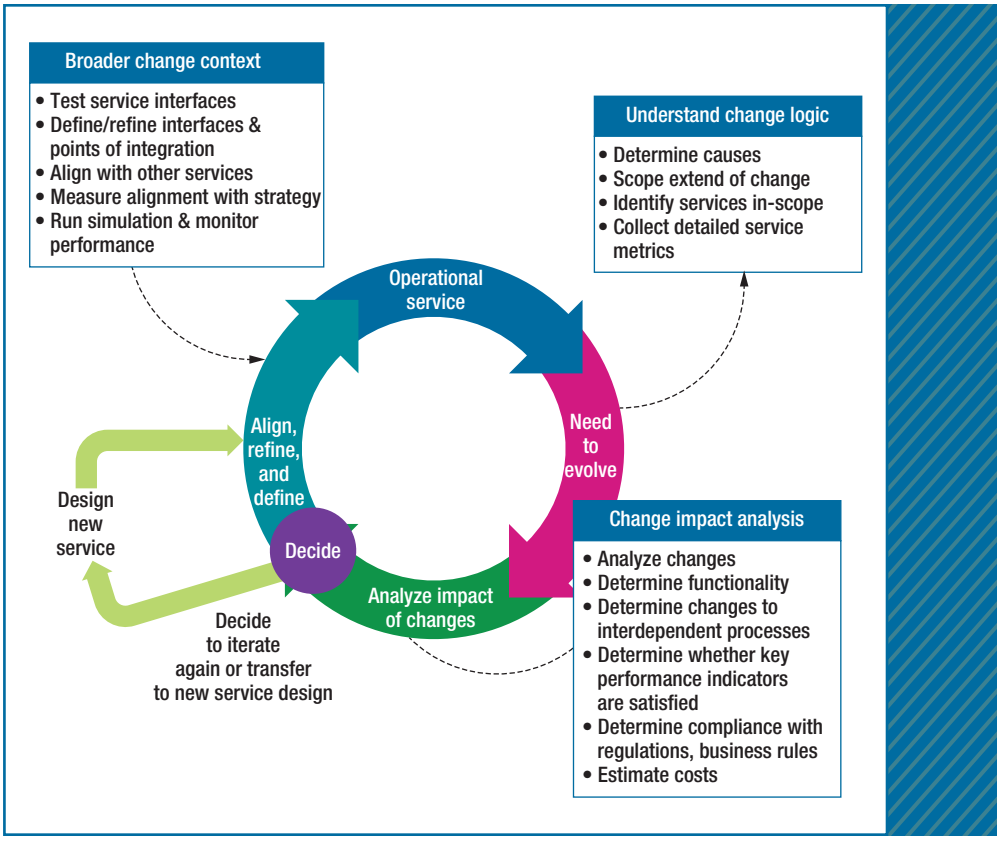



**FIGURE 2.** The change-oriented life cycle. This service life cycle contains a set of interrelated phases, activities, and tasks that define the change process from the beginning to end.

### The Change-Oriented Life Cycle: Phase 1

The initial phase (need to evolve) focuses on identifying the need for change and scoping its extent. One of the major elements of this phase is understanding the causes of the need for change and their potential implications. For instance, compliance to regulations is a major force for change. This might lead to the transformation of all services within a service network.

Here, the impacted individual services in an end-to-end service (or service-in-scope) must be identified. In addition, service performance metrics, such as key performance indicators (KPIs), must be collected. In the case of the AutoInc Purchase Order Processing service, performance analysis attributed a large percentage of the response time of the public (partner-exposed) services to the communication overhead among different business unit services. A redesign of the services of AutoInc is therefore required, which triggers the life cycle's next phase.

### Phase 2

The second phase in Figure 2 (analyze the impact of changes) focuses on the actual analysis, redesign, or improvement of existing services. The ultimate objective of service change analysis is to provide an in-depth understanding of the functionality, scope, reuse, and granularity of services identified for change. The problem lies in determining the difference between existing and future service functionality. To analyze and assess the impact of changes, organizations rely on the existence of an "as-is" and a "to-be" service model, rather than applying the changes directly on operational services. Analysts rely on an "as-is" service model to understand the portfolio of available services. This model serves

as the basis for conducting a thorough reengineering analysis of the current portfolio of available services that need to evolve. The "to-be" service model serves as the basis for describing the target service functionality and performance levels after applying the required changes.

To determine the differences between these two models, a *gap analysis model* helps prioritize, improve, and measure the impact of service changes. Gap analysis uses a services realization strategy by incrementally adding more implementation details to an existing service to bridge the gap between the "as-is" and "to-be" service models. Gap analysis commences with comparing the "as-is" with the "to-be" service functionality to determine differences in terms of service performance (KPI measures) and capabilities. Service capabilities determine whether a process can meet specifications, customer requirements, or product tolerances. The gap analysis for AutoInc revealed that, despite being able to reuse many existing services internal to the process, composite services that depended on services in different units had to be redrawn. This resulted in proposing changes to public services, potentially impacting clients of the end-to-end service. The resulting analysis, however, showed that the "to-be" model would perform better while respecting all existing SLAs.

Because service changes can spill over to other services in a service chain, one of the determining factors in service change analysis is the ability to recognize the scope of changes and functionality. When dealing with deep service changes several problems must be addressed:[11]

- *Service flow.* Typical problems involve the logical completeness of a service upgrade, sequencing and duplication of activities, decision-making, and a lack of service measures.
- *Service control.* Service controls define or constrain how a service is performed. Broadly speaking, there are two general types of control problems: problems with policies and business rules, and problems with external services.
- *Overlapping services' functionality.* In such cases, a service-in-scope might share identical business logic and rules with other related services. Here, there's a need to rationalize services and determine the proper level of service commonality. During this procedure, service design principles[12] such as *service coupling* and *cohesion* must be employed to achieve success.
- *Conflicting services' functionality* (including bottlenecks or constraints in the service value stream). Conflicts include problems where a service-in-scope isn't aligned with the business strategy, a service pursues a strategy that's in conflict with, or is incompatible with its value chain, and cases where introducing a new policy or regulation would make it impossible for the service-in-scope to function.
- *Service input and output problems.* These problems include low QoS input or output, and delayed input or output.

> Broadly speaking, there are two general types of control problems: problems with policies and business rules, and problems with external services.

Cost estimation involves identifying and weighing the services under consideration for the reengineering project to determine the approximate cost. When costs prove prohibitive for in-house implementation, outsourcing is worth considering. In the case of AutoInc, cost estimation showed that the implementation of the "to-be" model was within budget. Nevertheless, the analysis determined that part of the migration cost was borne by the end-to-end service clients because they needed to adapt to the new services' design. Therefore, it seemed appropriate to modify the service-charging policies.

### Phase 3

During the third and final phase (the align, refine, and define phase shown in Figure 2), the new services are aligned, integrated, simulated, tested, and then put into production. To achieve this, service developers create a *service integration model* to help implement the service integration strategy. The model establishes, among other things, integration relationships between service consumers and providers involved in business interactions. It also includes steps that determine message distribution needs, parties responsible for delivery, and provides a service delivery map. Finally, the service integration model considers message and process orchestration needs. The resulting service integration strategy includes service design models, policies, SOA governance options, and reliance on organizational and industry best-practices. All these concerns should be considered when designing integrated end-to-end services that span organizational boundaries.

The role of the services integration model ends when an upgraded service architecture is completely expressed and validated against technological

specifications from infrastructure, management, monitoring, and technical utility services. For AutoInc, the services integration model resulted in a comprehensive plan for restructuring and reorganizing the services comprising the end-to-end service, resulting in a more effective and cost-efficient product.

Our causal model for addressing service changes deals with the effects of both shallow and deep changes. In the case of shallow changes, we defined a theoretical approach to decide whether a change is shallow and a versioning strategy to support multiple versions of services.

To address the problems of deep changes, we introduced a change-oriented service life-cycle methodology and described its phases. In particular, we discussed when a change in a service is triggered, how to analyze its impact, and the possible implications of the implementation of the change for the service provider and consumers. Because of its wide scope and the multitude of issues related to the change-oriented life cycle, further research on this subject is essential. A formal model for deep changes (based on the one for shallow changes) is the main goal of our future work. Of particular interest is the transition between the formal models for shallow and deep changes and the way they handle changes according to contractual service specifications.[13]


## Acknowledgments

The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube).


## References


1. M.P. Papazoglou, "The Challenges of Service Evolution," *Proc. Int'l Conf. Advanced Information Systems Eng.*, Springer-Verlag, 2008, pp. 1–15.
2. M.M. Lehman, "Laws of Software Evolution Revisited," *Proc. 5th European Workshop Software Process Technology*, Springer-Verlag, 1996, pp. 108–124.
3. K.H. Bennett and V.T. Rajlich, "Software Maintenance and Evolution: A Roadmap," *Proc. Conf. Future of Software Eng.*, ACM Press, 2000, pp. 73–87.
4. A. Stuckenholz, "Component Evolution and Versioning State of the Art," *Proc. Sigsoft Software Eng. Notes*, vol. 30, no. 1, 2005, p. 7.
5. V. Andrikopoulos, *A Theory and Model for the Evolution of Software Services*, doctoral dissertation, Center for Economic Research dissertation series no. 262, Tilburg University, 2010.
6. D. Orchard, ed., "Extending and Versioning Languages," World Wide Web (W3C) Tech. Architecture Group, Nov. 2007; www.w3.org/2001/tag/doc/versioning.
7. D. Booth and C.K. Liu, eds., "Web Services Description Language (WSDL) Version 2.0 Part 0: Primer," W3C, June 2007; www.w3.org/TR/wsdl20-primer.
8. G. Castagna, "Covariance and Contravariance: Conflict without a Cause," *ACM Trans. Programming Languages and Systems*, vol. 17, no. 3, 1995, pp. 431–447.
9. S. Ryu et al., "Supporting the Dynamic Evolution of Web Service Protocols in Service-Oriented Architectures," *ACM Trans. Web*, vol. 2, no. 2, article 13, 2008; http://doi.acm.org/10.1145/1346237.1346241.
10. K. Jerijærvi and J.-J. Dubray, "Contract Versioning, Compatibility and Composability," *InfoQ Magazine*, Dec. 2008; www.infoq.com/articles/contract-versioning-comp2.
11. P. Harmon, *Business Process Change*, Morgan Kaufmann, 2007.
12. M.P. Papazoglou, *Web Services: Principles and Technology*, Prentice Hall, 2007.
13. V. Andrikopoulos, S. Benbernou, and M.P. Papazoglou, "Evolving Services from a Contractual Perspective," *Proc. Int'l Conf. Advanced Information Systems Eng.*, Springer-Verlag, 2009, pp. 290–304.


## ABOUT THE AUTHORS


**MICHAEL P. PAPAZOGLOU** is the chair of the Computer Science Department at Tilburg University. He's also the scientific director of the European Research Institute in Service Science (ERISS) and the EC's Network of Excellence, S-Cube. His research interests include service-oriented computing, Web services, large-scale data sharing, business process management, and federated information systems and distributed computing. Papazoglou has a PhD in microcomputers systems engineering from the University of Dundee, Scotland. He's a Golden Core Member and a Distinguished Visitor of the IEEE Computer Society. Contact him at m.p.papazoglou@uvt.nl.



**VASILIOS ANDRIKOPOULOS** works as a postdoctoral researcher for ERISS at Tilburg University. His research interests include service-oriented architecture and computing, with an emphasis on the evolution of services. Andrikopoulos received his PhD in information management from Tilburg University. Contact him at v.andrikopoulos@uvt.nl.



**SALIMA BENBERNOU** is a full professor of computer science at Paris Descartes University. Her research interests include formal models for service-oriented computing, privacy in information systems and databases, data sharing, and knowledge representation. Benbernou has a PhD in computer science from the Université de Valenciennes et du Hainaut-Cambrésis. Contact her at salima.benbernou@parisdescartes.fr.