On the Evolution of Services

Vasilios Andrikopoulos, Salima Benbernou, and Michael P. Papazoglou, *Senior Member*, *IEEE*

Abstract—In an environment of constant change and variation driven by competition and innovation, a software service can rarely remain stable. Being able to manage and control the evolution of services is therefore an important goal for the Service-Oriented paradigm. This work extends existing and widely adopted theories from software engineering, programming languages, service-oriented computing, and other related fields to provide the fundamental ingredients required to guarantee that spurious results and inconsistencies that may occur due to uncontrolled service changes are avoided. The paper provides a unifying theoretical framework for controlling the evolution of services that deals with structural, behavioral, and QoS level-induced service changes in a type-safe manner, ensuring correct versioning transitions so that previous clients can use a versioned service in a consistent manner.

Index Terms—Services engineering, service evolution, versioning, service compatibility.

1 Introduction

Services are subject to constant change and variation. Service changes originate from the introduction of new functionality to a service, the modification of already existing functionality to improve performance or the inclusion of new policy constraints that require that the behavior of services be altered. Such changes lead to a continuous service redesign and improvement effort. However, they should not be disruptive by requiring radical modifications in the very fabric of existing services and applications.

Changes can happen at any stage in the service life cycle and have an unpredictable impact on the service stakeholders [1]. Being therefore able to control how changes manifest in the service life cycle is essential for both service providers and service consumers.

Service evolution is the disciplined approach for managing service changes and is defined as *the continuous process of development of a service through a series of consistent and unambiguous changes* [2]. Service evolution is expressed through the creation, provisioning, and decommissioning of different variants of the service—called *versions*—during its lifetime. These versions must be aligned with each other in such a way as to allow a service developer to track the various modifications introduced over time and their effects on the original service. To control service development, a developer needs to know why a change was made, what its implications are, and whether the resulting service version is consistent and does not render its consumers inoperable.

- *V. Andrikopoulos is with IAAS, University of Stuttgart, Universitaetsstrasse 38, D-70569 Stuttgart, Germany. E-mail: vasilios.andrikopoulos@iaas.uni-stuttgart.de.*
- *S. Benbernou is with the Laboratoire d'Informatique Paris Descartes (LIPADE), Université Paris Descartes, Room 814 K, 45 rue des Saints Pères, Cedex 06, Paris 75270, France. E-mail: salima.benbernou@parisdescartes.fr.*
- *M.P. Papazoglou is with the European Research Institute in Service Science (ERISS), Tilburg University, Information Management Department, Warandelaan 2, 5000 LE Tilburg, The Netherlands. E-mail: M.P.Papazoglou@uvt.nl.*

Manuscript received 17 Nov. 2009; revised 26 Apr. 2010; accepted 26 Feb. 2011; published online 1 Mar. 2011.
Recommended for acceptance by B. Nuseibeh.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-2009-11-0359.
Digital Object Identifier no. 10.1109/TSE.2011.22.

Eliminating spurious results and inconsistencies that may occur due to uncontrolled changes is thus a necessary condition for services to evolve gracefully, ensure service stability, and handle variability in their behavior. With the above backdrop, we can classify service changes on the basis of their causal effects as

1. *Shallow changes*. Small-scale, incremental changes that are localized to a service and/or are restricted to the consumers of that service.
2. *Deep changes*. Large-scale, transformational changes cascading beyond the consumers of a service possibly to consumers of an entire end-to-end service chain.

Deep changes require an approach dealing with shallow changes, which form their foundation, as well as with intricacies of their own. For this purpose, deep changes rely on the assistance of a change-oriented service life-cycle methodology to respond appropriately to changes [2]. They are predominantly concerned with analyzing the effects and dealing with the ramifications of operational efficiency changes and changing compliance requirements which rely on service composition reengineering exercises. Deep changes therefore constitute a challenging and open research problem in the context of service engineering. In this work, we focus on developing a sound theoretical framework and approach for dealing with shallow changes as a precursor to dealing with deep changes.

Ensuring that a change is shallow requires service developers to reason about the effect of change both on service providers as well as service consumers. The number, type, and specific needs of the consumers are often unknown and their dependencies on the service are transparent to the developer. This reasoning can only be performed on the basis of formal principles and theories that control shallow changes. Such an approach for controlling shallow changes is currently unavailable and this work aims to address this need. Without a comprehensive formal framework for controlling and delimiting service evolution, service versioning cannot succeed. The goal of this research is therefore *to provide a theoretical framework to assist service developers in their effort to develop evolving services while constraining the*

effects of changes so that they do not spread beyond an evolving service. In this way, service changes are kept localized so that they neither lead to inconsistent results, nor do they disrupt service clients. In particular, we shall constrain our work to addressing the effects of shallow changes on service interface specifications.

Services typically evolve by accommodating a multitude of changes along the following, not mutually exclusive dimensions:

1. *Structural changes* focus on changes that occur on the service data types, messages, and operations, collectively known as service signatures.
2. *Behavioral changes* affect the business protocol of a service. Business protocols specify the external messaging and perceived behavior of services (viz. the rules that govern the service interaction between service providers and consumers) and, in particular, the conversations that the services can participate in.
3. *Policy-induced changes* describe changes in policy assertions and constraints on the invocation of the service. For instance, they express changes to the Quality of Service (QoS) characteristics and compliance requirements of a service.

Structural, behavioral, and QoS-related policy-induced changes refer to the externally observable aspects of a service (in terms of its signatures, protocols, etc.). These types of changes have a direct and profound impact on the service interfaces and as such they will be discussed extensively in the following sections. Changes due to legislative, regulatory, or operational requirements on the other hand are typically deep changes [2], and therefore outside the scope of this work.

The contribution of this paper is twofold:

- a language-independent, theoretically-backed approach which brings together structural, behavioral, and QoS-related service changes, and
- a rigorous formal framework, type-safety criteria and algorithms which control and delimit the evolution of services. The goal of this framework is to assist service developers in controlling and managing service changes in a uniform and consistent manner.

For this purpose, we develop a set of theories and models that unify different aspects of services (description, versioning, and compatibility). In particular, we provide a consistency theory for guaranteeing type-safety and correct versioning transitions so that previous clients can use a versioned service in a consistent manner.

The theoretical framework presented in this paper is at the cross section of programming language theories, service-oriented computing, and software engineering. It provides an innovative approach that challenges and redefines the state of the art in service evolution. At the same time, it calls into question the existing standards and support technologies as regards the facilities they provide for service change.

The rest of this paper is organized as follows: Section 2 discusses the background of our work by presenting a classification of existing service versioning approaches and their techniques for compatible service evolution. Section 3 presents a running example based on an industrial case study used throughout the rest of the paper. Section 4 formally defines the compatibility of services and presents our theoretical framework for the compatible evolution of services. Section 5 demonstrates the application of the framework to the empirical guidelines supporting the existing approaches and to the evolution scenarios of Section 3. In Section 6, we summarize our implementation effort. A qualitative evaluation and empirical validation of the framework is performed in Section 7. Finally, Sections 8 and 9 discuss related work, and conclusions and future work, respectively.

2 Background

By its definition, service evolution has two important facets: recording the continuous process of service development and controlling the consistency and unambiguity of its different versions. The following sections examine these two facets.

2.1 Service Versioning

Versioning as a concept has its roots in the *Software Configuration Management (SCM)* field that, together with type theory, has contributed in major ways to software maintenance and evolution [3]. From the aspects developed under SCM, of particular interest for service evolution is development support in terms of versioning as summarized in [4]. More specifically, versioning refers to the keeping of a historical record of software artifacts as they undergo change. The reliance of Service-Oriented Architecture (SOA) on the publishing of service interface descriptions (e.g., in WSDL 2.0[1]) and interaction protocols (in Abstract BPEL[2]), together with the predominant use of XML as the description language, adds an additional dimension to the versioning of services.

Versioning support during service development has two dimensions:

- *Interface versioning*. Versioning support for the *service description*, i.e., the artifacts that describe the interaction of the service with its environment (e.g., definitions of data types in XML Schema and WSDL and Abstract BPEL documents).
- *Implementation versioning*. Versioning support for the code, resources, configuration files, and documentation of a service.

Implementation versioning is by definition an SCM issue and as such the techniques from this domain can be applied to it (as discussed in [4]). Traditional SCM systems, such as the popular revision control systems CVS[3] and Subversion,[4] or their modern distributed counterparts like GIT[5] and Bazaar[6] (among others) can be used for this purpose. Service implementation is outside the scope of this work and as such it is not considered in the rest of this paper. In

1. Web Services Description Language (WSDL) Version 2.0 http://www.w3.org/TR/wsdl20.
2. Web Services Business Process Execution Language (BPEL) http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.
3. http://www.nongnu.org/cvs/.
4. http://subversion.apache.org/.
5. http://git-scm.com/.
6. http://bazaar-vcs.org/.

the following, we summarize the proposals of existing service interface versioning approaches.

2.1.1 Service Version Naming

Versioning approaches typically distinguish between (consumer) breaking and nonbreaking changes (cf., [5] and [6]). The former constitutes *major* releases and the latter *minor* ones. Naming a version usually follows the `Major#.Minor#` scheme where the sequential major release version number precedes the minor one; version "1.3," for example, denotes the third minor revision of the first major release (with "1.0" signifying the first version of that release). An alternative naming scheme incorporates a release date stamp instead of the sequence identifier [7].

Such naming schemes do not provide information about the position of the versions in the *version graph* which represents the relationships between versions [8]. In the VRESCo approach [9], the version graph is explicitly stored in the service registry, while in the WSDL and Universal Description Discovery and Integration (UDDI) extension approach of [10], the versioning graph can be reconstructed using the custom metadata of the annotated service description files.

2.1.2 Service Versioning Methods

Versioning in Web services is supported through the mechanisms offered by XML and XML Schema. More specifically, we can distinguish between the following service versioning methods:

1. New XML namespaces for each (major) version.
2. Version Identifiers (VIDs) unambiguously naming a version.
3. A combination of the above.

Approaches like [5], [10] that follow the new XML namespace technique purposely break the consumers of the service by assigning a different namespace to either the service itself or to its data types. The new namespace results in disrupting the binding of the service on the consumer side. New namespaces are therefore meant to be used only if a major version of a service is deployed. On the other hand, VIDs are used either as attributes (either in the root element of the document or in each element separately) [7], [6] or as part of the (endpoint) URL [11], [12]. This, however, requires the consumers to be somehow able to process the versioning information and understand the implications of the naming scheme for their application. Both types of approaches rely on the `Major#.Minor#` naming scheme either directly as a VID or by incorporating it into the namespace itself. They are not mutually exclusive and they can be used in conjunction for versioning control.

Some of the approaches to service interface versioning use a service registry mechanism like the Universal Description Discovery and Integration standard [13] or a custom registry [9] for storing and controlling the versioning information—either as an alternative or complementary to the XML-based techniques discussed above. For this purpose, they propose the addition of versioning metadata in the service description model that the registry is using.

2.1.3 Service Versioning Strategies

The versioning strategy varies depending on the goal of each approach with respect to the breaking or not of consumers. A number of approaches do not consider whether changes to a service version break the consumers of the service, e.g., [7] and [10]. As such, they leave to the developers the prerogative and responsibility of checking whether changes break the service consumers, but they also maintain a high degree of flexibility in the cases they can handle. In principle, these approaches allow for multiple versions of a single service to be accessible at a time.

The majority of existing approaches (e.g., [5], [12], [13]) propose a common compatibility-oriented strategy for versioning: maintain multiple active service versions for major releases, but cut maintenance costs by grouping all minor releases under the latest one. Nevertheless, the cost of maintenance varies in proportion to the number of active versions at a time. The creation of a major version therefore, apart from possibly breaking existing consumers, also increases the effort required for managing the service portfolio.

For this reason, approaches like [6] and [9] take special interest in discussing different *decommissioning* strategies for not current versions of the service. Despite differences in the details, the goal in each case is to decrease the number of active versions to the absolute minimum required to serve the service consumers. Usually, a grace period is given before decommissioning a service version and, depending on the change identification model used (see below), either the clients are notified in advance or they have to "discover" for themselves this information.

2.1.4 Change Identification Model

In a similar fashion to versioning strategies, the model of how service changes are perceived and identified by the service consumers varies according to the goals of the approach. This model can be classified in one (or more in the case of [14] and [12]) of the following categories: client, notification, and transparent.

In the *client* model [5], [6], [10], both nonbreaking and breaking changes result in a new version, and the identification of existence of this version is left to the consumer. The consumer is then required to adapt to the new version if necessary. In the *notification* model [13], the consumer is explicitly notified for the existence of a new version and asked to take action, usually within a given time period. This typically requires service consumers to subscribe to some sort of notification service. Finally, approaches that enforce nonbreaking changes do not have to inform their consumers of changes since in theory the changes are *transparent* to them. In reality, however, some of these approaches allow their clients to identify a new version using one or a variation of the methods above [15], [9].

2.1.5 Discussion

The investigation into existing service versioning approaches shows that the ease of use of VIDs in XML, in conjunction with the XML namespace disambiguation mechanism, is more than sufficient for recording and communicating the different versions of the service to its clients. In this respect, it is not necessary to provide additional methodologies for managing the versioning of services.

Since the goal of this work is to provide a service development approach that does not break the existing consumers, the emphasis is on enforcing the transparency of evolution. Major versions should be kept to a minimum

and minor versions should be bundled together under one major version in order to minimize the cost of maintenance and achieve a reasonable tradeoff between flexibility and constrained evolution. For this purpose, we provide a theoretical framework that allows for sound service evolution. The following sections assume that a robust versioning mechanism is already in place on the basis of which we explain how to analyze and evaluate the changes that lead to different versions of a service description.

2.2 Compatibility of Service Versions

Compatibility is a concept closely related to versioning. It is usually decomposed into backward and forward cases. A definition of forward and backward compatibility with respect to languages and message exchanges between producers and consumers is given in [16]. *Forward compatibility* means a new version of a message producer can be deployed without the need for updating the message consumer(s). *Backward compatibility* means a new version of a message consumer can be deployed without the need for updating the message producer. *Full* compatibility is the combination of both forward and backward compatibility.

The roles of message producers and consumers can be assigned in different ways for service providers and clients depending on the *message exchange pattern* they use (in WSDL 2.0 terms) and which defines the sequence and cardinality of abstract messages listed in an operation.

2.2.1 Forward Compatibility

Some of the approaches discussed in the previous section (in particular, [6], [15], and [14]) enforce forward compatibility through the use of extensibility. Extensibility is the property of a language to allow information that is not defined in the current version of the language; extensibility provides mappings from documents in any extended set to documents already defined [16]. Extensibility, therefore, is a relevant notion to both versioning and compatibility: whereas versions can be either compatible or incompatible (centralized) changes to the service, extensions are by definition compatible (decentralized) additions to an existing service. The underlying assumption in all cases is that the additional data can be safely ignored during the processing of a message, without any effect on the semantics of the message[7] [16].

2.2.2 Backward Compatibility

Backward compatibility is in practice a mechanism for distinguishing between major and minor releases—as long as the changes applied to a service lead to backward compatible versions of the service, they can be considered minor releases; otherwise, they are major. The usual approach for defining what constitutes a backward compatible change to a service has been to enumerate the possible compatible changes. This results to a list of permissible and prohibited changes usually, but not exclusively, to the WSDL document describing the service. This list reflects a combination of common sense, technological limitations, and empirical findings that results into a set of best practices—guidelines to be followed and not necessarily undisputed rules. These guidelines are presented in Table 1 and aggregate the guidelines from [5], [11], and [14].

TABLE 1
Guidelines for Backward Compatible Changes

Change	Backwards Compatible
Add (Optional) Message Data Types	Yes (input only)
Add (New) Operation	Yes
Modify Service Implementation[a]	Yes
Remove Operation	No
Modify Operation[b]	No
Modify Message Data Types	No

[a] *As long as it has no effect on the service interfaces.*
[b] *Includes renaming, changing parameters, parameter order, and message exchange pattern.*

All the changes are expressed in terms of changes to WSDL and XML Schema elements. In summary, the backward compatible changes are only additions of optional elements (either input data types or operations) or the modification of service implementation (as long as it does not affect the WSDL document). The removal or any kind of modification to an operation element is strictly prohibited, as is the modification of the message data types (with the exception of addition of optional data types).

This guideline-based approach is easily applicable and requires minimum support infrastructure and, for this reason, it is widely accepted. However, it exhibits certain disadvantages, the main of which is that it depends on service developers for deducing what is compatible and acting accordingly. Even if these rules are codified and embedded into a service development/versioning tool, as, for example, in the case of [17], they will always be limited by two factors: their dependency on technology (WSDL, in this case) and their lack of a solid theoretical foundation. In our approach, we extend the reasoning behind the backward compatible guidelines and we enhance it by showing how these rules can be generated as the result of a theoretical framework to control and delimit service evolution.

3 Running Example

In order to demonstrate the practical applicability of our work, we chose to use the industrial-strength Automotive Purchase Order Processing use case. The use case was developed in conjunction with IBM Almaden and is used as one of the validation scenarios in the S-Cube Network of Excellence[8] [18]. The scenario is based on the cross-industry, standardized Supply Chain Operations Reference (SCOR) model that provides abstract guidelines for building supply chains.[9] The SCOR model is comprised of four levels of processes (scope, configurations, business activities, and implementation, respectively).

This Automotive Purchase Order Processing use case is an example of how to realize SCOR level 3 activities using SOA-based processes for an enterprise in the automobile industry called Automobile Incorporation (a.k.a. AutoInc). AutoInc consists of different business units, e.g., Sales, Logistics, Manufacturing, etc., and collaborates with external partners

7. See [6] for an extensive discussion on the advantages and limitations of this assumption.

8. http://www.s-cube-network.eu/.
9. https://www.supply-chain.org/.

```
<definitions ...
targetNamespace="http://autoinc.com/POProcessing">
<types>
 <xsd:schema>
  <xsd:complexType name="PODocument">
  <xsd:sequence>
   <xsd:element name="OrderInfo" type="xsd:string"/>
   <xsd:element name="DeliveryInfo"
      type="xsd:string" minOccurs="0"/>
   <xsd:element name="TimeStamp" type="xsd:dateTime"/>
  </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="POAck">
   <xsd:element name="POStatus" type="xsd:string"/>
  </xsd:complexType>
 </xsd:schema>
</types>

<message name="POMessage">
 <part name="request" type="tns:PODocument"/>
</message>

<message name="POMessageAck">
 <part name="response" type="xsd:string"/>
</message>

<portType name="POServicePortType">
 <operation name="receivePO">
  <input name="poMessage" message="tns:POMessage"/>
 </operation>
</portType>

<!-- receivePOCallBack is used for the call back
invocation by the service. The operation is expected
to be implemented by the client. -->
<portType name="POServiceCallBackPortType">
 <operation name="receivePOCallBack">
  <output name="poCallBack" message="tns:POMessageAck"/>
 </operation>
</portType>
</definitions>
```

Fig. 1. POSERVICE definition in WSDL.

like suppliers, banks, and transport carriers. The use case describes a typical automobile ordering process, where customers can place automated orders with AutoInc. For a complete description of the use case, the reader is referred to [18]. In the following, we present one of the services that is part of the use case and discuss two possible evolution scenarios for it.

3.1 Purchase Order Processing Service (POSERVICE)

The Purchase Order Processing service supports the "receive purchase order" activity of the Sales unit of AutoInc. POSERVICE is at the core of the use case and has a critical function. In case of failure or underperformance, the whole chain of interlinked services in the use case will be detrimentally affected.

Fig. 1 contains the WSDL definition of the service. Starting from its port types, POSERVICE is communicating with its consumers in an asynchronous manner through the `receivePO` and `receivePOCallBack` operations. The actual protocol for communicating with the service is defined in BPEL. This is shown in Fig. 2, where a two-step interaction between POSERVICE and its consumers is explicitly defined. The consumer is supposed to invoke the `receivePO` operation with the Purchase Order document, codified by the `PODocument` data type, and wait for the call back invocation `receivePOCallBack` from the service side with the acknowledgment of the order receipt. The simple message payload for the operations of the service facilitates the demonstration of the theoretical constructs we develop.

```
<process ...
targetNamespace="http://autoinc.com/POProcessing">

 <import location="POService.wsdl"
  importType="http://schemas.xmlsoap.org/wsdl/"
  namespace="http://autoinc.com/POProcessing"/>

 ...

 <sequence>
  <receive name="ReceivePO" partnerLink="Client"
   portType="ns:POServicePortType"
   operation="receivePO" variable="PO"
   createInstance="yes"/>

  <!-- Process the purchase order message -->
  ...

  <invoke name="SubmitPOAck" partnerLink="Client"
   portType="ns:POServiceCallBackPortType"
   operation="receivePOCallBack" inputVariable="POAck"/>
 </sequence>
</process>
```

Fig. 2. POSERVICE definition in BPEL.

Finally, for the nonfunctional aspects of POSERVICE, and given the absence of a widely acceptable standard for the characterization of nonfunctional properties, we use the S-Cube Quality Reference Model (QRM) [19]. In particular, the QRM characteristics used for the definition of the POSERVICE nonfunctional properties are

- *Availability.* The degree of availability of the service to its consumers relative to a maximum availability of 24 hours, seven days a week.
- *Latency.* Time passed from the arrival of the service request until the end of its execution/service.
- *Performance.* The ability of a service to perform its required functions under stated conditions for a specified period of time. It is the overall measure of a service to maintain its service quality.

In a hypothetical case, we assume that latency is expected to vary between 0.15 and 0.3 seconds, availability to vary between 80 and 95 percent and performance to be at minimum 90 percent for the same conditions.

3.2 Evolution Scenarios

Since the goal of this work is to study the evolution of services, it is only natural to perceive POSERVICE itself as subject of change. In order to illustrate possible evolutionary paths that the service can take during its lifetime, we describe two *evolution scenarios*: a relatively simple service improvement scenario and a more complicated service redesign scenario.

3.2.1 Service Improvement Scenario

For this scenario, we assume that the service developers attempt to improve the QoS characteristics of the service by aiming for lower latency and focusing on less error-prone handling of incoming requests. For this purpose, a new version of the POSERVICE is designed, where:

- The customers, both new and returning, have to provide the delivery information along with the purchase order. In this way, the disruption caused by missing information in later stages of the process is minimized.

```
<xsd:complexType name="PODocument">
 <xsd:sequence>
  <xsd:element name="OrderInfo" type="xsd:string"/>
  <xsd:element name="DeliveryInfo" type="xsd:string"/>
  <xsd:element name="TimeStamp" type="xsd:dateTime"/>
 </xsd:sequence>
</xsd:complexType>
```

Fig. 3. POSERVICE WSDL—service improvement scenario (PODocument only).

- In order to streamline and accelerate the servicing time of each order, the service forward the delivery information to an intermediate service in the Logistics unit of AutoInc, in order to verify that the address does not contain an error and that it points to an existing delivery address.

The new version asks customers to always include in their requests the DeliveryInfo element, resulting in changing its multiplicity as shown in Fig. 3. The rest of the WSDL file remains as is. Furthermore, we assume the changes in the scenario result in changing the nonfunctional characteristics of the POSERVICE by introducing, for example, lower latency ranging from 0.075 to 0.15 secs, but worse performance of a minimum of 81 percent (due to additional service communication overheads).

3.2.2 Service Redesign Scenario

In this scenario, we assume a potential customer of POSERVICE requests to use a synchronous communication pattern with the service for application safety reasons. The service developers take the opportunity to also address some standing issues of the service and redesign it in two ways:

- In order to accommodate both existing consumers and the new customer, the service provides both synchronous and asynchronous interfaces. However, instead of running two versions of the service in parallel, both types of interaction with the service are grouped into one communication protocol. The new protocol allows consumers to decide which type of communication to use at its entry point. This allows for the synchronous interface to use the same message types as the asynchronous one since they carry the same payload.
- The time stamp on incoming messages is not necessary since it can be calculated by mining the service logs. However, all outgoing messages must carry time stamps for auditing purposes. The time stamp information is therefore removed from incoming messages and moved to outgoing messages.

These changes are bundled in one service redesign task with the goal to roll out a new version of the service as soon as possible. In particular, a new operation and wrapping port type for the synchronous communication is added to the WSDL of the service as shown in Fig. 4. receivePOSync reuses the same messages as its asynchronous counterpart receivePO in Fig. 1, taking into account the necessary modifications into the PODocument and POAck document types.

The simple sequence/receive set of BPEL activities in Fig. 2 is replaced by a pick activity that acts as a multiple option receive. Depending on whether the synchronous or asynchronous version of the operation is invoked, the appropriate response scheme is used (i.e., with a reply activity instead of the call back invocation for the synchronous part). These changes are depicted in Fig. 5.

```
<xsd:schema>
 <xsd:complexType name="PODocument">
  <xsd:sequence>
   <xsd:element name="OrderInfo" type="xsd:string"/>
   <xsd:element name="DeliveryInfo"
    type="xsd:string" minOccurs="0"/>
  </xsd:sequence>
 </xsd:complexType>
 <xsd:complexType name="POAck">
  <xsd:sequence>
   <xsd:element name="POStatus" type="xsd:string"/>
   <xsd:element name="TimeStamp" type="xsd:dateTime"/>
  </xsd:sequence>
 </xsd:complexType>
</xsd:schema>

<portType name="POServicePortType2">
 <operation name="receivePOSync">
  <input name="poMessage" message="tns:POMessage"/>
  <output name="poMessageAck" message="tns:POMessageAck"/>
 </operation>
</portType>
```

Fig. 4. POSERVICE WSDL—service redesign scenario.

While the redesign scenario also results in changes in the nonfunctional characteristics of POSERVICE, we will not consider them for purposes of simplifying the presentation.

4 COMPATIBLE EVOLUTION OF SERVICES

In the previous section, we discussed how existing works approach service evolution in terms of versioning and compatibility. Existing approaches use an informal notion of compatibility, relying on the empirical guidelines summarized in Table 1. In the following sections, we provide a classification of the different aspects of service compatibility before formalizing its definition using type theory.

4.1 Aspects of Compatibility

For the purposes of this discussion, we extend the definition for software components given in [20] and separate compatibility into two distinct dimensions:

```
<pick>
 <!-- If the asynchronous operation was invoked -->
 <onMessage partnerLink="Client"
   operation="receivePO"
  portType="ns:POServicePortType" variable="PO">

  <sequence>
   ...
   <invoke name="SubmitPOAck" partnerLink="Client"
    portType="ns:POServiceCallBackPortType"
    operation="receivePOCallBack" inputVariable="POAck"/>
  </sequence>
 </onMessage>

 <!-- If the synchronous operation was invoked -->
 <onMessage partnerLink="Client2"
  operation="receivePOSync"
  portType="ns:POServicePortType2" variable="PO">

  <sequence>
   ...
   <reply name="ReplyPOAck" partnerLink="Client2"
    portType="ns:POServicePortType2"
    operation="receivePOSync" variable="POAck"/>
  </sequence>
 </onMessage>
</pick>
```

Fig. 5. POSERVICE BPEL—service redesign scenario.

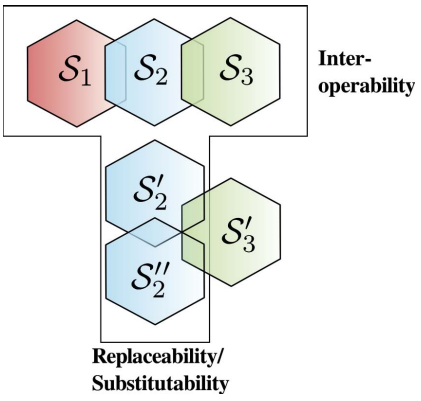


Fig. 6. Horizontal and vertical compatibility.

horizontal compatibility (or service interoperability) and vertical compatibility (or substitutability/replaceability). More specifically:

Definition 1. Horizontal compatibility *or* interoperability *of two services expresses the fact that the services can participate successfully in an interaction as service provider and service consumer.*

Horizontal compatibility is therefore a codependence relation between two interacting parties (services in the general case). The underlying assumption is that there is at least one *context* under which the two services can fulfill their roles. Context in this case is a configuration of the environment in terms of the execution state of both service producer and service consumer, along with the status of their resources, and for a particular message exchange history. This assumption is also implicit in the definition of the vertical dimension, and permeates all the definitions formal and informal that follow.

Definition 2. Vertical compatibility *or* substitutability *(from the provider's perspective) or* replaceability *(from the consumer's perspective) of service versions expresses the requirements that allow the replacement of one version by another in a given context.*

The combination of the two compatibility dimensions leads to the notion of *T-shaped changes* as depicted in Fig. 6. In the example of Fig. 6, overlapping hexagons denote compatible service versions. Service $\mathcal{S}_1$ is horizontally compatible with $\mathcal{S}_2$, meaning that $\mathcal{S}_1$ interoperates with $\mathcal{S}_2$—either as a consumer or a provider or both. Similarly, $\mathcal{S}_2$ is horizontally compatible with service $\mathcal{S}_3$. There exist two more versions of service $\mathcal{S}_2$ denoted by $\mathcal{S}'_2$ and $\mathcal{S}''_2$ that are vertically compatible with each other (and horizontally compatible with $\mathcal{S}'_3$) but incompatible with $\mathcal{S}_2$ as denoted by the gap between $\mathcal{S}_2$ and $\mathcal{S}'_2, \mathcal{S}''_2$. This signifies the existence of a major release of $\mathcal{S}_2$ (namely, $\mathcal{S}'_2$), which broke the interoperability of $\mathcal{S}_2$ with $\mathcal{S}_1$ and $\mathcal{S}_3$; $\mathcal{S}'_2$ was then replaced by a minor release (i.e., $\mathcal{S}''_2$).

The two dimensions are therefore interrelated: *substitutability and replaceability can be perceived as the property of preservation of interoperability for internalized changes to one or both of the interacting parties (producers or consumers)*. This enables referring simply to compatibility and denoting both aspects. If compatibility is achieved under *all* possible contexts, either on the vertical or the horizontal dimension, or both, then it is called *strict* substitutability/replaceability and interoperability or strict compatibility, respectively.

4.2 Formal Definition of Service Compatibility

For the formalization of the compatibility between any two services, we assume that each service (version) is denoted by a description schema $\mathcal{S}$ comprised of records s. Records represent the structural dependencies inside the service description using *elements* and their *relationships* as in [21], behavioral constraints in the form of *behavioral contracts* [22], and/or non-functional characteristics expressed as QoS properties [23] or dimensions [24].

Based on this assumption, we can take advantage of the existence of a *subtyping* relation that allows us to (partially) order different records based on their characteristics for defining compatibility. Subtyping allows us to check whether two records participate in a specialization/generalization relation and whether (under certain conditions that will be discussed later in this section) one record can replace another. *We will be using the notation $s \leq s'$ to denote that record s is a subtype of record s', irrespective of whether s is a structural, behavioral, or nonfunctional record.*

To formally define the compatibility of two service versions, we first define a distribution of the set $\mathcal{S}$ into two proper subsets $\mathcal{S}_{pro}$ and $\mathcal{S}_{req}$, representing the set of records for which the service acts as a producer and a consumer (of messages), respectively [25]:

Definition 3. *$\mathcal{S}_{pro}$ is the set of output-type records of a service description and $\mathcal{S}_{req}$ is the set of input-type records.*

Compatibility between service versions $\mathcal{S}$ and $\mathcal{S}'$ can be defined based on this distribution as follows:

Definition 4 (Service Compatibility). *We define three cases of compatibility:*

- *Forward.* $\mathcal{S} <_f \mathcal{S}' \Leftrightarrow \forall s \in \mathcal{S}_{pro}, \exists s' \in \mathcal{S}'_{pro}, s' \leq s$ *(covariance of output).*
- *Backward.* $\mathcal{S} <_b \mathcal{S}' \Leftrightarrow \forall s' \in \mathcal{S}'_{req}, \exists s \in \mathcal{S}_{req}, s \leq s'$ *(contravariance of input).*
- *Full.* $\mathcal{S} <_c \mathcal{S}' \Leftrightarrow \mathcal{S} <_f \mathcal{S}' \wedge \mathcal{S} <_b \mathcal{S}'$.

These definitions are in line with both traditional type theory and with the language-producing set-based theory proposed in [16]. Given the fact that the subset $\mathcal{S}_{pro}$ represents the language produced by the service, then this definition of forward compatibility is equivalent to the informal definition given in the Background section. Furthermore, armed with this definition, we can reason directly on new versions $\mathcal{S}', \mathcal{S}'', \mathcal{S}''', \ldots$ of the service, comparing them on a record to record basis for checking their compatibility. The following sections build the necessary components for such a reasoning. We start with an abstract model for describing services, on which we define the subtyping relation between service records. From there, we proceed by presenting an algorithm for checking the compatibility of services.

4.3 Service Description

In order to describe a service in our framework, we use the notion of *Abstract Service Descriptions (ASDs)* we introduced in [21] and which we refine here. An ASD represents a

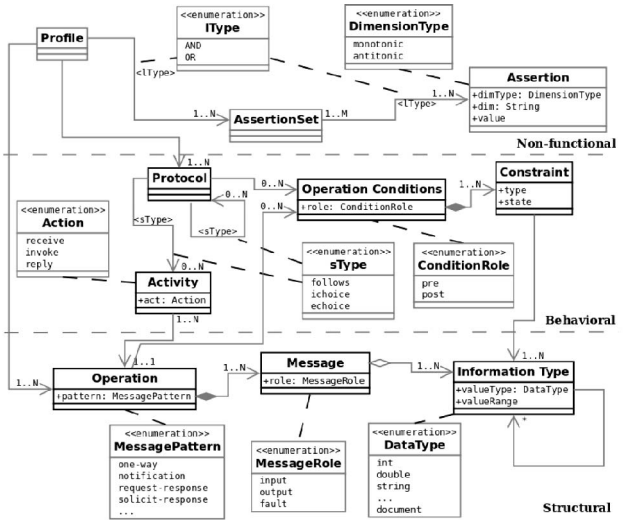


Fig. 7. The ASD metamodel.

particular version of a service and can be defined as the set of all its versioned records $\mathcal{S} := \{s_i^j\},\ i = 1, \ldots, N,\ j \in \mathcal{V}$, where $\mathcal{V}$ is the set containing the version identifiers vid for all records s. $\mathcal{S}$ therefore contains one particular version for each of its constituent records.

Each ASD respects a particular metamodel (Fig. 7). This metamodel divides constructs in three layers: a structural, a behavioral, and a nonfunctional layer. The essential characteristics of this metamodel can be found in metamodels for services description languages such as WSDL and BPEL [26] and in initiatives as the OASIS SOA Reference Architecture.[10] The ASD Metamodel aggregates information from these models and acts as a foundation from which all possible service descriptions can be generated or, alternatively, can be validated against, as discussed in [21].

4.3.1 Structural Layer

The structural layer of the ASD, as shown in the lower part of Fig. 7, contains an `Operation` and a `Message` concept corresponding to the WSDL operation and message constructs, respectively. `Information Type` is a wrapper for XML Schema complex types or elements that are used as parts of the message exchange. A (structural) ASD consists of *elements*—informational constructs representing the building blocks of the service—and their *relationships*—expressing the structural dependencies of elements.

Elements and their relationships are formally defined as

Definition 5. *An element e is a tuple $e := (name : string, (att_{i,i\geq 1} : attribute)^*, (pr_{j,j\geq 1} : property)^*)$. A relationship $r(e_s, e_t)$ between elements e_s (the* source *element) and e_t (the* target *element) is a tuple $r(e_s, e_t) := (name_s : string, name_t : string, rel : relation, mul : multiplicity)$, where*

- *$name, name_s, name_t$ are the unique element identifiers of elements e, e_s, e_t, respectively (of type string), e.g., $RequestMessage$.*
- *$(att_i, i = 1, \ldots, m)^*$ a set of zero or more generic types of attributes (int, char, string, etc.). Each attribute is assigned a value during the generation of an ASD from the metamodel. An example of an attribute is $Currency : String$, denoting the currency to be used in the scope of a specific message.*
- *$(pr_j, j = 1, \ldots, n)^*$ a set of zero or more property values, that is, attributes with predefined value ranges and characteristics. Properties contain information about the elements generated by the concept and belong to a property domain. The* `messagePattern` *to be used for an operation is an example of a property domain, containing properties like* `One-way`, `Request-Response`, *etc. The property domains for each element are depicted as enumerations in Fig. 7.*
- *rel is the type of relation between the elements (a, c, s—aggregation, composition, or association with the semantics defined in [21]).*
- *mul is the multiplicity of the relationship, defined as $mul := [min_{crd}, max_{crd}]$ where $min_{crd}, max_{crd} \in \mathbb{N}$ (the set of natural numbers) is the minimum and maximum, respectively, multiplicities allowed for each member of the relationship, as denoted in Fig. 7.*

For example, let's assume the WSDL definition of the POSERVICE as shown in Fig. 1. For the purchase order document and its wrapping message, we have elements *PODocument* and *POMessage*, generated by the ASD Metamodel concepts `Information Type` and `Message`, respectively

$$e_{pod} = (name = PODocument, valueType = document, valueRange = N/A),$$

i.e., there are no attributes, `valueType` is "document" and `valueRange` is undefined, and

$$e_m = (name = POMessage, role = input),$$

(as before). We can equivalently write these elements in shorthand notation as: $e_{pod} = (PODocument, document)$ and $e_m = (POMessage, input)$, respectively.

From the message schema of POSERVICE, the *PODocument must* contain exactly one order description item *OrderInfo* and one *TimeStamp* item, but it *may* contain one delivery description item *Delivery Info*. The respective elements ASD elements are $e_{oi} = (OrderInfo, string)$, $e_{ts} = (TimeStamp, dateTime)$, and $e_{di} = (DeliveryInfo, string)$, and the multiplicities of the relationships between e_m and e_{oi}, e_{ts}, e_{di} elements must be $[1,1]$, $[1,1]$, and $[0,1]$, respectively. The relationships of the e_{pod} element are therefore written in this notation as

$$r(e_{pod}, e_{oi}) = (name_s = PODocument, name_t = OrderInfo, rel = a, mul = [1,1]),$$

or, in shorthand,

$$r(e_{pod}, e_{oi}) = (PODocument, OrderInfo, a, [1,1]),$$
$$r(e_{pod}, e_{di}) = (PODocument, DeliveryInfo, a, [0,1]),$$
$$r(e_{pod}, e_{ts}) = (PODocument, TimeStamp, a, [1,1]).$$

Elements $e_m, e_{pod}, e_{oi}, e_{di}, e_{ts}$ and their relationships constitute the ASD representation of the `POMessage` data type in Fig. 1. Different versions of elements and relationships

10. http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/soa-ra.html.

can be represented by referring to the set $\mathcal{V}$ of all VIDs: $e^i, r^j, i, j \in \mathcal{V}$ denote versions i and j of element e and relationship r, respectively. In the following, we will simply write e' or r' as shorthand for $e^{i+k}, k \geq 1$ and $r^{j+m}, m \geq 1$.

4.3.2 Behavioral Layer

The behavioral layer contains the records describing the perceived behavior of the service in terms of *exchanges of messages* grouped under service operations and the *conditions under which message exchanges* may occur.

A number of different techniques have been proposed for describing and reasoning on the exchange of messages, such as business protocols based on finite state machines [27], [28] or deterministic finite automata [29], formal languages like TLA$^+$ [30], communication action schemas [31], workflows [32], automata [33], timed protocols [34], and Calculus of Communicating Systems (CCS)-like constructs [22]. A notation for the behavioral description of services (called *(behavioral) contracts*) has been proposed in [22], which is conceptually very similar to our approach. For this reason, we rely on that work for the definition of behavioral description and show how the necessary constructs for applying it are incorporated into our model.

Behavioral contracts σ under [22] use three operators: *continues with* "$.$", *external choice* "$+$", and *internal choice* "$\oplus$". The behavioral contract $\sigma_1 = a_1.a_2$ means that a_1 is followed by action a_2. $\sigma_2 = a_1 + a_2$ signifies that the external party (the service client) chooses which action to perform (a_1 or a_2 but not both), whereas for $\sigma_3 = a_1 \oplus a_2$, it is the service that decides which action is to be performed. Furthermore, actions are distinguished as input (to the service), denoted by a simple action a, and output type (from the service to the client) actions, denoted by barred actions $\overline{a}$. If not specified explicitly, it is assumed that an action can be either input or output type.

The ASD Metamodel in Fig. 7 contains the concepts `Protocol` (for behavioral contracts σ) and `Activity` (for actions a_i), and the stereotyped relation `sType` with possible names `follows`, `eChoice`, and `iChoice` corresponding to the operators ., +, and $\oplus$, respectively. `Activity` defines a specific type of action to be performed on the basis of an operation (in the same manner as BPEL simple activities). The protocol described in Fig. 2 maps to the behavioral contract expression $\sigma(e_{sequence}) = a_{ReceivePO}.\overline{a}_{SubmitPOAck}$, which in turn corresponds to the ASD elements and relationships:

$$
\begin{aligned}
&e_{sequence} = (sequence),\\
&e_{ReceivePO} = (ReceivePO, receive),\\
&e_{SubmitPOAck} = (SubmitPOAck, invoke),\\
&r(e_{sequence}, e_{ReceivePO})\\
&\quad = (sequence, ReceivePO, follows, [1,1]),\\
&r(e_{sequence}, e_{SubmitPOAck})\\
&\quad = (sequence, SubmitPOAck, follows, [1,1]).
\end{aligned}
$$

The expression above is the equivalent of Fig. 2 in ASD notation. *In a similar fashion, we can always map*[11] *a protocol e_{prt} and its relationships to other protocols $r(e_{prt}, e_{prt_i})$ or activities $r(e_{prt}, e_{act_i})$ to the respective behavioral contract $\sigma(e_{prt})$.*

Dealing with *operational pre and postconditions* to represent the conditions under which message exchanges can occur is more straightforward: we update the classic extension of behavioral specification by Liskov and Wing [36] and Meyer [37] that describe the behavior of an object in terms of pre- and postconditions. The conditions are expressed as relatively simple logical expressions like $pre.elems \neq \{\}$ denoting a nonempty list of input elements. For the purposes of this discussion, we will assume that these conditions are codified as groups of expressions that must be in a specific (Boolean) status. The ASD Metamodel contains for this purpose the following concepts: `Constraint` elements to allow the definition of specific conditions to be satisfied, and `Operation Conditions` to group them and define whether they are to be used as pre or postconditions for protocols or operations. `Operation Conditions` and `Constraint` elements and their relationships are covered by the discussion on the structural layer since they use only basic relationships (association and composition).

4.3.3 Nonfunctional Layer

The nonfunctional layer consists of QoS and policy constraints in the forms of assertions that are associated with evolving services. In a similar manner to the behavioral layer, we overload the semantics of the layer elements and their relationships. Since our approach depends on the ordering of the service description records, the remainder of this section will focus on *ordinal QoS dimensions* [24], i.e., QoS dimensions whose values can be ordered according to some predefined criteria.

More specifically, we adopt a simplified version of WS-Policy[12] for the description of QoS-related expressions. The concepts for these records and their property domains are depicted on the upper layer of Fig. 7. We assume that `Assertions`, containing statements about the acceptable and expected value ranges of ordinal QoS dimensions are organized by conjunctions or disjunctions into `AssertionSets`, grouped in turn as `Profiles`.

Using the ASD notation, an element of `Assertion` type e_{asrt} is a tuple $e_{asrt} := (name : string, dim : string, dimType : dimensiontype, value)$. For the values of the nonfunctional profile of the POSERVICE, for example, we have

$$
\begin{aligned}
&e_{assert_1} = (assert_1, availability, monotonic, [80, 95]),\\
&e_{assert_2} = (assert_2, latency, antitonic, [.15, .3]),\\
&e_{assert_3} = (assert_3, performance, monotonic, [90, 100]).
\end{aligned}
$$

Each dimension dim belongs to the `DimensionType` property domain denoting its behavior with respect to the ordering of its values. `Monotonic` dimensions order their values with increasing order; `Antitonic` order them in decreasing order. Availability values (a monotonic dimension) can be considered more general if they are higher, whereas response time values (an antitonic dimension) are more general if they are closer to 0. Elements of `Profile` and `AssertionSet` type act as anchors for the relationships between the assertions. Expressing the grouping of assertions under an assertion set (e.g., $aset_1 = assert_1 \wedge assert_2 \wedge assert_3$ in the example) is denoted in a similar manner to the behavioral layer as

11. For a more detailed discussion on the mapping between BPEL, behavioral contracts and the behavioral layer the reader is referred to [22] and to [35].

12. Web Services Policy Version 1.2 http://www.w3.org/Submission/WS-Policy/.

$$r(e_{aset_1}, e_{assert_1}) = (aset_1, assert_1, AND, [1,1]),$$
$$r(e_{aset_1}, e_{assert_2}) = (aset_1, assert_2, AND, [1,1]),$$
$$r(e_{aset_1}, e_{assert_3}) = (aset_1, assert_3, AND, [1,1]).$$

POSERVICE offers only one profile to its consumers e_{pfl_1}, containing exactly one assertion set e_{aset_1}

$$e_{pfl_1} = (pfl_1),$$
$$r(e_{pfl_1}, e_{aset_1}) = (pfl_1, aset_1, OR, [1,1])(by\ convention).$$

Combinations of disjunctions and conjunctions and more complex logical expressions can be denoted in the ASD notation in a similar manner.

4.4 Subtyping of ASD Records

4.4.1 Structural Subtyping

By their definition, each element and relationship are *types* themselves and we can compare two elements or relationships by extending the subtyping relation as follows:

Definition 6 ((Structural) Subtyping of Elements and Relationships).

- *For $e = (name, att_1, \ldots, att_k, pr_1, \ldots, pr_l)$ and $e' = (name', att'_1, \ldots, att'_m, pr'_1, \ldots, pr'_n)$, we define the subtype relation between e and e' as*

$$e \leq e' \Leftrightarrow name \equiv name' \wedge$$
$$k > m, att_i \leq att'_i, 1 \leq i \leq m \wedge$$
$$l > n, pr_j \leq pr'_j, 1 \leq j \leq n,$$

  that is, they have the same name identifier, and e' has less attributes and properties than e, but the ones it has are more generic (supertypes) of the respective attributes and properties of e. By definition it holds that $(e = \emptyset) \leq e'$.
- *For $r(e_s, e_t) = (name_s, name_t, rel, mul)$ and $r(e'_s, e'_t) = (name'_s, name'_t, rel', mul')$, we define the subtype relation between r and r' as*

$$r(e_s, e_t) \leq r(e'_s, e'_t) \Leftrightarrow e_s \leq e'_s \wedge e_t \leq e'_t \wedge rel = rel' \wedge mul \subseteq mul',$$

  that is, the elements e'_s, e'_t participating in the (new) relationship are supertypes of e_s, e_t (and therefore $name_s \equiv name'_s \wedge name_t \equiv name'_t$) and the multiplicity domain of the relationship is a superset of the respective one in the old relationship. We assume $\emptyset \leq r(e_s, e_t)$, iff $mul = [0, N)$, $N \geq 1$.

With respect to the property domains of Fig. 7, it holds that $int \leq double \leq string \leq document$ for `DataType` and

$$one - way \leq request - response,$$
$$notification \leq solicit - response$$

for `MessagePattern`. This allows us to modify not only the message payload but also the interaction protocol of the service operations under certain conditions that we discuss in the following. Both of these options are not allowed by the guidelines of Table 1.

Consider, for example, the new version of POSERVICE from the Service Improvement Scenario as shown in Fig. 3, which requires the delivery information to be obligatorily submitted together with the purchase order (instead of optionally as in the previous version in Fig. 1, as indicated by the `minOccurs=0` attribute). $r(e_{pod}, e_{di})$ is therefore replaced in the service description $\mathcal{S}'$ of the service by $r'(e_{pod}, e_{di}) = (PODocument, DeliveryInfo, a, [1,1])$ and it holds that $r'(e_{pod}, e_{di}) \leq r(e_{pod}, e_{di})$ since $mul' \subseteq mul$, that is, $r'(e_{pod}, e_{di})$ is a subtype of $r(e_{pod}, e_{di})$. This should be expected since an optional data type in the message schema is more generic than the same message schema with the data type as mandatory.

4.4.2 Behavioral Subtyping

In [22], the authors introduce a behavioral subcontracting relation $\preceq$ between behavioral contracts. A behavioral contract σ is called a behavioral subcontract of σ', that is, $\sigma \preceq \sigma'$ if it manifests less interaction capabilities than σ'. In [22], the authors present the description and proofs required for checking whether this relation holds given σ and σ'. During the presentation of the records of the behavioral layer, we showed how to map protocol elements to their respective behavioral contracts. Therefore, applying the subtyping relation and checking for compatibility between versions of elements in the behavioral layer is reduced to mapping them to the respective contracts and applying the subcontracting relation $\preceq$ between them. This is achieved by overloading the semantics of the subtyping relation for `Protocol` elements and adding the following condition in Definition 6:

$$e_{prt} \leq e'_{prt} \Leftrightarrow \sigma(e_{prt}) \preceq \sigma(e'_{prt}).$$

That is, elements of the `Protocol` type are mapped to their respective behavioral description and the subtyping check is performed in that formalism.

The addition of an option of *synchronous communication* mode to the input of POSERVICE initiated by the Redesign Scenario from Section 3, for example, results in a protocol that is a supertype of the initial protocol of the service. We have shown above that the protocol of the initial version of the service maps to the behavioral contract $\sigma(e_{sequence}) = a_{ReceivePO}.\overline{a_{SubmitPOAck}}$. The BPEL description of the POSERVICE in the Redesign Scenario (Fig. 5) is mapped to the ASD records

$$e'_{pick} = (pick), e_{seq_1} = (seq_1), e'_{seq_2} = (seq_2),$$
$$e_{ReceivePO} = (ReceivePO, receive),$$
$$e_{SubmitPOAck} = (SubmitPOAck, invoke),$$
$$r(e_{seq_1}, e_{ReceivePO}) = (seq_1, ReceivePO, follows, [1,1]),$$
$$r(e_{seq_1}, e_{SubmitPOAck}) = (seq_1, SubmitPOAck, follows, [1,1]),$$
$$e'_{ReceivePOSync} = (ReceivePOSync, receive),$$
$$e'_{ReplyPOAck} = (ReplyPOAck, reply),$$
$$r(e'_{seq_2}, e'_{ReceivePOSync}) = (seq_2, ReceivePOSync, follows, [1,1]),$$
$$r(e'_{seq_2}, e'_{ReplyPOAck}) = (seq_2, ReplyPOAck, follows, [1,1]),$$
$$r(e'_{pick}, e_{seq_1}) = (pick, seq_1, eChoice, [1,1]),$$
$$r(e'_{pick}, e'_{seq_2}) = (pick, seq_2, eChoice, [1,1]).$$

The equivalent expression in behavioral contract notation is

$$\sigma(e'_{pick}) = (a_{ReceivePO}.\overline{a_{SubmitPOAck}}) + (a_{ReceivePOSync}.\overline{a_{ReplyPOAck}}),$$

from which it can be seen that $\sigma(e_{sequence}) \preceq \sigma(e'_{pick})$ since $\sigma(e'_{pick})$ contains $\sigma(e_{sequence})$ and allows for further interactions. Therefore, and according to the extension of Definition 6 we introduced, we can conclude that $e_{sequence} \leq e'_{pick}$. This means that a client that works with protocol $e_{sequence}$ can also work normally with protocol e'_{pick}.

Reasoning on the `Operation Conditions` and `Constraint` elements and their relationships is sufficiently covered by Definition 6. Adding new `Constraints` $e'_{con_i} = (con_i, expression_i, true)$ and $e'_{con_j} = (con_j, expression_j, true)$ to an existing `Operation Conditions` element $e_{opcon} = (opcon, pre)$, for example, creates additional relationships $r(e_{opcon}, e'_{con_i}) = (opcon, con_i, c, [1,1])$ and

$$r(e_{opcon}, e'_{con_j}) = (opcon, con_j, c, [1,1])$$

for which we know that $(r(e_{opcon}, e_{con_i}) = \emptyset) \not\leq r(e_{opcon}, e'_{con_i})$ and $(r(e_{opcon}, e_{con_j}) = \emptyset) \not\leq r(e_{opcon}, e'_{con_j})$ since $e_{opcon} \neq \emptyset \wedge [1,1] \neq [0,N]$, $N \geq 1$.

4.4.3 Nonfunctional Subtyping

Extending the subtyping relation as defined in Definition 6 in the model of nonfunctional description we introduced requires two things: providing operators for ordering the value ranges for each assertion element with respect to how general/specific they are, and handling the special semantics of disjunctions and conjunctions. For the former, we base the ordering of assertions on the nature of their dimension and we use the relations already defined in Allen's Interval Algebra [38] for relatively positioning intervals (here value ranges) on a dimension. For the latter, we use the simple observation that an assertion set with more conjunctions is more restrictive (i.e., more specific) than one with less, while the reverse is true for disjunctions.

Definition 6 therefore needs to be supplemented by the conditions

- $e_{asrt} \leq e'_{asrt} \Leftrightarrow name \equiv name' \wedge dim \equiv dim' \wedge v \leq v'$ with

  $$v \leq v' \Leftrightarrow \begin{cases} v\{=, <, \mathrm{s}, \mathrm{fi}, \mathrm{m}, \mathrm{o}\}v' \text{ (monotonic dimensions)} \\ v\{=, >, \mathrm{f}, \mathrm{si}, \mathrm{mi}, \mathrm{oi}\}v' \text{ (antitonic dimensions)} \end{cases}$$

  where the relations have the following semantics:

  - $=$ *v has equal length* (and overlaps totally) with v',
  - s *v starts* together with v' but finishes before it,
  - f v starts after v' but it *finishes* together with it,
  - m *v meets* v' at its finishing point (v finishes when v' starts),
  - o v partially *overlaps* with v'—having started before v',
  - $<$ *v takes place before* v'.

  The inversions $\mathrm{si}, \mathrm{fi}, \mathrm{mi}, \mathrm{oi}, >$ signify that the roles of v and v' are reversed ($v > v'$, for example, means that v' takes place before v, etc.).

- $\emptyset \leq r(e_s, e_t, OR, mul)$ and $r(e_s, e_t, AND, mul) \leq \emptyset$.

The first addition expresses the ordering of values based on their monotonic or antitonic nature. More generic means extending the value range toward the direction that is considered "better," allowing even for partial or no overlaps, provided that the range moves toward better values. The second addition formalizes the notion that the more options exist for a profile the more generic it is; the more conditions to be satisfied on the other hand in each profile, the more specific it is.

For the QoS characteristics of POSERVICE after the Service Improvement Scenario, for example, we have in ASD notation the records

$$e'_{assert_2} = (assert_2, latency, antitonic, [.075, .15]),$$
$$e'_{assert_3} = (assert_3, performance, monotonic, [81, 100]),$$
$$e'_{aset_1} = (aset_1), e'_{pfl_1} = (pfl_1),$$
$$r(e'_{aset_1}, e'_{assert_2}) = (aset_1, assert_2, AND, [1,1]),$$
$$r(e'_{aset_1}, e'_{assert_3}) = (aset_1, assert_3, AND, [1,1]),$$
$$r(e'_{pfl_1}, e'_{aset_1}) = (pfl_1, aset_1, OR, [1,1]).$$

By their definition, latency is an antitonic dimension (lower values are better) and performance is monotonic (the closer to 100 percent, the better), and we have

- $assert_2 = assert_2 \wedge latency = latency \wedge antitonic = antitonic \wedge ([.15, .3] \text{ mi } [.075, .15])$,
- $assert_3 = assert_3 \wedge performance = performance \wedge monotonic = monotonic \wedge ([81, 100] \text{ fi } [90, 100])$.

From the extension of Definition 6 for nonfunctional elements, we have $e_{assert_2} \leq e'_{assert_2}$ and $e'_{assert_3} \leq e_{assert_3}$ (with e_{assert_1} remaining unchanged). These results correspond to the observation that a client that accepts latency between 0.15 and 0.3 seconds can also accept the more favorable latency between 0.075 and 0.15 seconds. In contrast, a client that expects performance between 90 and 100 percent cannot accept a performace that ranges between 81 and 100 percent since this may result in performance below the acceptable lower limit (90 percent). Changing the performance of the service from 81-100 percent to 90-100 percent on the other hand is acceptable (for the client).

4.5 Reasoning on Service Evolution

Having established a type theory for all the layers of an ASD, it becomes possible to use the subtyping relation of ASD records to check for the compatibility of service versions. Reasoning about this decision is quite straightforward: By combining Definitions 4 and 6 (as extended accordingly for each layer), we can check whether both cases of compatibility are satisfied using the definition of subtyping for ASDs. The following sections discuss how this can be achieved.

4.5.1 T-Shaped Changes

We informally define the concept of *T-shaped changes* as the set of changes that respect service compatibility. We use three fundamental operators to describe the changes occurring to service descriptions: add for the addition of record, del for the removal of a record, and mod for the modification of the record (addition/removal of attributes

or properties and so on). Combinations of these fundamental operators can be further used to produce more advanced operators like the renaming of a record. By applying these operators to an ASD $\mathcal{S}$ and for a record s, we get the respective *change primitives*: $add(s, \mathcal{S})$, $del(s, \mathcal{S})$, and $mod(s, \mathcal{S})$. Depending on whether s is an element or a relationship, the change primitives are expanded accordingly: $add(e, \mathcal{S})$ and $add(r(e_s, e_t), \mathcal{S})$, for example, signify the addition of an element e or a relationship $r(e_s, e_t)$ to $\mathcal{S}$, respectively.

The evolution of services rarely occurs in simple increments and usually encompasses a number of changes to the service description that occur simultaneously. For this reason, we define a *change set* as the fundamental degree of change to a service description:

Definition 7. *A change set $\Delta\mathcal{S}$ is a set of change primitives*

$$\Delta\mathcal{S} := \{operator(s_i, \mathcal{S}) | operator \in \{add, del, mod\}\},$$

that when applied to an ASD $\mathcal{S}$ results in a new version of the service $\mathcal{S}'$, signified by $\mathcal{S}' = \mathcal{S} \circ \Delta\mathcal{S}$.

Versions of services can therefore be expressed in terms of the change sets that are required for reconstructing a version from a baseline (original) version, following the conventions of SCM. For the purposes of this discussion, we assume that the change sets between ASDs are recorded during the development of the services. If not, then the application of an algorithm like the one presented in [39] could generate them. We can classify change sets with respect to compatibility as

Definition 8 (T-shaped Changes). *A change set $\Delta\mathcal{S}$ is called* T-shaped *iff, when applied to a service description $\mathcal{S}$, it results in a fully compatible service description $\mathcal{S}' = \mathcal{S} \circ \Delta S$, that is, $\mathcal{S} <_c \mathcal{S}'$ (using Definition 4).*

The term "T-shaped change" refers to the relation between the two aspects of compatibility as illustrated in Fig. 6. As long as a change set $\Delta\mathcal{S}$ results in a horizontally or vertically compatible (or both) version of a service, then it belongs to the set $\mathbb{T}$ of all possible T-shaped changes. *Constraining the evolution of services is therefore reduced to checking $\Delta\mathcal{S} \in \mathbb{T}$.*

4.5.2 Checking for Compatibility

Reasoning on a change set is performed in two steps:

- Distribution of the elements of $\mathcal{S}$ in sets $\mathcal{S}_{pro}$ and $\mathcal{S}_{req}$ sets using Definition 3.
- Checking whether the change set is T-shaped or not using Definition 8.

For the creation of the $\mathcal{S}_{pro}$ and $\mathcal{S}_{req}$ sets, we initially select all elements of input or output type in Fig. 7, starting with elements like `Messages`. Then, by taking advantage of the relationships between elements in Fig. 7, we propagate this property to all elements connected to them, following the direction of the arrow of the relationship. Then, we "mark" both the relationship and the connected element with the same type (input or output) and we continue this process until there are no more relationships to traverse. This way, we construct the $\mathcal{S}_{pro}$ and $\mathcal{S}_{req}$ sets, which contain

- $\mathcal{S}_{pro}$. `Message` elements with property value `role=output` or `fault` and all `Information Type` elements that are in an aggregation relationship with them, together with the respective relationships. `Activity` elements with property `act=invoke` or `act=reply`. `Operation Conditions` elements with property value `role=post` and all `Constraint` elements that are in a composition relationship with them, together with the respective relationships.
- $\mathcal{S}_{req}$. `Message` elements with property value `role=input` and all `Information Type` elements that are in an aggregation relationship with them, together with the respective relationships. `Activity` elements with property `act=receive`. `Operation Conditions` elements with property value `role=pre` and all `Constraint` elements that are in a composition relationship with them, together with the respective relationships.

`Protocol` and `Profile` records are distributed by convention to the $\mathcal{S}_{req}$ set since they refer to the input aspect of the interaction of the consumer with the service. Determining if a change set is T-shaped is performed by using the *Compatibility Checking Function (CCF)* (Listing 1).

Listing 1. $\mathrm{CCF}(\mathcal{S}, \mathcal{S}')$

```
 1: for all s' ∈ S'_req do
 2:     if ∄s ∈ S_req, s ≤ s' then
 3:         return false;
 4:     end if
 5: end for
 6: for all s ∈ S_pro do
 7:     if ∄s' ∈ S'_pro, s' ≤ s then
 8:         return false;
 9:     end if
10: end for
11: return true;
```

The two steps of CCF correspond to the two legs of Definition 4: lines 1 to 5 to covariance of input and lines 6 to 10 to contravariance of output. For this purpose, we use the definition of subtyping for records in different layers of the ASD as discussed in the previous. If both checks pass successfully, then the function returns *true*; in any other case, it returns *false*. The following section provides a series of examples of how the reasoning on T-shaped change sets is performed in practice.

5 Application of the Theoretical Framework

In this section, we demonstrate how our theoretical compatible service evolution framework can be applied by (theoretically) validating and extending the empirical guidelines for backward compatibility (Table 1), and revisiting the evolution scenarios defined in Section 3.

5.1 T-Shaped Change Patterns

Table 2 contains a number of patterns of change sets, some of which correspond to the backward-compatibility preservation guidelines in Table 1. It also contains an indication of whether each pattern of change set is T-shaped or not,

TABLE 2
Patterns of Change Sets

Pattern	T-shaped Change	Corresponding Guideline[a]
$\Delta\mathcal{S}_{P1} = \{add(it', \mathcal{S}), add((msg_i, it'), \mathcal{S})\}$, $r(msg_i, it') = \{msg_i.name, it'.name, a, mul\}$, $mul = [0, N), N > 0$	Yes, if $msg_i \in \mathcal{S}_{pro}$ then there is no violation of covariance; if $msg_i \in \mathcal{S}_{req}$ then it holds by definition that $\emptyset \leq r(msg_i, it')$.	Add (Optional) Message Data Types
$\Delta\mathcal{S}_{P2} = \{add(op', \mathcal{S}), add((op', msg_i), \mathcal{S})\} \vee \{add(op', \mathcal{S}), add(msg', \mathcal{S}), add((op', msg'), \mathcal{S}), \dots\}$	Yes, reasoning in a similar fashion as above.	Add (New) Operation
$\Delta\mathcal{S}_{P3} = \{del(op_i, \mathcal{S}), del((op_i, msg_{i,j}), \mathcal{S}), \dots\}$	No, if $\exists j, msg_{i,j} \in \mathcal{S}_{pro}$ due to covariance; Yes, otherwise[b]	(*Remove Operation*)
$\Delta\mathcal{S}_{P4} = \{mod(op_i, \mathcal{S})\} \vee \{mod((op_i, msg_{i,j}), \mathcal{S})\}, r'(op_i, msg_{i,j}) = \{\dots, mul'\}$	Yes, if $mul \subseteq mul' \wedge msg_{i,j} \in \mathcal{S}_{req}$ (contravariance) or $mul' \subseteq mul \wedge msg_{i,j} \in \mathcal{S}_{pro}$ (covariance); No, otherwise.	(*Modify Operation*)
$\Delta\mathcal{S}_{P5} = \{mod(it_i, \mathcal{S})\} \vee \{mod((it_i, it_{i,j}), \mathcal{S})\}, r'(it_i, it_{i,j}) = \{\dots, mul'\}$	Yes, if $mul \subseteq mul' \wedge it_i, it_{i,j} \in \mathcal{S}_{req}$ (contravariance) or $mul' \subseteq mul \wedge it_i, it_{i,j} \in \mathcal{S}_{pro}$ (covariance); No, otherwise.	(*Modify Message Data Types*)
$\Delta\mathcal{S}_{P6} = \{add(it', \mathcal{S}), add((msg_i, it'), \mathcal{S})\}$, $r(msg_i, it') = \{msg_i.name, it'.name, a, mul\}$, $mul = [M, N), 0 < M < N$	Yes, iff $msg_i \in \mathcal{S}_{pro}$ (covariance); No, for all other cases.	*Add Mandatory Data Types*
$\Delta\mathcal{S}_{P7} = \{del(it_i, \mathcal{S}), del((it_j, msg_{j,i}), \mathcal{S}), \dots\}$	Yes, iff $it_i \in \mathcal{S}_{req}$ (contravariance); No for all other cases.	*Remove Data Types*

[a] *Referring to Table 1; guidelines in italics are either new or not originally allowed.*
[b] *That is, a one-way operation, for example, can be deprecated without an effect on the consumer—the service can just ignore the incoming message.*

together with an explanation of the reasoning that led to this conclusion.[13] More specifically:

$\Delta\mathcal{S}_{P1}$ (which corresponds to the guideline of adding optional message data types) is T-shaped, irrespective of whether the data types (represented by an *it* element) are added to a message that belongs to the provided or required set. This is because if it is the former case, then it does not affect the second step of CCF; if it is the latter case, then due to the fact that an optional relationship (with minimum multiplicity 0) is a supertype of the "empty" relationship by definition and given that the rest of $\mathcal{S}$ remains unaffected, then it also passes the first step. $\Delta\mathcal{S}_{P2}$ is also T-shaped under all cases, following a similar reasoning.

$\Delta\mathcal{S}_{P3}$ on the other hand is T-shaped only if the deleted operation has exclusively input messages and *under the assumption that these messages can be ignored without affecting either the producer or the consumer*. The respective guideline explicitly forbids this change set by being too conservative for the sake of safety. Our approach shows that such a modification to a service would not necessarily break existing consumers. If the receipt of the message is part of a larger communication protocol though, then this change set may not be T-shaped due to the respective constraints on the behavioral layer. Such cases should be handled with care.

$\Delta\mathcal{S}_{P4}$ and $\Delta\mathcal{S}_{P5}$ work in a different manner. They allow for flexible input messages and associated data types by allowing a more general multiplicity domain in their relationship. This implies that the service can accept more incoming messages or a wider message payload than before. They also restrict the output messages accordingly.

$\Delta\mathcal{S}_{P6}$ and $\Delta\mathcal{S}_{P7}$ accept as T-shaped the addition of a nonoptional data type to an output message and the removal of a message data type from input messages. The reasoning is the same as in the case of $\Delta\mathcal{S}_{P3}$: as long as the consumer or the producer, respectively, can ignore the "additional" message payload, then the compatibility is preserved. Further T-shaped change sets can be generated in a similar fashion.

The set of T-shaped change sets that can be produced in an incremental way is therefore a superset of the guidelines-based one in Table 1. Enumerating all possible T-shaped change sets for all layers of service description, even by starting with a relatively simple metamodel as that of Fig. 7, is too lengthy of a process to be presented here and defeats the purpose of the framework. Nevertheless, if necessary or desired, it is shown that this process is feasible.

5.2 Evolution Scenarios Revisited

The following section revisits the evolution scenarios defined in Section 3 and discusses if the changes they are proposing are T-shaped or not. The purpose of this discussion is to further illustrate our proposal and demonstrate the effect of each scenario to service developers and consumers.

5.2.1 Service Improvement Scenario

This scenario results in changes in both the structural and nonfunctional layers of POSERVICE. More specifically and as discussed throughout the previous sections, $\mathcal{S}'$ differs from $\mathcal{S}$ by

$$r'(e_{pod}, e_{di}) = (PODocument, DeliveryInfo, s, [1, 1]),$$
$$e'_{aset_1} = (aset_1), e'_{pfl_1} = (pfl_1),$$
$$r(e'_{aset_1}, e_{assert_1}) = (aset_1, assert_1, AND, [1, 1]),$$
$$r(e'_{aset_1}, e'_{assert_2}) = (aset_1, assert_2, AND, [1, 1]),$$
$$r(e'_{aset_1}, e'_{assert_3}) = (aset_1, assert_3, AND, [1, 1]),$$
$$r(e'_{pfl_1}, e'_{aset_1}) = (pfl_1, aset_1, OR, [1, 1]).$$

13. In the table and the following discussion, we will write *msg* denoting a `Message` element, and *it* for `Information Type`, and *op* for `Operation` elements, respectively.

We have previously established that

- $r'(e_{pod}, e_{di}) \leq r(e_{pod}, e_{di})$, with $r(e_{pod}, e_{di}) \in \mathcal{S}_{req}$ and $r'(e_{pod}, e_{di}) \in \mathcal{S}'_{req}$.
- $e_{pfl_1} \not\leq e'_{pfl_1}$ since $e'_{assert_3} \leq e_{assert_3} \Rightarrow e_{aset_1} \not\leq e'_{aset_1}$.

By combining the above, we find the change set $\Delta\mathcal{S}_{SIS}$ required by the scenario not to be T-shaped: The first step of CCF is violated since $r'(e_{pod}, e_{di}) \leq r(e_{pod}, e_{di})$ and $e_{pfl_1} \not\leq e'_{pfl_1}$. In service versioning terms, this signifies the need for the creation of a major version of the service, requiring the consumers of POSERVICE to adapt or migrate to the new version.

This scenario illustrates the case for shallow changes: By trying to minimize errors and improve the service, the service developers unintentionally generated additional development effort for the service consumers. While this cost may appear small, it is impossible to predict the actual impact of such a change for service-based applications (SBA) consuming the service. Furthermore, it has to be considered that the creation of the new version of POSERVICE must be accompanied by the execution of an appropriate decommissioning plan for the existing version to facilitate the transition to the new version (as discussed in Section 2). This plan comes with additional costs in communicating the change to the consumers and running two active versions of the service (and their supporting implementation) in parallel for the transitional period. The costs of implementing the Service Improvement Scenario therefore may outweigh its benefits and in this case it has to reconsidered.

5.2.2 Service Redesign Scenario

This scenario has two major effects on the POSERVICE: It changes its interaction protocol by replacing the simple sequence with a pick activity (adding a new operation to the structural description to support the additional entry point) and it also modifies the incoming and outgoing messages. We previously showed during the discussion on behavioral subtyping that $e_{sequence} \leq e'_{pick}$ and thus passes the first step of CCF. We therefore only have to check the records of the structural layer.

With respect to the (new) input and output messages, as shown in Fig. 4, the new ASD $\mathcal{S}'$ differs from the previous version by

- the removal of the relationship $r(e_{pod}, e_{ts}) = (PODocument, TimeStamp, a, [1, 1])$,
- the addition of the relationship $r(e'_{poack}, e_{ts}) = (POAck, TimeStamp, a, [1, 1])$.

Since $r(e_{pod}, e_{ts}) \notin \mathcal{S}'_{req}$ and the rest of `Information Type` elements connected to e_{pod} remain unaffected, then all records $s' \in \mathcal{S}'_{req}$ pass the first step of CCF. The second step of CCF operates on all elements of $\mathcal{S}_{pro}$, for all elements of which we can see that $s' \leq s$ (since $r(e'_{poack}, e_{ts})$ does not appear in $\mathcal{S}_{pro}$) and therefore this step passes too.

Furthermore, the addition of the `receivePOSync` operation to the WSDL of the service depicted in Fig. 4 is mapped in ASD notation to the addition of element $e'_{recsync}$ to $\mathcal{S}'$, together with its (structural) relationships $r(e'_{recsync}, e_{msg})$ and $r(e'_{recsync}, e_{msgack})$ to the existing `POMessage` and `POMessagesAck` messages, respectively. In addition, the elements $e'_{pick}, e'_{seq_1}, e'_{seq_2}, e'_{ReceivePOSync}, e'_{ReplyPOAck}$ and the respective relationships have to be added, and the $e_{sequence}$ to be removed and replaced by e_{pick}. For these elements, it holds $e'_{recsync}, e'_{ReceivePOSync} \in \mathcal{S}'_{req}$ and $e'_{ReplyPOAck} \in \mathcal{S}'_{pro}$. Using a similar reasoning as above, we can see that CCF returns successfully with *true* and therefore $\Delta\mathcal{S}_{SRS} \in \mathbb{T}$.

In contrast to the Improvement Scenario, the Redesign Scenario is actually *shallow*. This means that the new version of POSERVICE $\mathcal{S}'$ can be implemented and deployed by replacing the previous version *without any effect to existing consumers*. Both new (using the synchronous communication capability) and old (using the asynchronous one) consumers can interact with the service using the same service interfaces. No particular decommissioning plan is necessary, and no additional costs (further than the development of the new service) are required. Being able to reason that the Redesign Scenario is T-shaped therefore guarantees that the effort and impact of implementing the change is minimum.

It has to be noted that using only the backward-compatibility preservation guidelines, summarized in Table 1, service developers would not be able to decide on the compatibility of the new version produced by the scenario since no behavioral aspects are covered by the guidelines. Based exclusively on the structural layer, they would conclude the new version is not compatible since no modification of incoming and outgoing messages is allowed (other than the addition of optional elements). In the following sections, we will focus on this divergence by investigating its roots and proposing specific solutions.

6 Implementation

In order to demonstrate the efficacy and practicality of our approach, we performed a proof-of-concept implementation. Using this implementation, we can empirically validate our proposal in a controlled setting through a case study. In the following sections, we discuss the technologies, methods, and outcomes of these procedures.

For the implementation of our proposal, we used widely supported and open source tools. The resulting *Service Representation Modeler (SRM)* prototype[14] provides two key facilities required for the empirical validation of our proposal [40]: a graphical editor for defining ASD models of service versions, and a reasoning module that compares two ASD models and checks them for compatibility as discussed in the previous section. A high-level flowchart view of the architecture of the SRM prototype is shown in Fig. 8. More specifically, service description versions (i.e., WSDL, BPEL, and WS-Policy documents) are converted into ASD models by the model transforming capability of the graphical editor and then given as input to the reasoning module. The reasoning module uses CCF in order to check their compatibility and includes the results of this check into a report. This compatibility report is then returned to the graphical editor for visualization.

We developed the SRM prototype as a plug-in for the Eclipse platform.[15] The first step for the implementation

14. Available at http://srmod.wordpress.com/.
15. http://www.eclipse.org/.

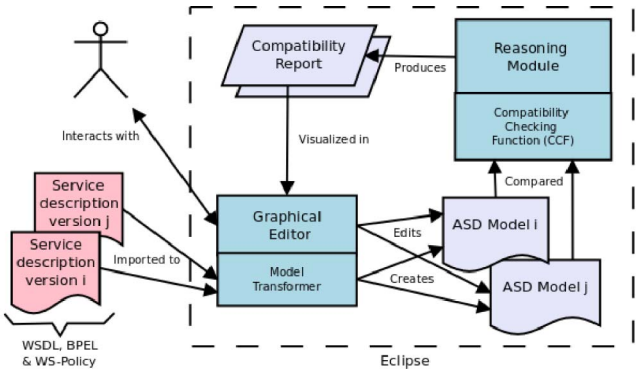


Fig. 8. SRM prototype architecture.

consisted of the definition of the metamodel to be used for the representation of the services in the prototype. For this purpose, we used the bottom layer of the ASD Metamodel (Fig. 7). The various elements and relationships of the structural layer were encoded in the Emfatic language (part of the EMF Eclipse plug-in [41]). We also annotated the Emfatic specification of the metamodel with GMF-specific instructions. We then used the injection facilities of the Epsilon plug-in[16] to convert the Emfatic specification of the metamodel into an ecore-type metamodel. Using this ecore metamodel, we automatically generated the Java code required for the graphical editor and the reasoning module. Fig. 9 shows an example of both functionalities in action.

The top part of the figure shows the ASD model for one of the versions of the POSERVICE loaded in the graphical editor of the SRM prototype. The editor provides the tools for creating and modifying a graphical representation of ASD models. It achieves this by offering a *Palette* panel (on the upper right part of Fig. 9), which contains widgets corresponding to the various structural records in the ASD Metamodel. Selecting one of these widgets and pointing in the white canvas area adds the respective element to the ASD model. Adding or modifying the name, properties, relationships, and the $\mathcal{S}_{pro}/\mathcal{S}_{req}$ distribution (according to Definition 3) of the records is done through the *Properties* perspective (at the bottom of Fig. 9).

The reasoning module of the SRM tool was implemented as a fully functional Epsilon program. We started by translating the CCF into a set of rules for records of the structural layer as follows:

$$\begin{aligned} op \leq op' &\Leftrightarrow name = name' \\ &\quad \wedge messagePattern = messagePattern' \\ msg \leq msg' &\Leftrightarrow name = name' \wedge role = role' \\ r(op, msg) \leq r'(op', msg') &\Leftrightarrow op \leq op' \\ &\quad \wedge msg \leq msg' \wedge mul \subseteq mul' \\ &\ldots \end{aligned}$$

This unrolling of the rules allowed us to encode the CCF in a straightforward manner and provide it as a module of the SRM prototype. The module takes as input two ASD models and compares them, checking for compatibility as shown in Fig. 9. Currently, the results are returned in the Epsilon console perspective inside Eclipse but we are working on exporting them in XML format and visualizing them using the graphical editor.

Fig. 9 shows the results of such a comparison between two ASDs, checking for compatibility on a record-per-record basis. If all checks are successful then the reasoner concludes with a *true*; otherwise, it returns *false*. In the particular case, the reasoning module returned a *false* since an `Information Type` to `Information Type` relationship was found to violate the CCF.

7 Evaluation

Providing service developers with the means to control the evolution of services belongs conceptually to design science. As such, the evaluation of this work is performed along the lines of requirements for effective design science research. For this purpose, we use the guidelines proposed in [42].

In particular, the utility, quality, and efficacy of the proposed framework for the compatible evolution of services were evaluated in the previous sections using descriptive methods. Due to the nature of our approach, which combines theoretical with empirical aspects, we evaluated the design of our approach using the scenarios driven from the industrial case of the POSERVICE. In the previous sections, we demonstrated the impact of changes with varying complexity to the consumers of POSERVICE, describing how to avert consumer disruption through the application of the service compatibility theory. The proposed method of controlling service compatibility was compared and contrasted with existing approaches in Section 5. We concluded that our approach by far covers and improves results of existing service evolution approaches and therefore our *contribution* is significant.

7.1 Empirical Validation

The prototype implementation of our proposal, described in Section 6, allowed us to design and execute a case study in order to validate our (theoretical) findings in a controlled environment of evolving services. The validation focused on the structural layer of services, working exclusively with WSDL descriptions, as typically used in practical settings.

More specifically, we modeled in the SRM prototype the structural description of all versions of the POSERVICE discussed in Section 3 and checked them for compatibility. The results of this procedure agree with the theoretical results as presented in Section 5: The Service Improvement Scenario was found not to be T-shaped, while the Redesign Scenario was concluded to be T-shaped as far as the structural layer is concerned—since the SRM prototype is currently limited to checking for structural compatibility. Ongoing work is concentrating on providing the appropriate extensions to cover the rest of the layers of our approach.

For the empirical validation, we deployed the various versions of the POSERVICE in the Apache Axis2 Web services engine,[17] hosted in an Apache Tomcat servlet container.[18] We used the WSDL2Java code generation tool[19]

16. http://www.eclipse.org/gmt/epsilon/.

17. http://ws.apache.org/axis2/.
18. http://tomcat.apache.org/.
19. http://ws.apache.org/axis2/tools/1_4_1/CodegenToolReference.html.

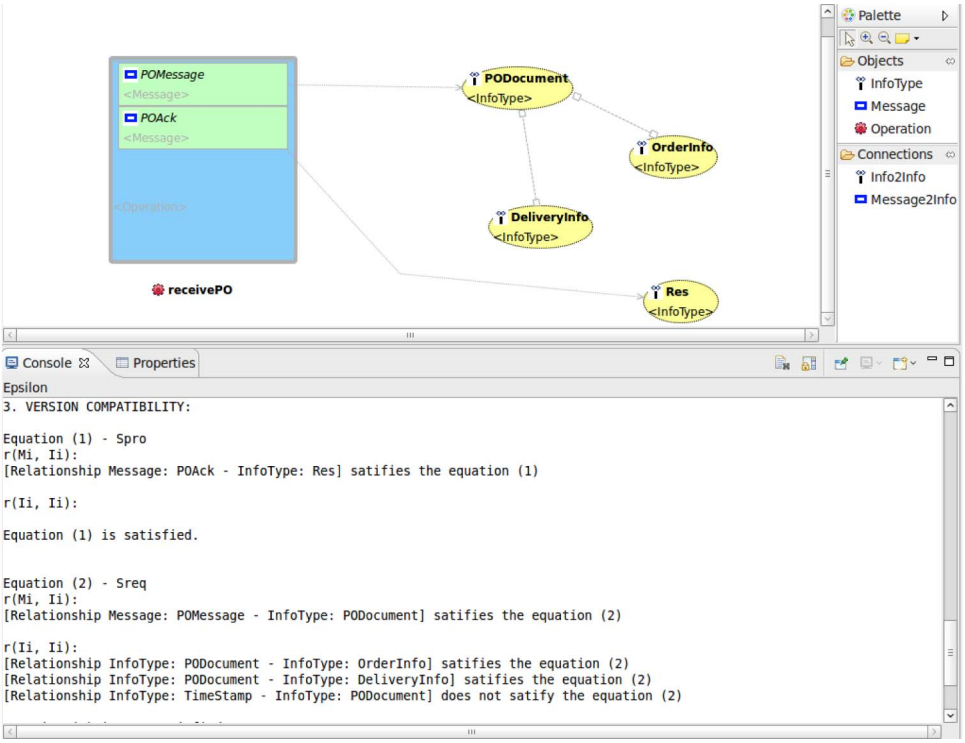


Fig. 9. SRM prototype.

from the Axis2 toolkit to generate a skeleton of the service from the initial version of the service, to which we added the necessary business logic for implementing the functionality of the service. Developing a Service-Based Application to act as the client of the service was also achieved by using the same code generation tool on the initial version of POSERVICE.

We then proceeded to use the generated SBA to invoke the different versions of the deployed service. During runtime, we monitored the server and mined the client logs in order to check whether each service version is invoked successfully, or whether the service version (or the client) breaks. It was observed that the service and the SBA broke in both scenarios, signifying the incompatibility between the different versions of the service. In the case of the Improvement Scenario, this was expected and confirms our theoretical findings. In the case of the Redesign scenario, however, this result diverges from the theoretical prediction due to the inadequacy of the service implementation technology to handle evolution in connection with XML Schema validation.

This conclusion was derived on the basis of an exhaustive investigation of this divergence between theory and practice. In particular, we traced back the produced error messages in order to locate the service container module that produced them. Our findings confirm that the problem stems from the processing of XML messages in both service provider and client sides (see also [43]). This divergence manifests when either side tries to validate the incoming and outgoing messages against an XML Schema that is no longer valid—but not necessarily incompatible. This means that current XML-based implementation technologies are unable to ignore dynamically information not contained in the schema vocabulary. If both parties could ignore the data types not in their message schema and validate the rest of the message, then the empirical result would be in agreement with the theoretical findings of this work.

A workaround for enabling these types of changes is to intercept the messages and apply to them an appropriate transformation using a technology like XSLT[20] or Schematron,[21] as discussed in [43]. Nevertheless, it is our belief that this issue is better handled on the level of service description languages rather than building ad hoc workarounds. In the following section, we explain how this issue can be handled in accordance with the proposed theoretical framework.

7.2 Realization

To evaluate the ability of service description language specifications to handle the mechanisms that support service evolution as discussed in the previous sections, we surveyed their latest versions. In particular, we referred to the WSDL 2.0, BPEL 2.0, and WS-Policy 1.5 specifications. With the exception of WSDL, all surveyed specifications do not contain the notion of versioning. The WSDL 2.0 Primer[22] briefly discusses evolutionary strategies for evolving services, but in a nonnormative manner and by incorporating the strategies found in [16].

As such, the WSDL 2.0 Primer concentrates on the guidelines for backward and forward compatibility as presented in Table 1. WSDL 2.0 is therefore more restrictive than our approach with respect to service evolution. The authors of the Primer, however, acknowledge that changes in the message content depend on the type system used to

20. eXtensible Stylesheet Language Transformations (XSLT) Version 2.0 http://www.w3.org/TR/xslt20/.
21. http://www.schematron.com/.
22. WSDL Version 2.0 Part 0: Primer http://www.w3.org/TR/wsdl20-primer.

describe them. The weak typing approach taken in the processing of messages and the static binding of service and client implementations to WSDL documents leave little space for improvement given the limitations of existing technologies and standards for Web services.

In order to facilitate the realization of our compatible service evolution framework, we propose that the current WSDL specification works in tandem with BPEL and WS-Policy in order to integrate all aspects of service description into a tightly connected set of documents. While both BPEL and WS-Policy can currently refer to WSDL elements in their documents, the integration of the three languages is quite loose on purpose. As we have already discussed, and despite its market dominance, WSDL is limited in the amount of information that it can carry with respect to the needs of consumers. Providing a tighter integration of the three specifications into a vertically integrated document that combines all three specifications would make the serialization of the ASD model easier.

More importantly, a stronger typing system than the current one has to be used both on the level of XML processing and on the level of the respective standard specifications. The model of simple XML parsing backed by XML Schema validation, currently used in most Web services technologies, stifles evolution and creates unnecessary coupling in both service provider and consumer sides. Despite the option of XML extensibility, it is difficult to design for compatible service evolution without the possibility of ignoring the parts of a message that are not understood by the message consumer.

We therefore propose that the parsing and validation model should be replaced by an automatic *marshaling* of messages (that is, their transformation into the respective objects) and the *check for compatibility* on the level of records (using the CCF presented in Section 4). Static bindings should be replaced by dynamic bindings to interface classes. These classes are able to accommodate the subtyping of the messages and representations through the use of inheritance (in static languages like Java) or a combination of inheritance and dynamic binding of types (in dynamic languages like Ruby[23]).

Finally, the use of XML namespaces for version identification should be replaced by (or be combined with) a more robust versioning mechanism. For instance, version attributes should be natively included in the service description document. While very practical and easy to implement, namespace-based techniques depend exclusively on the service developer to be realized, as shown by our case study. This dependence increases the propensity for errors and miscommunication. Furthermore, using a different namespace identifier for each modification unnecessarily breaks the service clients and increases the maintenance costs by introducing additional versions. As such, they should be used with caution.

8 Related Work

Evolution is particularly important in distributed systems due to a complex web of software component interdependencies. As Bennet and Rajlich point out [3], attempting to apply conventional maintenance procedures (halt operation, edit source, and reexecute) in large distributed systems (like the ones emerging in SOA) is not sensible. On one hand, the difficulty of identifying which services form the system itself is nontrivial, especially in the context of large service networks. On the other hand, the matter of ownership and access to the actual source code of third-party services, directly linked to the encapsulation and loose coupledness principles promoted by service orientation, does not allow the application of many of the maintenance techniques like refactoring or impact analysis.

Historically and conceptually, Component-Based Systems (CBS) are the predecessor of SOA and share the principles of encapsulation, independence, and unambiguous definition of interfaces. However, components and services are quite different in terms of coupling, binding, granularity, delivery, and communication mechanisms and overall architecture [44]. As a result, the applicability of a component evolution theory or technique as summarized by Stuckenholz [45] should always be examined carefully before being adopted.

For purposes of presentation, we distinguish between two categories of approaches for compatible service evolution:

1. *Corrective*—adaptation-based approaches that actively enforce the nonbreaking of existing consumers by modifying the service, and
2. *Preventive*—that attempt to delimit and forbid changes that would break the consumers (instead of fixing them). The compatible evolution framework as discussed in the previous sections falls in this category.

Corrective approaches involve different mechanisms for adapting either the interface or the implementation of the service (or both) to the interoperability requirements of the service consumers. Adaptation has been introduced in CBS, where adapting a component-based system means modifying one or more of its components. In practice, most components cannot be integrated directly into a system-to-be because they are incompatible. Software Adaptation aims at generating, as automatically as possible, adapters to compensate for these incompatibilities. Numerous adaptation approaches have been proposed, see, for example, [46], [47]. These approaches, however, address low-level component mismatches and they cannot ensure the invariants of the overall system. This is critical for service-oriented systems which are composed—rather than implemented—and for which the adaptation to one mismatch may outweigh its utility.

Service adaptation can be further distinguished into two categories: *interface adaptation*, where the goal is to solve mismatches in the signature and/or protocol of collaborating services by modifying the interfaces accordingly, e.g., [31], [48], and *composition adaptation*, where the subject of change is the aggregation of services constituting the composite service [49], [50], and [51]. In this case, either the services participating in the composition are replaced by other equivalent services, or the "gluing" connecting them is modified, or both.

Adapters are an alternative approach for preserving compatibility without modifying the service itself. The basic

23. http://www.ruby-lang.org/en/.

idea is to resolve the mismatches between the expected by the consumers and the supported by the implementation interfaces. Brogi and Popescu [32], Nezhad et al. [33], for example, support the (semi)automated generation of adapters between service interfaces and implementations, based on the parametric transformation of the expected and the actually offered interfaces of the service. Interface adapters can also be layered on top of each other (e.g., in cross stubs and custom handlers in [52] or chain of adapters in [53]) to "mask" the mismatches and maintain compatibility between providers and consumers. By using this technique, service developers deal with an (ideally) unique implementation endpoint that exposes multiple versions of interfaces, instead of multiple versions of the service. The maintenance cost then is moved to the consistency and efficiency of the layering of the adapters and out of the service life cycle itself.

There are a number of issues with these corrective approaches with respect to service evolution. First, adaptation does not necessarily happen in response to change; it may actually be the cause of change. For example, adaptation may be used for enabling the reuse of services (e.g., [31]). In this respect, adaptation is one of the means by which evolution manifests, the other being the replacement of the services used for the composition and the redeployment of a service in case of service compositions. Furthermore, the application of service adaptation techniques—both for interface and composition—is not always possible without explicit manual intervention. In this sense, these approaches are limited in their automation. The required modifications may also interfere with the operation of other services by the same organization in terms of resources (computational and financial) and code. Service adapters avoid this risk by not requiring the redevelopment of the service. They, however, transfer the service adaptation cost to the effort required for developing, and more importantly, maintaining the adapters. Finally, the majority of the corrective approaches discussed above focus on the generation of the adaptation with the goal to automate the process without first checking whether the adaptation is necessary (in terms of compatibility). However, this is not always true as, for example, discussed in [52] and [53]. These approaches incorporate compatibility checks before attempting to generate suitable adapters.

Most of the approaches discussed in Section 2 can be classified as preventive approaches. As we explained in length, all these approaches take the very pragmatic road of providing a set of guidelines for the compatible evolution of services based on existing technologies. Of particular mention is the work of Becker et al. [17] that also presents an approach that constrains the evolution of services based on backward compatibility. They also, however, depend on a guideline-based approach which is limited in expressiveness and portability in other technologies. In this work, we are abstracting from the particular technology used for the implementation of services and present a theoretical framework that not only covers, extends, and explains the outcome of these approaches, but also provides a formal foundation on which the effect of changes to the interface of service can be reasoned on. In [21], we discuss the fundamentals of the framework for compatible evolution, but we focus exclusively on the structural aspect of services. In this work, we extend this framework to the behavioral and QoS-related aspects of service description and we update it to cover compatibility for those aspects accordingly.

Finally, as discussed in Section 4, our approach and all the approaches discussed in Section 2 assume that the change sets between service versions are available as part of the development process of the service. In case they are not, then a number of works on version differencing in UML models [54], MOF-based models [55], or business process models [39] can be used. However, these works focus on model consistency, and as such they do not provide a decidability theory for guaranteeing type-safety and correct versioning transitions so that previous clients can use a versioned service in a consistent manner. This is the very essence of the approach followed by this paper.

9 Conclusions and Future Work

In this paper, we presented a theoretical framework and language-independent mechanisms to assist service developers in controlling and managing service changes in a uniform and consistent manner. For this purpose, we distinguished between shallow (small-scale, localized) changes and deep (large-scale, cascading) changes and we focused on shallow changes. In particular, we provided a sound theory for service compatibility and reasoning mechanisms for delimiting the effect of changes which we keep local to and consistent with a service description.

The approach proposed in this paper is in contrast to the vast majority of existing approaches in the field. Most such approaches view service compatibility as strictly enforced by a set of empirical and technology-specific rules (e.g., WSDL-dependent guidelines) which indicate which changes are characterized as being compatible. This results in a very strict service evolution regime which prohibits potentially legitimate changes from being applied due to the limitations of current service technologies.

We presented a formally backed *compatible service evolution* framework which is based on a technology-agnostic notation for the representation of services in the form of *Abstract Service Descriptions*. ASDs act as a springboard to explain the versioning mechanisms for services. Using these results, we formally defined service compatibility and developed a theory for the compatible evolution of services. As part of this approach, we introduced the notion of *T-shaped changes*, which enforce service compatibility between interrelated service versions in two dimensions: horizontally and vertically. We also demonstrated how to reason about the compatibility of service versions using a *Compatibility Checking Function* in order to decide if changes to them are T-shaped or not. We validated our compatible service evolution framework in practice by means of a proof-of-concept prototype implementation in the form of the *Service Representation Modeler* tool and a case study. Based on the findings of this validation, we provided a series of recommendations for the improvement of service description languages in the context of service evolution.

The compatible service evolution framework will be extended in the immediate future to show that the set of

shallow changes is *closed* under compatibility-preserving change sets, and that our approach is *complete* in the mathematical sense by illustrating that the CCF can generate all possible T-shaped sets, as discussed in [35]. Following this, the SRM prototype will be extended with appropriate behavioral and nonfunctional layer capabilities to ensure consistency in service versioning. While reasoning on the structural layer was sufficient for the purposes of this paper, extending this capability to the other layers is considered critical for the full implementation of the framework. Furthermore, the option to import ASD models directly from WSDL, BPEL, and WS-Policy (currently performed semi-automatically) and to visualize the results of the compatibility check will also be added to the SRM capabilities.

These changes will allow us to provide service developers with a comprehensive toolset for controlling the different aspects of service evolution. The application of the SRM prototype in-the-field would allow us to draw useful conclusions about the efficacy of both the prototype and our work in general. Of course, this process would also allow us to further improve and extend our research findings.

Finally, throughout all the work presented here, we have assumed a direct bilateral consumer/provider type of interaction between services and their clients. This introduces a certain amount of rigidity in service evolution. To relax this rigidity and allow for additional change scenarios that guarantee type-safe evolution, we experimented with service contracting and Service Level Agreements (SLAs) [25]. In particular, we investigated the application of an intermediary construct in the form of a *service contract* interposed between service providers and consumers. A service contract can be used to represent an SLA between a service provider and consumer. Explicit contracts between providers and consumers allow for greater flexibility in evolving both parties in a compatible manner as they relax some of the assumptions regarding the ability of services to evolve while preserving their compatibility. We furthermore showed how even the contract itself can be a subject to change without affecting the interacting parties that it binds. Due to the fact that service contract formation depends on the subtyping relation as defined here, the work on contracting can be easily incorporated with the compatible evolution framework described in this paper.

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). The authors wish to thank the reviewers for the many insightful comments and constructive criticism that have resulted in substantially improving the quality of this paper. They would also like to thank the members of ERISS, and especially Michael Parkin, Michele Mancioppi, and Oktay Türetken, as well as Barbara Pernici, Hossein Siadat, and Mariagrazia Fugini at Politecnico di Milano for their feedback and support while writing and improving this paper. They also thank Juan Vara and David Granada in the Kybele Research Group at the University Rey Juan Carlos for their invaluable help with the prototype implementation.

References

[1] Q. Yu, X. Liu, A. Bouguettaya, and B. Medjahed, "Deploying and Managing Web Services: Issues, Solutions, and Directions," *VLDB J.*, vol. 17, no. 3, pp. 537-572, May 2008.

[2] M.P. Papazoglou, "The Challenges of Service Evolution," *Proc. 20th Int'l Conf. Advanced Information Systems Eng.*, pp. 1-15, 2008.

[3] K.H. Bennett and V.T. Rajlich, "Software Maintenance and Evolution: A Roadmap," *Proc. Conf. Future of Software Eng.*, pp. 73-87, 2000.

[4] J. Estublier, D. Leblang, A. van der Hoek, R. Conradi, G. Clemm, W. Tichy, and D. Wiborg-Weber, "Impact of Software Engineering Research on the Practice of Software Configuration Management," *ACM Trans. Software Eng. Methodology*, vol. 14, no. 4, pp. 383-430, 2005.

[5] K. Brown and M. Ellis, "Best Practices for Web Services Versioning," http://www.ibm.com/developerworks/webservices/library/ws-version/, Jan. 2004.

[6] K. Jerijærvi and J. Dubray, "Contract Versioning, Compatibility and Composability," http://www.infoq.com/articles/contract-versioning-comp2, Dec. 2008.

[7] C. Peltz and A. Anagol-Subbarao, "Design Strategies for Web Services Versioning," http://soa.sys-con.com/node/44356, 2004.

[8] R. Conradi and B. Westfechtel, "Version Models for Software Configuration Management," *ACM Computing Surveys*, vol. 30, no. 2, pp. 232-282, 1998.

[9] P. Leitner, A. Michlmayr, F. Rosenberg, and S. Dustdar, "End-to-End Versioning Support for Web Services," *Proc. IEEE Int'l Conf. Services Computing*, vol. 1, pp. 59-66, July 2008.

[10] M.B. Juric, A. Sasa, B. Brumen, and I. Rozman, "WSDL and UDDI Extensions for Version Support in Web Services," *J. Systems and Software*, vol. 82, no. 8, pp. 1326-1343, Aug. 2009.

[11] M. Endrei, M. Gaon, J. Graham, K. Hogg, and N. Mulholland, "Moving Forward with Web Services Backward Compatibility," http://www.ibm.com/developerworks/java/library/ws-soa-backcomp/index.ht ml?ca=drs-, May 2006.

[12] R. Weinreich, T. Ziebermayr, and D. Draheim, "A Versioning Model for Enterprise Services," *Proc. 21st Int'l Conf. Advanced Information Networking and Applications Workshops*, vol. 2, pp. 570-575, 2007.

[13] R. Fang, L. Lam, L. Fong, D. Frank, C. Vignola, Y. Chen, and N. Du, "A Version-Aware Approach for Web Service Directory," *Proc. IEEE Int'l Conf. Web Services*, pp. 406-413, July 2007.

[14] D. Parachuri and S. Mallick, "Service Versioning in SOA," http://www.infosys.com/offerings/IT-services/soa-services/white-papers/pages/index.aspx, Dec. 2008.

[15] A. Narayan and I. Singh, "Designing and Versioning Compatible Web Services," http://www.ibm.com/developerworks/websphere/library/techarticles/0705_narayan/0705_narayan.html, Mar. 2007.

[16] D. Orchard ed. "Extending and Versioning Languages: XML Languages [ed. Draft],"World Wide Web Consortium (W3C), http://www.w3.org/2001/tag/doc/versioning-xml, July 2007.

[17] K. Becker, A. Lopes, D.S. Milojicic, J. Pruyne, and S. Singhal, "Automatically Determining Compatibility of Evolving Services," *Proc. IEEE Int'l Conf. Web Services*, pp. 161-168, 2008.

[18] R. Kazhamiakin ed., "CD-IA-3.2.1 Initial Definition of Validation Scenarios," S-Cube Consortium, http://www.s-cube-network.eu/, Oct. 2009.

[19] A. Gehlert and A. Metzger eds., "CD-JRA-1.3.2 Quality Reference Model for SBA," S-Cube Consortium, http://www.s-cube-network.eu/, Mar. 2008.

[20] M. Belguidoum and F. Dagnat, "Formalization of Component Substitutability," *Electronic Notes in Theoretical Computer Science*, vol. 215, pp. 75-92, 2008.

[21] V. Andrikopoulos, S. Benbernou, and M.P. Papazoglou, "Managing the Evolution of Service Specifications," *Proc. 20th Int'l Conf. Advanced Information Systems Eng.*, pp. 359-374, 2008.

[22] G. Castagna, N. Gesbert, and L. Padovani, "A Theory of Contracts for Web Services," *ACM Trans. Programming Languages and Systems*, vol. 31, no. 5, pp. 1-61, 2009.

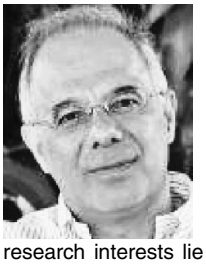
[23] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli, "Model Evolution by Run-Time Parameter Adaptation," *Proc. IEEE 31st Int'l Conf. Software Eng.*, pp. 111-121, 2009.

[24] M. Comuzzi and B. Pernici, "A Framework for QoS-Based Web Service Contracting," *ACM Trans. Web*, vol. 3, no. 3, pp. 1-52, 2009.

[25] V. Andrikopoulos, S. Benbernou, and M.P. Papazoglou, "Evolving Services from a Contractual Perspective," *Proc. 21st Int'l Conf. Advanced Information Systems Eng.*, pp. 290-304, 2009.
[26] M.P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-Oriented Computing: State of the Art and Research Challenges," *Computer*, vol. 40, no. 11, pp. 38-45, Nov. 2007.
[27] B. Benatallah, F. Casati, and F. Toumani, "Representing, Analysing and Managing Web Service Protocols," *Data & Knowledge Eng.*, vol. 58, no. 3, pp. 327-357, 2006.
[28] S.H. Ryu, F. Casati, H. Skogsrud, B. Benatallah, and R. Saint-Paul, "Supporting the Dynamic Evolution of Web Service Protocols in Service-Oriented Architectures," *ACM Trans. Web*, vol. 2, no. 2, pp. 1-46, 2008.
[29] M. Mancioppi, M. Carro, W. Heuvel, and M.P. Papazoglou, "Sound Multi-Party Business Protocols for Service Networks," *Proc. Sixth Int'l Conf. Service-Oriented Computing*, pp. 302-316, 2008.
[30] J.E. Johnson, D.E. Langworthy, L. Lamport, and F.H. Vogt, "Formal Specification of a Web Services Protocol," *Electronic Notes in Theoretical Computer Science*, vol. 105, pp. 147-158, Dec. 2004.
[31] M. Dumas, M. Spork, and K. Wang, "Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation," *Proc. Fourth Int'l Conf. Business Process Management*, pp. 65-80, 2006.
[32] A. Brogi and R. Popescu, "Automated Generation of BPEL Adapters," *Proc. Int'l Conf. Service Oriented Computing*, pp. 27-39, 2006.
[33] H.R.M. Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati, "Semi-Automated Adaptation of Service Interactions," *Proc. 16th Int'l Conf. World Wide Web*, pp. 993-1002, 2007.
[34] J. Ponge, B. Benatallah, F. Casati, and F. Toumani, "Analysis and Applications of Timed Service Protocols," *ACM Trans. Software Eng. and Methodology*, vol. 19, no. 4, pp. 1-38, 2010.
[35] V. Andrikopoulos, *A Theory and Model for the Evolution of Software Services*, CentER Dissertation Series. Tilburg Univ. Press, 2010.
[36] B.H. Liskov and J.M. Wing, "A Behavioral Notion of Subtyping," *ACM Trans. Programming Languages and Systems*, vol. 16, no. 6, pp. 1811-1841, 1994.
[37] B. Meyer, *Object-Oriented Software Construction*, second ed. Prentice Hall PTR, 1997.
[38] J.F. Allen, "Maintaining Knowledge about Temporal Intervals," *Comm. ACM*, vol. 26, no. 11, pp. 832-843, 1983.
[39] J.M. Küster, C. Gerth, and G. Engels, "Dynamic Computation of Change Operations in Version Management of Business Process Models," *Proc. Sixth European Conf. Modelling Foundations and Applications*, pp. 201-216, June 2010.
[40] J. Vara, D. Granada, V. Andrikopoulos, and E. Marcos, "Modeling and Comparing Service Descriptions," Technical Report TR-29032010, Univ. Rey Juan Carlos, Dept. of Computing Languages and Systems II, http://kybele.es/research/TR/TR-29032010.pdf, June 2010.
[41] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T.J. Grose, *Eclipse Modeling Framework*. Addison-Wesley Professional, Aug. 2003.
[42] A. Hevner, S. March, J. Park, and S. Ram, "Design Science in Information Systems Research," *MIS Quarterly*, vol. 28, no. 1, pp. 75-105, 2004.
[43] I. Robinson, "Consumer-Driven Contracts: A Service Evolution Pattern," *The ThoughtWorks Anthology: Essays on Software Technology and Innovation*, pp. 101-120, Pragmatic Bookshelf, http://martinfowler.com/articles/consumerDrivenContracts.html, Mar. 2008.
[44] M.P. Papazoglou, *Web Services: Principles and Technology*. Prentice Hall, July 2007.
[45] A. Stuckenholz, "Component Evolution and Versioning State of the Art," *ACM SIGSOFT Software Eng. Notes*, vol. 30, no. 1, p. 7, 2005.
[46] S. Becker, A. Brogi, S. Overhage, E. Romanovsky, and M. Tivoli, "Towards an Engineering Approach to Component Adaptation," *Architecting Systems with Trustworthy Components*, vol. 3938, pp. 193-215, Springer-Verlag, 2006.
[47] C. Canal, P. Poizat, and G. Salaün, "Model-Based Adaptation of Behavioral Mismatching Components," *IEEE Trans. Software Eng.*, vol. 34, no. 4, pp. 546-563, July/Aug. 2008.
[48] D. Ardagna, M. Comuzzi, E. Mussi, B. Pernici, and P. Plebani, "PAWS: A Framework for Executing Adaptive Web-Service Processes," *IEEE Software*, vol. 24, no. 6, pp. 39-46, Nov./Dec. 2007.
[49] W. Kongdenfha, R. Saint-paul, B. Benatallah, and F. Casati, "An Aspect-Oriented Framework for Service Adaptation," *Proc. Int'l Conf. Service Oriented Computing*, pp. 15-26, 2006.
[50] M. Colombo, E.D. Nitto, and M. Mauri, "SCENE: A Service Composition Execution Environment Supporting Dynamic Changes Disciplined through Rules," *Proc. Int'l Conf. Service Oriented Computing*, pp. 191-202, 2006.
[51] D. Ardagna and B. Pernici, "Adaptive Service Composition in Flexible Processes," *IEEE Trans. Software Eng.*, vol. 33, no. 6, pp. 369-384, June 2007.
[52] S.R. Ponnekanti and A. Fox, "Interoperability among Independently Evolving Web Services," *Proc. Fifth ACM/IFIP/USENIX Int'l Conf. Middleware*, pp. 331-351, 2004.
[53] P. Kaminski, M. Litoiu, and H. Müller, "A Design Technique for Evolving Web Services," *Proc. Conf. Center for Advanced Studies on Collaborative Research*, pp. 303-317, 2006.
[54] Z. Xing and E. Stroulia, "Differencing Logical UML Models," *Automated Software Eng.*, vol. 14, no. 2, pp. 215-259, 2007.
[55] M. Alanen and I. Porres, "Difference and Union of Models," *UML 2003—The Unified Modeling Language*, vol. 2863/2003, pp. 2-17, Springer, 2003.

Vasilios Andrikopoulos completed the pre- and postgraduate studies in the Computer Engineering and Informatics Department of the University of Patras, Greece, and received the PhD degree cum laude from the Department of Information Management at Tilburg University, The Netherlands. He works as a postdoctoral researcher for the Institute of Architecture of Application Systems (IAAS), University of Stuttgart, Germany. His research interests include service-oriented architecture and computing, with an emphasis on the evolution of services and cloud application engineering.

Salima Benbernou is a full professor of computer science at Paris Descartes University. She participated in more than seven French and EU-funded research projects, being coordinator in some of them. Her research interests include formal models for service-oriented computing, privacy in information systems and databases. She has authored more than 50 papers published in peer-reviewed conferences, journals, and book chapters, and served as a PC member and coorganizer in several conferences and workshops.



Michael P. Papazoglou holds the chair of computer science at Tilburg University. He is the scientific director of the European Research Institute in Service Science (ERISS), Tilburg University, and European Commission's Network of Excellence, S-Cube. He is also an honorary professor at the University of Trento in Italy, and professorial fellow at the Universities of Lyon, France, New South Wales, Australia, and Rey Juan Carlos, Madrid, Spain. His research interests lie in the areas of service-oriented computing, web services, large scale data sharing, business processes, and federated and distributed information systems. He has published numerous books, monographs, and international conference proceedings, journal, and conference papers. He is the editor-in-charge of the MIT Press book series on Information Systems as well as the founder and editor-in-charge of the new Springer-Verlag book series on Service Science. He is one of the most cited researchers in the area of service-oriented computing. He is a senior member of the IEEE and a Golden Core member and a distinguished visitor of the IEEE Computer Society.