WSDL and UDDI extensions for version support in web services

Matjaz B. Juric\textsuperscript{a,}\textsuperscript{*}, Ana Sasa\textsuperscript{b}, Bostjan Brumen\textsuperscript{a}, Ivan Rozman\textsuperscript{a}

\textsuperscript{a}University of Maribor, FERI, Institute of Informatics, Smetanova 17, SI-2000 Maribor, Slovenia
\textsuperscript{b}University of Ljubljana, FRI, Information Systems Laboratory, Trzaska 25, SI-1000 Ljubljana, Slovenia

\textbf{1. Introduction}

In this article, we address versioning of web services. Versioning is an important aspect of web service development, particularly in complex mission-critical systems and in service-oriented architectures (SOA). Versioning is important because web services evolve over time (Vinoski, 2004). Therefore, more than one version of a single service interface has to be maintained. In SOA, versioning is also related to the development of reusable services, which is one of the key objectives of SOA. Reusable services are developed in several iterations. In each iteration, the service is enhanced to be better suited for reuse. Enhancements most likely require changes to the interface and/or to the behavior of the service. In this article, we focus on service interfaces. In loosely-coupled environments we cannot instantly upgrade all service consumers to use the latest version of a service interface. We have to maintain the older versions of the service interface as well.

To describe the interfaces of web services, WSDL (Web Services Description Language) is used. WSDL does not support versioning. Therefore evolving a service interface through its life cycle becomes challenging. Usually it requires managing a separate instance of a web service for each version. This quickly increases the number of service instances, making it difficult to manage and govern. To manage a central registry of web services, a UDDI (Universal Description, Discovery, and Integration) compliant registry is used. UDDI also does not support versioning and does not provide a standard way to denote service interface versions, to query for versions and to notify the service consumers about version changes. In SOA service composition is done in a dynamic way, therefore version information has to be available at run-time and at development-time.

In this article, we present extensions to WSDL and UDDI to support run-time and development-time versioning of service interfaces. We have designed the extensions for WSDL 2.0 and 1.1, and for UDDI v3. Our proposed solution uses the WSDL and UDDI extension mechanisms, which makes it relatively easy to incorporate the extensions into existing development environments. We also propose annotation extensions for developing versioned web services in Java. The annotations are designed in a portable manner and can be used with other languages than Java, such as C#. As a proof-of-concept, we have tested the presented solution for versioning in two large real-world environments and observed several important improvements in service development and maintenance efficiency, improved service reuse, and simplified governance.

\textsuperscript{*} Corresponding author.
\texttt{E-mail address: matjaz.juric@uni-mb.si} (M.B. Juric).

© 2009 Elsevier Inc. All rights reserved.

\textbf{A R T I C L E I N F O}

Article history:
Received 20 July 2008
Received in revised form 3 March 2009
Accepted 4 March 2009
Available online 12 March 2009

\textbf{A B S T R A C T}

Versioning is an important aspect of web service development, which has not been adequately addressed so far. In this article, we propose extensions to WSDL and UDDI to support versioning of web service interfaces at development-time and run-time. We address service-level and operation-level versioning, service endpoint mapping, and version sequencing. We also propose annotation extensions for developing versioned web services in Java. We have tested the proposed solution for versioning in two real-world environments and identified considerable improvements in service development and maintenance efficiency, improved service reuse, and simplified governance.

\textsuperscript{0}164-1212/$ - see front matter © 2009 Elsevier Inc. All rights reserved.

doi:10.1016/j.jss.2009.03.001

\textbf{Keywords:}
Versioning
Web services
SOA
WSDL
UDDI
results of testing our proposed solution in two real-world environments. In Section 6, we discuss the related work. We provide conclusions in Section 7.

2. Brief overview of WSDL and UDDI

WSDL is a W3C (World Wide Web Consortium) recommendation and has become the de-facto standard for defining web service interfaces (W3C, 2001; W3C, 2007). In SOA, WSDL is used to describe the interfaces of all services irrespective of the underlying technology. In June 2007, W3C has published the WSDL 2.0 recommendation. However, at the time of writing not many vendors have adopted version 2.0. It is not clear whether version 2.0 will gain the necessary support (Salz, 2004; Little, 2007). Therefore, we have designed the versioning extensions for both WSDL versions 1.1 and 2.0.

WSDL 1.1 (W3C, 2001) describes the interface of a service within the <portType> section, where the operations are listed. WSDL supports one-way and two-way (request/response) operations. Each operation is defined by input, output and optional fault messages. The input message represents the payload an operation receives in order to perform the processing, while the output message represents the resulting payload. The fault message is used to signal faults. Output and fault messages can only be used in request/response operations. Each message consists of several parts. Message parts are defined within the <message> section. Each message part is defined by the corresponding schema as a simple XML type, a complex type or an element. The schema can be found in the WSDL document under the <types> section. A service can be embedded in the WSDL document or can be imported from a standalone schema (XSD) document. The <binding> defines the mapping of the interface/payload to the corresponding protocol, such as SOAP, or any other supported protocol. Within the binding part we also define the style (document or RPC) and the payload representation (literal or encoded). The <service> defines the service endpoint location.

WSDL 2.0 (W3C, 2007) has simplified the interface description. The interface is defined within the <interface> section. WSDL 2.0 has dropped the concept of input and output messages. Rather the operations directly specify XML elements for input and output. For faults, fault messages can be specified within the <interface> section. This allows that the same fault message is used in different operations, which simplifies fault handling. WSDL 2.0 has introduced the message exchange patterns, which specify the sequence in which the associated messages are to be transmitted between the service and the client. WSDL 2.0 also allows interface inheritance. The <binding> and <service> parts have not changed significantly apart from the new syntax. WSDL 2.0 has retained support for literal style of web services only. Neither WSDL 2.0 nor 1.1 supports versioning.

UDDI defines a standardized model for service registries (OASIS, 2004). It defines the information model, the service providers API for registering and publishing services, and the API for service consumers to inquire for services. UDDI is an OASIS specification. Current version is v3. Although UDDI has not been successful with public registries, it has been widely used for private registries in enterprises. The vast majority of commercial and open-source service registries are compliant with the UDDI specification. UDDI does not support versioning.

3. Versioning in web services development

Versioning is very important in software architectures (Hofmeister et al., 2000) and particularly in web services development (Gaur and Zirn, 2007). There are two aspects of web service versioning. First is interface (WSDL) versioning. Interfaces are the contracts between the service providers and service consumers. Second is the versioning of web service implementation. In this article, we focus on web service interface versioning; that is on WSDL versioning. Service implementation versioning does not differ from general source code versioning.

The two most important reasons for versioning service interfaces are evolution of services over time, and reusability (Juric et al., 2007). The need to evolve services through time requires extending and modifying their interfaces according to the new requirements, which are, to a large extent, unpredictable (Lassing et al., 2003). The second important reason is reusability. Reusable services are usually developed in several iterations (van Vliet, 2008). In the first iteration, the service fulfills the specific requirements of the application being developed. When such service is about to be reused in a different context, service usually has to be adapted, extended, or modified in a certain way.

In web service and SOA environments we usually do not control all the service consumers (clients) a service has. Service consumers can come from other organizations. Sometimes we might even not know all the service consumers that are using a service. In such cases, it is virtually impossible to upgrade all the service consumers at the same time as we deploy the new version of a service with a changed or modified service interface. The same holds true for scenarios where web services are parts of orchestrations, such as BPEL processes (Juric et al., 2006). One orchestration might require a different version of a service than the other. Therefore, we have to retain different versions of a service at run-time. Because web services can be orchestrated at run-time, we have to retain the interface version information not only at development-time but also at run-time.

We can classify the changes in a new version of a service interface into backward-compatible and backward-incompatible changes. Backward-compatible changes are such changes that do not break the contract with the service consumer. This means that the client can use the new service interface version without modifications. Backward-compatible changes include:

- Adding new operations to existing WSDL. For example, a WSDL, which specifies a single operation checkBookPrice() is extended with an operation checkBookRating().
- Extending types in XML schemas with addition of new elements. For example, checkBookPrice() operation returns a message of XML element checkBookPriceResponse, which originally consisted of two subelements price and currency. Now a third subelement shippingCosts is added.

In both cases, existing service consumers can continue to use the new service interface without breaking the compatibility. In practice however, even such modifications might break the compatibility with clients. The most likely reason is that the clients use static bindings to WSDL interfaces and related XML schemas. In such cases, additions to WSDL and schemas cannot be incorporated automatically, because WSDL and schemas have to be compiled into implementation classes, and the client has to be recompiled and redeployed. Another reason can be that the clients cannot accommodate changes to schemas. This can happen for example if XML is processed statically using XML serialization. Similarly as before, the client has to be adapted to a new version of the service interface, recompiled and redeployed.

Even more often, we have to deal with backward-incompatible interface changes, such as:

- Renaming or removing an operation.
- Changing the behavior of an operation.
Several workarounds have been proposed, such as WSDL interfaces. UDDI v3 also does not support WSDL versions.

Support for versioning in WSDL and UDDI is out of the scope of this article and is usually addressed as a part of the selected software development methodology or more specifically change management methodology. In service-oriented development methodologies, such as SOMA (service-oriented modeling and architecture, Arsanjani, 2004), specific aspects of service interface versioning and service implementation versioning should be addressed. Versioning of service interfaces and implementations are related, however not every service implementation version is reflected in service interface version. Service consumers might be interested in service interface changes related to functional changes, such as backward compatible and incompatible changes described earlier in this section. They might also be interested in non-functional changes, such as quality of service aspects (response time for example). Service consumers might not be interested in minor changes, such as bug fixes, if they do not influence functional and non-functional requirements. Service providers can accommodate such changes independently of the service consumers. This means that the version identifier schema for versioning service interfaces can be simpler that for versioning service implementations; and version strategy should consider this. Service interface and service implementation versioning should be related through version model, as described in Conradi and Westfechtel (1998).

Service interface versioning strategy should also address the relation between backward-compatible and backward-incompatible service interface versions, and the corresponding version identifiers. For example, backward-incompatible changes can be associated with major versions and backward-compatible changes to minor versions. Or, backward-incompatible changes can be associated with major and minor versions, backward compatible with maintenance versions (third and succeeding numbers in versioning schema).

4. Support for versioning in WSDL and UDDI

Neither WSDL 1.1 nor WSDL 2.0 provides support for versioning WSDL interfaces. UDDI v3 also does not support WSDL versions (Vinoski, 2004). Several workarounds have been proposed, such as (Kenyon, 2003; Irani, 2004; Brown and Ellis, 2004; Pelz and Subbarao, 2004; Posley, 2006). The usual approach to service versioning has been to publish different versions of the same service under different names. All these names have been registered in the UDDI. This approach has been usually combined with the use of naming conventions for XML namespaces. For example, we might have used namespace: http://uni-mb.si/bookService/1.0, where “1” stands for major version and “0” stands for minor version.

This approach has sometimes been extended with the introduction of a façade (or mediation) service, which has decoupled the actual service implementation from the client. Façade service has been used to mask the differences between the implementation of the service and different versions of service interface.

The described approaches have numerous disadvantages. Web service consists of several development artifacts, such as the WSDL interface, the interface mapped to the implementation language (such as Java or C#) interface, the types mapped to the implementation language classes, the implementation class (or classes), etc. All these artifacts usually had to be copied and modified in order to develop a new version of an existing service. There has been no traceability between the versions. If a modification had to apply to more than one version, developers had to change multiple instances of source code. This problem can be diminished if a version control system is in place. However, web service development tools generate code from WSDL and XML schemas (interface and type mappings). They also generate proxies (stubs), skeletons, deployment descriptors and other artifacts. In addition web services might be developed by a team of people. A version control system with branch and merge support is not enough to support all the above-mentioned requirements.

Because service versions have been published under different names, a large number of service endpoints existed. Managing and governing such a large number of services has been difficult and has hindered development of sound architectures. Large number of services also makes reuse more difficult. There have been no commonly accepted naming conventions for versioning services. There has also been no standard way of telling service consumers which is the current version of a service and which versions are deprecated.

5. Proposed solution for web service versioning

Our objective has been to design a solution for web service versioning, which would address all the disadvantages, mentioned in the previous section. The main objectives for developing our solution have been to provide the abilities to:

- Provide run-time and development-time versioning support.
- Develop multiple versions of a service from the same codebase.
- Trace changes between the versions of a service interface.
- Manage the dependencies between different versions of a service interface.
- Support simultaneous operation of different versions of a service interface.
- Design a solution that is compatible with service consumers, which are not aware of versioning extensions.
- Design a solution that can be supplemented with registry extensions to make the service registry version aware.
- Design a solution that can be mapped to programming language annotation extensions for versioning.
- Notify the service consumers about interface version identifiers, new versions, and version deprecation.
- Design a solution that is simple to understand and implement. The solution that we present can be implemented using a preprocessor and is therefore usable with all major development environments and tools.

We propose an extensional versioning solution with explicit versions. To make the solution flexible we support different ver-
sion identifier schemas. For example, we can use numeric, alphanumeric, date, or any other version identifier schema that best suits our development methodology. To achieve this we define the version identifier (vid) as a custom simple XML schema type. To specify the required version identifier schema, we use a pattern restriction with a specific regular expression to match our required version identifier schema. This approach is compatible with the xADL 2.0 (Institute for Software Research, 2005).

Listing 1 shows an example of a simple version identifier that consists of major.minor versions numbers. We will use this version identifier in the rest of the article for simplicity purposes. In real-world we might want to use a more sophisticated identifier.

Our approach to versioning supports different intents achieved through interface versioning. According to Conradi and Westfechtel (1998) this can be revisions, which are intended to supersede its predecessor; variants, which are intended to coexist; and version cooperation for multiple service consumers to work on different versions of interfaces.

Our proposed solution consists of three parts: the extensions to WSDL, the extensions to UDDI, and the extensions of Java language annotations. We will present them in the next subsections.

5.1. Versioning extensions for WSDL

To enable versioning of web service interfaces, we propose an extension to WSDL. WSDL interfaces are described with the following elements: types, messages, interface (port-type), bindings, and service location. These elements might or might not have the same version life cycle. For example, types (XML schemas) can have a different versioning strategy than the interfaces. Types might be extended or modified without the need to change or modify the operations. And vice-versa – operations can be changed without modifying types, for example if an operation is converted from request/response to one-way operation. New operations might be added to the interface. These operations might use the same types therefore no change is required in the schema. A service might be amended with a new binding, for example in addition to SOAP-over-HTTP we add SOAP-over-JMS and HTTP-POST bindings. In this case, no change on the interface (port-type) is necessary. Finally, we might want to specify an additional service endpoint location (for example for secure access), which only requires change in the service section.

So far, we have identified examples where only one part of the WSDL needs to be changed for a new version of a service. However, we can also identify examples, where two or more parts have to be changed simultaneously. For example, we might want to modify the operations in the interface (port-type) together with the types and bindings, but we want to use the messages and the service endpoint location without changes. The next possibility is that we do not want to change the interface, but want to change messages and types (for example, we change the message parts and the corresponding types). In this case, the bindings and service parts might stay unchanged. Or they can change if we add additional protocol binding, or if we change the service style (document or RPC) or payload representation (literal or encoded).

We can conclude that all parts of the WSDL description can have individual version cycles. Such approach to WSDL versioning is fine-grained approach and allows us to have detailed control over the versioning cycle. We call this approach operation-level versioning.

Operation-level versioning has advantages from the perspective of control and allows detailed control over the versioning of types, messages, interface, bindings, and service endpoints. However, this approach requires careful planning and is time consuming. In cases where we do not need such detailed control, or in cases where the WSDL changes require changes to the interface, types, messages, and other elements at the same time, we propose a more coarse-grained approach. We call this coarse-grained approach service-level versioning. We will first look at service-level versioning and then on operation-level versioning.

5.2. Service-level versioning

Service-level versioning enables version control over the WSDL as a whole. It enables to control versioning of single service and sets of multiple services (i.e. application that consists of several services) in a well-organized manner. To achieve this, we propose an extension to WSDL. We use the XML namespace extension mechanism, which allows us to extend WSDL while assuring backward compatibility with tools, which are unaware of the extensions. This is particularly important, because these extensions do not break the compatibility of WSDL with existing tools. To achieve this we have introduced a namespace URI http://uni-mb.ti/wsdلى/versioning, for which we use the alias wsdlx.

Within this namespace we propose an extension element `<wsdlx:versions>`, which declares and describes versions and specifies their dependencies. Declaring versions makes sense as this gives overview over versions in a graph-compliant way, as proposed by Conradi and Westfechtel (1998) and implemented in xADL 2.0 (Institute for Software Research, 2005). Version specification is also used in the UDDI extensions to automatically generate version info in the registry, as we will describe later. Dependencies between versions are useful to provide traceability of changes between versions. Our solution allows to declare the default (current) version of a service and to mark deprecated versions. We will use this information to notify service consumers about version changes.

The `<wsdlx:versions>` element is nested as the first element within the `<description>`, which is the root element of the WSDL. The code in Listing 2 shows a declaration of versions within the WSDL document. Under `<wsdlx:versions>` the complete version history is declared using `<wsdlx:version>` elements. In our example versions 1.0, 2.0, and 2.1 are declared. For each version, we declare the type of version, which can be backward-compatible or backward-incompatible. We also declare intent, which can be revision, variant, or cooperation. This classification is based on Conradi and Westfechtel (1998), but can be adapted to cover other specific requirements. Intent information is used by the server at the deployment time to select the appropriate policy related to service deployment and execution and the policy for handling older versions of the service.

Version 1.0 is declared as the initial version using `<wsdlx:initialVersion/>`. For traceability all other versions are required to specify the previous version using `<wsdlx:previousVersion/>` element with version identifier (vid) parameter. We use `<wsdlx:defaultVersion/>` to specify the current default version. This is used for service consumers, which do not provide version information, to resolve the most up-to-date version of a service, or for version-unaware consumers. It is also used for version-aware consumers to notify them about the current default version (which they can use as a signal that update to a newer version is applicable). For older versions we can use the `<wsdlx:deprecated/>` element, to notify service consumers about version deprecation. Service consumers can use deprecated versions, however service providers can retire deprecated interfaces without further notification. If a deprecated interface is

```xml
<xs:element name="vid" type="tcs:VersionID"/>
<xs:simpleType name="VersionID">
  <xs:restriction base="xs:string">
    <xs:pattern value="[0-9]{1,3}\.[0-9]{0,3}"/>
  </xs:restriction>
</xs:simpleType>
Listing 1. Version identifier declaration example.
```
retired, service consumers will be unable to bind to a service and will have to upgrade to a newer version. Service consumers should therefore interpret the deprecation notification as an explicit signal to upgrade to a newer version. An optional deprecationDate attribute can be used to denote the date until a deprecated service version will be available. Service consumers can use the deprecated version until the specified date, after the date the service interface becomes unavailable.

For each version we can include an optional <wsdlx:versionInfo> where we describe the changes in this specific version. Service consumers can programmatically query version information, such as version identifier, version info, type, intent, and other to decide which interface version to use, and to plan the development activities.

After version declarations, we have to specify each version. We do this within the <wsdlx:version> element. Here we have two options: to embed the WSDL, or to import external WSDLs. Usually we will embed the WSDL if we want to manage the versioning of a single web service, and import external WSDLs if we want to manage the versioning of a set of related web services (i.e. application that consists of several services).

The proposed version declarations also address version understanding, which is essential for service reuse. The descriptive version information element is intended for detailed description of version-specific information for the service. This element should be used with descriptive annotations of WSDL interface. Traceability of version-related changes is supported with the declaration of previous versions. Each version also specifies the deprecation status and deprecation date. This way deprecation and retirement of versions is supported. The default version tag denotes default service version. All version-related information can be stored in the UDDI registry, as we will describe later. Clients can programmatically access version information and use it for searching for specific versions of services. Our proposed approach also supports the use of language-specific approaches, such as JavaDoc, to describe version-specific behavior.

Please note that our solution works with WSDL 2.0 and WSDL 1.1. In the examples, we will use WSDL 2.0, but explain how to use WSDL 1.1 if differences exist.

5.2.1. Manage versions of single web service

We will first explain how to version the WSDL to manage the versioning of a single web service. We specify <types>, <interface>, <binding>, and <service> for WSDL 2.0. If we use WSDL 1.1 we specify <types>, <message>, <portType>, <binding>, and <service>. For the initial version, we have to specify the complete structure. For the following versions, we have two options:

- We specify the changes only. All other parts are automatically merged (reused) from previous versions. This is the default behavior for backward-compatible versions.
- We specify the complete WSDL structure. Nothing is merged from previous versions. This is the default behavior for backward-incompatible versions.

For automatic merge, we use a simple algorithm that detects the missing elements of the new version of the WSDL and reuses them from the first previous version where the elements exist. Version sequence is defined in versions declaration <wsdlx:versions>.

The merge algorithm performs static type checks. It is the responsibility of the developer to ensure that he defines all the necessary elements of the current WSDL interface version and omits only those that can be safely merged from previous versions and are type-compatible.

For automatic merge, we use a simple algorithm that detects the missing elements of the new version of the WSDL and reuses them from the first previous version where the elements exist. Version sequence is defined in versions declaration <wsdlx:versions>.

The merge algorithm performs static type checks. It is the responsibility of the developer to ensure that he defines all the necessary elements of the current WSDL interface version and omits only those that can be safely merged from previous versions and are type-compatible.

To remove specific elements from a new version and prevent that they are automatically merged from previous version when automatic merge is enabled, we specify the wsdlx:auto-Merge="false" attribute at the element declaration. It is sufficient to declare an empty element for this purpose. For example, to remove the operation checkBookPrice in the version 2.1 of the service interface while automatic merge is enabled, we specify an empty <operation/> tag (Listing 4). Please notice however that removing
of elements is only applicable with backward-incompatible versions where automatic merge is manually enabled.

Example in Listing 3 is using WSDL 2.0. We can see a specification of interface version 1.0, where a service declares an interface

```xml
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:ns1="http://example.com/">
    <message name="checkBookRequest">
        <part name="tns:checkBookRequest" type="xsd:string"/>
    </message>
    <message name="checkBookResponse">
        <part name="tns:checkBookResponse" type="xsd:string"/>
    </message>
    <portType name="checkBook">
        <operation name="checkBook" type="ns1:checkBookRequest"/>
    </portType>
    <binding name="checkBookSOAPBinding" type="ns1:checkBook">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http"/>
        <fault name="checkBookFault">
            <soap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
        </fault>
    </binding>
</wsdl:version>
```

Listing 3. Service-level versioning example.
called checkBookInt with a single operation checkBookPrice. The operation is request/response and accepts a checkBookRequest element as input and returns checkBookResponse element. In this example we show and embedded XML schema. Our approach also works with external schemas.

Version 2.0 of the service interface is backward-incompatible. It has modified the checkBookRequest element (obligatory publisher subelement is added). Because this is the only change, we can make use of auto merge functionality. To do this we have to explicitly specify the autoMerge attribute. This way we can omit all other parts of the WSDL. They will be merged from version 1.0.

If we would want to modify the interface, binding or service, we could do that. This is shown in version 2.1, where another operation checkBookRating is added to the interface. Let us assume that the checkBookRating will use the same checkBookRequest element. It will return a rating between 0.00 and 5.00 (for example 4.31). This means that we can also reuse checkBookResponse element, which is of type xsd:float. Therefore, in version 2.1 specification we only need to specify the <interface> and <binding> parts and only list the new operation. Everything else is automatically merged from version 2.0, because version 2.1 is backward-compatible with version 2.0.

In Listing 4 we show how to remove an operation while using automatic merge. In our example, we remove the checkBookPrice operation. Please be aware however that removing operations is only applicable with backward-incompatible versions where automatic merge is manually enabled.

5.2.2. Manage versions of multiple services

If we want to manage versions of a set of related web service interfaces, we can import one or more WSDL definitions. To do the imports we use standard WSDL <import> or <include>. We use <import> to import WSDL that belongs to a different namespace and <include> for WSDL that belongs to the same namespace. We can import more than one WSDL and nest them. This provides flexibility to organize complex collections of versioned services. Here the same rules for automatic backward merges apply as for service-versioning (described in Section 5.2.1). Please notice that traceability outside a WSDL file is out of the scope of this article.

Example is shown in Listing 5, where in version 1.0 we import three services. In version 2.0 we only import service number 2, which has changed. Other services are reused from version 1.0 using the auto merge algorithm.

5.3. Service endpoint mapping

To publish different versions of a service there are several options. First option is to manually specify an endpoint for each service version. This can be done in the <service> section, but this has to be done manually for each version. Because versions are usually published using a certain naming pattern, we propose an extension, which will generate service endpoint bindings automatically. To use this we include a <wsdlx:useDefaultInterfaceMapping/> element in the <wsdlx:versions/> section. We specify the endpoint mode and the pattern to which the versions are published. The URL is defined in the <service/> section. The example in Listing 6 shows a pattern where major and minor version numbers are added at the end of the URL. In this example, version 1.0 would be published as http://www.uni-mb.si/bookService/1, version 2.0 as http://www.uni-mb.si/bookService/2 and so on.

We can also specify the default attribute, which specifies whether to publish the default version of a service at the URL without version information (in our case http://www.uni-mb.si/bookService). This is appropriate for service consumers that do not specify the version and prefer to use the latest version of a service.

Alternative to endpoint mode is the parameter mode. This mode does not use distinct URLs. Rather it publishes all versions under the same URL, but uses URL parameter encoding to specify the version. An example URL for parameter mode would be http://www.uni-mb.si/bookService?major=1&minor=0. The advantage of this approach is that it only requires a single server component (such as servlet) to publish different versions, which consumes less server resources. Similar as with endpoint mode we can specify whether to publish the default version using the non-version URL. Listing 7 shows the service parameter mapping extension.

We could put the <wsdlx:useDefaultInterfaceMapping/> into an external deployment descriptor. However, we had two reasons for putting it into the WSDL. First, WSDL requires that service endpoints are specified within WSDL description under the <service> tag. We could argue that the <service> tag could also belong to an external deployment description, particularly because it contains the service endpoint URL. Endpoint URL can change, for example if a service is moved to another server, or when migrating a solution between development, test, and production environments. Second, web services are implemented in various programming languages. Only WSDL is common for all environments. If we would put <wsdlx:useDefaultInterfaceMapping/> into a deployment descriptor, this would have to be language specific, which we have tried to avoid. However, although <wsdlx:useDefaultInterfaceMapping/> is put into the WSDL file, the usual approach is to split the WSDL file into abstract interface definition (<types> and <interface> for WSDL 2.0: <types>, <message>, and <portType> for WSDL 1.1) and concrete interface definition (<bindings> and <service>). This way this issue is addressed to a certain degree.

Listing 4. Removing an operation while automatic merge is enabled.

--- Remove checkBookPrice operation while auto merge is enabled. -->

```
<operation name="checkBookPrice"
  pattern="http://www.w3.org/ns/wsdl/in-out:"
  style="http://www.w3.org/ns/wsdl/style/1.1"
  wsdlx:autoMerge="false"/>
```

Listing 5. Manage Versions of Multiple Services,

```
<wSDL:version vid="1.0"/> <!-- Version 1.0 -->
<import namespace="http://uni-mb.si/services1" 
  location="http://www.uni-mb.si/1.0/services1.wsd1"/>
<import namespace="http://uni-mb.si/services2" 
  location="http://www.uni-mb.si/1.0/services2.wsd2"/>
<import namespace="http://uni-mb.si/services3" 
  location="http://www.uni-mb.si/1.0/services3.wsd3"/>
</wSDL:versions>

<wSDL:version vid="2.0" autoMerge="true"/> <!-- Version 2.0 -->
<import namespace="http://uni-mb.si/services2" 
  location="http://www.uni-mb.si/2.0/services2.wsd2"/>
</wSDL:versions>
```

Listing 6. Automatic service endpoint mapping to distinct URLs.

```
<wSDLx:useDefaultInterfaceMapping
  mode="endpoint" pattern="/major/$minor" default="true"/>
```

Listing 7. Automatic service parameter mapping to single URL.
5.4. Operation-level versioning

When fine-grained version control is required, we can use the proposed operation-level versioning extensions. In contrary to service-level versioning extensions, which are applied to the whole service or a set of services, operation-level extensions are applied to WSDL elements. This allows to separately control the version cycle of types, messages, interface, bindings, and endpoint locations. Here some differences between WSDL 1.1 and 2.0 exist. Let us first look at WSDL 2.0.

In WSDL 2.0 we have to nest the `<wsdlx:version>` within the `<types>` and the `<interface>` declarations. The `<binding>` and `<ser-

Listing 8. Operation-level versioning in WSDL 2.0.

```xml
<wsdlx:version mode="operationLevel">
  <!-- Version declarations similar to Listing 2 -->
</wsdlx:version>

<types>
  <wsdlx:version vid="1.0">
    <wsdlx:initialVersion/>
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"/>
  </wsdlx:version>
</types>

<interface name="checkBookInt">
  <!-- Faults can be defined outside the version tag if no need for versioning -->
  <fault name="processingFault">
    <element ref="tns:checkBookPrice"/>
  </fault>
</interface>

<binding name="bookSOAPBinding">
  <interface name="checkBookInt"/>
  <soap:operation name="checkBookPrice"/>
</binding>

<service name="bookService" interface="tns:checkBookInt">
  <wsdlx:interfaceVersion vid="1.0"/>
</service>

<service name="bookService" interface="tns:checkBookInt">
  <wsdlx:interfaceVersion vid="2.0"/>
</service>

<!-- Alternatively we could use automatic endpoint mapping -->
</service>
</description>
```
vice declarations are related to the interface version, therefore we use the `<wsdlx:interfaceVersion vid=""/>` element. This is shown in Listing 8. Similar rules apply as with service-level versioning. We have to specify the initial version. We also have to specify the ordering of versions. For subsequent versions, the behavior differs for backward-compatible and backward-incompatible interfaces. For backward-compatible interfaces the default behavior is that we only have to specify the changed parts of WSDL. Other parts are automatically merged from previous versions. For backward-incompatible interfaces the default behavior is that we have to specify the complete interface and nothing is automatically merged. We can alter this behavior using the `autoMerge` attribute at the level of `<wsdlx:version>` and `<wsdlx:interfaceVersion>` elements.

The merge algorithm merges missing elements from the most recent previous version and works the same way as described in Section 5.2.1. If we use automatic merge, we can use `wsdlx:vid` attribute to explicitly define the previous versions to be merged (described in Section 5.5). Fault messages can be specified inside interface version declarations. They can also be specified outside version declarations if we do not need to version them.


Listing 10. Controlling version sequence in WSDL 2.0.


In WSDL 1.1 in addition to <types> and the interface (specified as <portType>), we can also control the version cycle of messages. We have two possibilities. First, we can manually specify message versions as shown in Listing 9. Second, we can use automatic message versioning. In WSDL 1.1 messages are often in one-to-one relationship to schema elements (that is why they have been abandoned in WSDL 2.0). In such cases automatic message versioning saves effort. It is specified using the <wsdlx:useAutomaticMessageVersioning>.

5.5. Version sequence

The default behavior for backward-compatible interfaces is that everything that is not specified in the current version is automatically merged from the most recent previous version, according to the version sequence specified by <wsdlx:initialVersion> for the first version, and <wsdlx:previousVersion> for next versions (Listing 2). For backward-incompatible interfaces, automatic merge is disabled by default, but can be activated by setting autoMerge="true" attribute.

We can also explicitly declare which version to use. To achieve this we have introduced an optional attribute for version identifier, wsdlx:vid. In WSDL 2.0 this attribute applies to the <input>, <output>, <outfault>, and <infault> elements of the <interface> specification. In Listing 10, we can see an example where we explicitly declare that we want to use the checkBookRequest element from the <types> section declared by version 1.0 for the checkBookPrice operation.

In WSDL 1.1 the wsdlx:vid attribute applies to the <input>, <output>, and <fault> elements of <portType> specification. It also applies to the <part> of the <message> specification. In Listing 11, we have declared that for the checkBookRequest element of the payload part of the checkBookRequestMsg message a version 1.0 <types> section should be used. We have also declared that for checkBookPrice operation version 2.0 of checkBookRequestMsg should be used.

5.6. Extensions for UDDI registry

So far, we have presented extensions to WSDL. These extensions are sufficient in environments where no service registry is required. In complex environments, UDDI-compliant service registries are common. UDDI does not support versioning. Therefore, we propose extensions to UDDI. A UDDI registry should allow registering version information of a service. It should allow specifying the default version, so that service consumers that acquire for a service without version information can get the current version. Registry should also support marking versions as deprecated. For service consumer queries, registry should support acquiring a specific service version, acquiring default (latest) version, and notifying service consumers about version changes and version deprecation.

To achieve this, we propose specific extensions to the UDDI model, through which version information will become available in all relevant operations, such as save_service(), find_service(), and get_serviceDetail(). To understand the proposed extensions let us first briefly look at the UDDI model structure. UDDI model consists of five basic data types: publisherAssertion, businessEntity, businessService, bindingTemplate, and tModel (UDDI, 2004). BusinessEntity contains data about party publishing a set of services and publisherAssertion contains data about relationships between parties. These two data types are not of interest for versioning. BusinessService contains descriptive information about a service and bindingTemplate contains technical information about a service. BindingTemplate is an ancestor of businessService. Both are related to the tModel, which contains specifications of service.

Version information has to be incorporated into the businessService and tModel data structures. All these data structures are composed of several subelements. One of them is the categoryBag. CategoryBag contains a list of categories that describe a specific service aspect.

5.6.1. BusinessService and tModel extensions

We propose to extend the categoryBag with specific parameters required for version information. To achieve this we have introduced an UDDI extension namespace uddix. We propose the following specific key reference extensions within the categoryBag:

- uddix:version, which specifies the version number in the defined version identifier format (we use a simplified major.minor version identifier for examples in this paper).
- uddix:type, which specifies the backward compatibility or incompatibility of service interface.
- uddix:intent, which specifies the intent of the service interface version, which can be revision, variant, or cooperation.
- uddix:defaultVersion, which denotes whether this version is the default version of the service. Allowed values are true and false.
- uddix:deprecated, which specifies whether the current version is deprecated. Allowed values are true and false.
- uddix:deprecationDate, which specifies the date until a deprecated service version will be available.
- uddix:versionInfo, which describes the version-related information about the service interface.

Listing 12 shows the proposed categoryBag extensions (in bold).

5.6.2. Service publisher extensions

Table 1 shows the list of service publisher UDDI API operations and the necessary extensions achieved through tModel and businessService extensions. The delete operations do not require version extensions, because a unique key of the item for deletion has to be specified. Get and set operations are related to service publishers, where versions are not applicable.

<table>
<thead>
<tr>
<th>UDDI operation name</th>
<th>Version extension through tModel</th>
<th>Version extension through businessService</th>
<th>Version extension not applicable</th>
</tr>
</thead>
<tbody>
<tr>
<td>save_binding</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>save_business</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>save_service</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>save_tModel</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>delete_binding, delete_business,</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>delete_publisherAssertions,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>delete_service,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>delete_tModel</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>get_assertionStatusReport, get_publisherAssertions, get_registeredInfo, set_publisherAssertions</td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
5.6.3. Service consumer extensions

Table 2 shows the list of service consumer inquiry UDDI API operations and the necessary extensions achieved through tModel and businessEntity extensions.

We also propose two mandatory service consumer behavior recommendations. First, service consumers can inquire for default service version using the uddix:defaultVersion key set to true. In such scenario, the UDDI registry has to resolve the default version of a service. Second, service consumers are required to check the uddix:defaultVersion, uddix:deprecated, and uddix:deprecationDate response values at each query to the registry and act upon those values to ensure they are updated accordingly.

5.7. Mapping to Java and other languages

The proposed extensions to WSDL and UDDI are designed in a way that they can be mapped to any programming language or environment used to implement services. The extensions are applicable for top-down development where the WSDL is defined first and then the source code is implemented. The extensions are also applicable for bottom-up development where the service is first implemented in a programming language and WSDL is generated from the code. The proposed extensions are well suited for modern environments, which use code annotations for web service development (such as Java and .NET).

To use the proposed extensions in Java, we have developed annotation extensions for Java using JAX-WS (Java API for XML-Based Web Services, JSR 224, 2007). Proposed WSDL extensions map straightforward into the extended Java annotations. To implement web service version management in Java we propose extensions to the @WebService and @WebMethod JSR 181 metadata annotations (JSR 181, 2006) and to @WebServiceClient and @WebServiceProvider JSR 224 metadata annotation. The proposed extensions are shown in Listing 13 (in bold) and contain attributes that accommodate the proposed WSDL extensions, as described previously in this article.

The proposed annotation extensions can be used by the service endpoint implementations to set and read the values. They can be used by the service consumers to read the values. To version Java source code artifacts required for web service implementation, we use Java package versioning, a standard feature of Java platform (Sun Microsystems, 2002). To use the proposed UDDI version extensions from Java, we have to define a custom external taxonomy in JAXR. In JAXR categoryBag is mapped to a Collection of Classification instances. Each keyedReference in categoryBag is mapped into a Classification object. The external taxonomy has to be defined as a JAXR classification scheme. It has to include the concepts from our extension proposal, as shown in Listing 14. After including this taxonomy in the JAXR run-time environment using the com.sun.xml.registry.userTaxonomyFilenames property, the versioning extensions can be used in Java code in the same way as standard UDDI classifications. This means that Java service consumers can invoke UDDI operations with version information.

5.8. Prototype implementation

To implement the proposed WSDL extensions for versioning, we have several options. We need to extend the tools, which are used by specific programming environment for generating source code out of WSDL interfaces. We also need to modify the server for hosting web services to understand the WSDL version extensions. The extended server has to support version-aware and version-unaware service consumers. For version-aware service consumers it has to propagate version information to the client. For version-unaware consumers it has to convert version-extended WSDLs to standard WSDLs based on the default version declarations. To implement the UDDI version extensions, we need to implement them for a specific UDDI server. We also need to implement the extensions to the API used to communicate with the UDDI. In case of Java this is JAXR (described in Sections 5.6 and 5.7). Version-aware UDDI clients will deal with extended version information, while version-unaware clients are supported in a standard way, where UDDI server uses the default version annotation to provide the appropriate service interface version.

We have used the proposed versioning extensions in real-world web service development on Java EE platform. We have developed JAX-WS wsimport and wsgen Ant tool extensions to support the code generation from proposed WSDL versioning extensions (wsimport) and WSDL generation from annotated Java code (wsgen). We have modified the server for hosting web services. We have also developed an Eclipse plug-in to support visual development of version-extended WSDLs. We have implemented the proposed UDDI extensions on the UDDI4J 2.0.5 open-source implementation (UDDI4j, 2006). We have also developed a tool for auto-

<table>
<thead>
<tr>
<th>Table 2</th>
<th>Extensions to service consumer inquiry UDDI API.</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDDI operation name</td>
<td>Version extension through tModel</td>
</tr>
<tr>
<td>find_binding</td>
<td>X</td>
</tr>
<tr>
<td>find_service</td>
<td>X</td>
</tr>
<tr>
<td>find_tModel</td>
<td>X</td>
</tr>
<tr>
<td>find_business, find_relatedBusinesses</td>
<td></td>
</tr>
<tr>
<td>get_bindingDetail</td>
<td>X</td>
</tr>
<tr>
<td>get_serviceDetail</td>
<td>X</td>
</tr>
<tr>
<td>get_tModelDetail</td>
<td>X</td>
</tr>
<tr>
<td>get_businessDetail, get_operationalInfo</td>
<td></td>
</tr>
</tbody>
</table>

package naming conventions an appropriate transformation has to be done first (for example, version “1.0” is transformed into package name “_1_0”).

In a similar way, the annotations could be extended for the .NET framework. Assemblies in .NET support versioning using manifests, which could be used for source code versioning (MSDN, 2008). XML namespaces are transformed into language namespaces (C# for example), where similar rules have to be defined as for Java for incorporating version identitfies into namespaces.

To access UDDI from Java, Java API for XML Registries is used (Sun Microsystems, 2002). To use the proposed UDDI version extensions from Java, we have to define a custom external taxonomy in JAXR. In JAXR categoryBag is mapped to a Collection of Classification instances. Each keyedReference in categoryBag is mapped into a Classification object. The external taxonomy has to be defined as a JAXR classification scheme. It has to include the concepts from our extension proposal, as shown in Listing 14. After including this taxonomy in the JAXR run-time environment using the com.sun.xml.registry.userTaxonomyFilenames property, the versioning extensions can be used in Java code in the same way as standard UDDI classifications. This means that Java service consumers can invoke UDDI operations with version information.

5.8. Prototype implementation

To implement the proposed WSDL extensions for versioning, we have several options. We need to extend the tools, which are used by specific programming environment for generating source code out of WSDL interfaces. We also need to modify the server for hosting web services to understand the WSDL version extensions. The extended server has to support version-aware and version-unaware service consumers. For version-aware service consumers it has to propagate version information to the client. For version-unaware consumers it has to convert version-extended WSDLs to standard WSDLs based on the default version declarations. To implement the UDDI version extensions, we need to implement them for a specific UDDI server. We also need to implement the extensions to the API used to communicate with the UDDI. In case of Java this is JAXR (described in Sections 5.6 and 5.7). Version-aware UDDI clients will deal with extended version information, while version-unaware clients are supported in a standard way, where UDDI server uses the default version annotation to provide the appropriate service interface version.

We have used the proposed versioning extensions in real-world web service development on Java EE platform. We have developed JAX-WS wsimport and wsgen Ant tool extensions to support the code generation from proposed WSDL versioning extensions (wsimport) and WSDL generation from annotated Java code (wsgen). We have modified the server for hosting web services. We have also developed an Eclipse plug-in to support visual development of version-extended WSDLs. We have implemented the proposed UDDI extensions on the UDDI4J 2.0.5 open-source implementation (UDDI4j, 2006). We have also developed a tool for auto-
matic registration of services into the UDDI registry using version- ingestion extensions. This tool gathers version information from WSDLs and registers it in the registry, as described earlier in this article.

5.9. Example

In this section, we will demonstrate how to use the proposed versioning extensions on a simple example. We will develop a CheckBook service, which we have described in the previous sections of this article. We will use the top-down development, starting with WSDL and generating source code artifacts out of WSDL (contract-first development). Let us briefly describe the implementation aspect and the deployment and run-time aspects.

5.9.1. Implementation aspect

Let us suppose that we first create the version 1.0 of the CheckBook WSDL, as defined in Listings 2 and 3. To develop the service implementation using Java and JAX-WS, we use the wsimport tool to generate the implementation side artifacts: Java interface, implementation class, and schema classes. We also use wsimport to generate the client side artifacts: client class, proxy class, interface, and schema classes. As already discussed, the service artifacts will map to Java packages with added version identifier. For this example, we have selected the following mapping for the version identifier: si.uni_mb.example.[vid]. For version 1.0 the package name is si.uni_mb.example._1.0. Please notice that we have other possibilities as to where to include the version information in the package name (Section 5.7). All generated artifacts use the annotation extensions for version information.

To implement the service, we have to implement the service implementation class, and pack the service into the war archive to deploy it on the server. A part of the war archive is the manifest, which has to contain the version information. The development of service implementation is discussed in Section 5.9.3.

To implement the client, we have two options. Clients for services can be developed statically or dynamically. A static client is a client, which uses generated proxy classes to access a service. These classes are usually generated by a tool from a WSDL (wsimport in our case). We need to implement the client implementation class, which will use the generated proxy. The generated proxy contains version information in the annotations. This way the client specifies the version of the service it is intended to use.

A dynamic client binds to the WSDL at run-time, parses the service description and can invoke service operations dynamically without the need to statically generate artifacts and compile them. The development of the client is discussed in Section 5.9.4.

In the next iteration, we create version 2.0 of the service. First, we modify the WSDL and add the description of version 2.0. Then we follow the same steps as for version 1.0. We generate service implementation artifacts and client artifacts using wsimport tool, which are mapped to the si.uni_mb.example._2.0 package. If we use autoMerge, which is the case in our example, then the generated implementation class for version 2.0 delegates the methods that have not changed from previous version to the implementation class of the previous version. For version 2.1 we repeat the same steps.

If WSDL would already contain the definition of all versions, we could use the -v switch of the wsimport to specify which version to generate. We could specify “all” to generate all versions. If WSDL contains more than one version then the default version is generated if we do not specify the version. Fig. 1 shows the implementation aspect for this example.
Please notice that our approach also supports the bottom-up development, where we first write the Java source classes and interfaces and generate WSDL out of source code. For this approach, we use the version-extended wsigen tool and have to follow the same package and class naming conventions as in the top-down approach.

5.9.2. Deployment and run-time aspects

The deployment and run-time aspects are shown in Fig. 2. It shows the service war, which is deployed on the server. It contains the WSDL interface, the implementation artifacts in Java, and the manifest. Version information in war manifest is used for deployment time version checking on the server.

For run-time versioning aspects, the version information in the WSDL and in the class annotation extensions is used for the service and the client. Version extensions support both static and dynamic clients. The static client can bind to the service directly using the URL, or it can use the UDDI registry. This scenario is shown in Fig. 2, where the static client queries the UDDI registry for a specific version of the service, and then binds to the specific version of the service and invokes the operations.

Fig. 2 also shows a dynamic client, which first queries the UDDI, then retrieves the WSDL for the specific version of the service, then dynamically generates the stub, and finally binds to the service version and invokes operations. Dynamic clients in Java can be implemented using the Dispatch API of the JAX-WS.

Version extensions support version-unaware and version-aware clients. Version-unaware clients are such that do not understand the version extensions. Such clients get the default version of the service. Version aware clients can select which version they want to use. Fig. 2 shows version-aware clients only.

5.9.3. Service implementation

We will first describe how to develop the service implementation. After using the version-extended wsimport tool, the following artifacts for the service implementation will be generated: CheckBookPriceInt.java and CheckBookPriceImpl.java. The Java interface is generated in the package si.uni.mb.example._1.0 and the schema classes in the package si.uni.mb.example._1.0.schema. Additionally, the annotation extensions with version info are automatically generated, as shown in Listing 15.

To develop the service implementation we have to create a Java implementation class that implements this interface. The extended wsimport can generate a skeleton CheckBookPriceImpl.java class. The responsibility of the developer is to implement the methods, in our case the CheckBookPrice operation.

For version 2.0, the WSDL needs to be extended with the section, which describes the additions in version 2.0 (shown in Listings 2 and 3). In our example version 2.0 is a backward-incompatible revision that uses the autoMerge option to specify the changes only and automatically merge the other unchanged parts. In our case the changes in version 2.0 are related to the schema. To specify which version of the service should be generated, we use the −v switch. This way we specify the required version, in our case 2.0. If we omit the −v switch the default version is generated. We can also specify "all" to generate all versions.

After using the wsimport new artifacts for service implementation will be generated. This time the package si.uni.mb.example._2.0 for the interface and the package si.uni.mb.example._2.0.schema for the schema will be used. Because the autoMerge has been used, the skeleton implementation class will automatically delegate the methods that have stayed unchanged to the previous version of the implementation class (if such unchanged methods exist). The developer can override this default behavior. The versioning of Java implementation artifacts is out of the scope of this article and can be solved using a version management system. Listing 16 shows the CheckBookPriceInt.java interface for version 2.0.

For version 2.1, the procedure is very similar. The WSDL has to be extended with the new version declaration. In our case version 2.1 is backward-compatible variant. After using the wsimport tool, the artifacts are generated in the packages si.uni.mb.example._2._1 and si.uni.mb.example._2._1.schema. The interfaces are annotated with version extensions in the same way as in previous examples.
Version information is used in the manifest when the classes are packed into the corresponding .war file for deployment on the server. The version information in the manifest can be used for deployment time version information. Example manifest file for version 2.1 for our example is shown in Listing 17.

```
package si.uni_mb.example._1._0;
import ...;

@WebService(name = "checkBookPriceInt",
            targetNamespace = "http://www.uni-mb.si/example",
            version = "1.0",
            initialVersion = Boolean.TRUE,
            deprecated = Boolean.TRUE)
@SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
@XmlSeeAlso({
    ObjectFactory.class
})
public interface CheckBookPriceInt {
    @WebMethod(operationName = "checkBookPrice",
                version = "1.0",
                initialVersion = Boolean.TRUE)
    @WebResult(name = "checkBookPriceResponse",
                targetNamespace = "http://www.uni-mb.si/example/schema",
                partName = "payload")
    public double checkBookPrice(
        @WebParam(name = "checkBookPriceRequest",
                   targetNamespace = "http://www.uni-mb.si/example/schema",
                   partName = "payload")
        CheckBookPriceRequest payload);
}
```

Listing 15. CheckBookPriceInt.java interface version 1.0 for service implementation.

5.9.4. Client implementation

To develop a static client in Java and JAX-WS, we use the wsimport tool. It generates the BookService client dynamic proxy class, which can be used by the client to access the service. We have to specify the \(-v\) switch of the wsimport tool to specify the version for the service client. If omitted, the default version is used. Client artifacts are also put into the packages that are supplemented by the version number. In our case, this would be `si.uni_mb.example._2._1` for client version 2.1. As already mentioned, the default package naming can be changed. The generated dynamic proxy class for the client is annotated with the version information, as shown in Listing 18.

Clients can bind to the service directly suing the service URL, or they can query the UDDI registry. In both cases, the version information within the @WebServiceClient annotation can be used to simplify the work.
5.10. Proof-of-concept

We have tested our proposed solution in two large real-world IT environments: in a telco company with more than 2000 employees and in a power distribution company with more than 500 employees. The telco operator has started to introduce SOA in 2005. Until 10 months ago, when we started our testing they had 44 distinct web services available. Each service had 3 to 4 versions (in average 3.25) giving a total of 143 published web service endpoints. Hundred and seven service endpoints were implemented on Java platform. The other 36 endpoints were implemented on SAP and some other proprietary applications. So far, we were able to migrate all Java web services to use our versioning solution. This reduced the number of endpoints to 69. This number does not include new developed services, which have used our versioning solution from the start. The considerable reduction of the number of services has resulted in sounder architecture. It has improved the service reuse rate. Until 10 months ago only 36.4% of services have been used more than once. Today 65.9% of services are reused in two or more applications. The effort required for deployment and service maintenance has been reduced. IT governance procedures have been simplified and are now more efficient and take less time.

The developers also report that the service development and maintenance time has been reduced. We have measured development and maintenance time improvements. Improvements between 19% and 24% in scenarios, where three or more versions of a single service had to be managed, have been achieved.

The power distribution company started to introduce SOA in 2006. We have started to test the versioning solution 12 months ago. Until then the company had 28 distinct services. Most of them had two versions, some three. In total, they had 59 published web service endpoints. All services have been implemented on the Java platform. Using our versioning solution, we reduced the number of service endpoints to 28. Two services still do not use our solution because source code has not been available from the vendor. The results have been similar to the telco operator. In addition to improving the architecture, higher reuse has been achieved. Twelve months ago 53.8% of services have been reused more than once and seven services are reused eight times or more. Today each service has at least three versions. Some services already have up to 11 versions. The company has also improved the development-time between 23% and 26% for services with three or more versions. SOA governance effort has been reduced. The administrators have reported reduced service maintenance time.

6. Related work and discussion

So far, versioning of web services has not been addressed in a way comparable to the approach proposed in this paper. Non-existing support for versioning has been identified as an important problem (Vinoski, 2004). Several authors have proposed workarounds, such as (Kenyon, 2003; Irani, 2004; Brown and Ellis, 2004; Peltz and Subbarao, 2004; Posley, 2006), summarized in Sections...
tion 4. They however cannot replace the full-featured solution, as the one proposed in this article.

We have found only a few publications that address web service versioning. Kaminski et al. (2006a,b) addressed the problem of simultaneous deployment of multiple versions of a web service in the face of independently developed unsupervised clients. They proposed to use a form of a design technique called chain of adapters to make version-related reconfiguration tasks safe. Our approach is based on explicit version information and version-aware clients. Ponnekanti and Fox (2004) studied substitution of one vendor service for another, assuming that these services are derived from a common base, and addressed the analysis of incompatibilities. We think that our approach is complementary to Ponnekanti and Fox (2004) and Kaminski et al. (2006a,b). M'Bareck and Tata (2007) proposed a novel approach to enhance web service discovery based on, among others, requesters' preferences. Although his approach extends the service discovery, it does not address service versioning and does not extend service registries to include versioning information, as we do.

Versioning has also not been addressed by distributed object models and component models, predecessors of web services. This has been analyzed by Stuckenholz (2005). CORBA, RMI, and EJBs do not provide support for versioning. Only COM provides a multi-interface feature, which can also be used for versioning of interfaces of COM components, as described by Stets et al. (1999). COM multiple interfaces have not been designed specifically for versioning. Interfaces in COM are denoted by unique ids. The COM approach differs considerably from our proposal. Our proposal provides the ability to track version changes, to organize interface into versions, to gradually deploy different versions of the same service, and to use version information in the registry.

Some solution for versioning in distributed object and component models have been proposed. Senivongse (1999) presented a model to alleviate the problem of evolving services by making different distributed service versions substitutable. He uses a media-model to alleviate the problem of evolving services by making different models have been proposed. Senivongse (1999) presented a registry. Serei and Fox (1999) presented a versioning of the same service, and to use version information in the registry.

Our proposal provides the ability to track version changes, to partially deploy different versions from these articles. Above-mentioned articles however focus on the development-time versioning (versioning of artifacts that developers deal with) only. At that time, run-time versioning has not been very important, because applications have been built as self-contained packages. In today’s distributed, interoperable, loosely-coupled environments, such as SOA, run-time versioning is increasingly important. Initial ideas of run-time versioning, at that time called instance versioning, have been mentioned by Katz (1990) and Talens et al. (1993). Our approach addresses development-time and run-time versioning in an integrated manner.

Westechelet et al. (2001) presented a uniform version model and support architecture for software configuration management. They defined a base model that is built on a small number of concepts. Specific version models may be expressed in terms of this base model. This is conceptually similar to our approach that also proposes an initial version, which is a base for other versions. Our approach addresses the specific requirements of web services, which operate in loosely-coupled environments and require binding to a specific version of the service at run-time (for example in BPEL orchestrations). Our approach also addresses service interface versioning together with the registry versioning.

7. Conclusion

In this article, we have proposed a solution to support versioning of web services. In web service architectures and in SOA support for versioning is very important. Web services however do not provide explicit support for versions and all proposed workarounds have considerable disadvantages.
Our proposed solution addresses versioning of web services at the time of development. It also addresses versioning support at run-time, such as allowing dynamic bindings to specific web service versions. Our solution consists of extensions to WSDL, to UDDI, and to Java annotations. It supports coarse-grained versioning, where we version a single WSDL or a set of related WDSDLs. It also supports fine-grained versioning, where we can individually version different parts of WSDL interface, such as types, messages, interface, bindings, or service endpoints. To improve consistency and to simplify service endpoint mapping for different versions, we propose an automated approach. With our solution, we can define the order of versions to achieve traceability of changes. We can mark the default version of a service, which is useful for version-aware service consumers. We can also denote deprecated versions and notify service consumers about deprecation. For backward-compatible interfaces, we have proposed automatic merge with previous versions, which enables developers to define only new and modified parts of a new version.

To incorporate version information into the service registry, our solution extends the UDDI. It introduces new key references to the category bag, which is a part of the UDDI information model used in the business service and tModel descriptions. This way version information becomes available in all relevant operations, such as save_service(), find_service(), and get_serviceDetail(). Our solution also proposes mechanism for notifying service consumers about version changes and for querying the default and specific service versions. To use version information for developing service implementation classes, our solution introduces Java JSR 181 and JSR 224 annotation extensions for JAX-WS and UDDI classification extension. In a similar way, we could extend other languages, such as C#.

To verify our solution, we have developed the necessary tools and tested the solution in real-world IT environments in two large companies, both introducing SOA. In both companies, we have confirmed the anticipated advantages, such as considerable reduction of the number of service endpoints due to better version management, significantly improved service reuse rate, and reduced development-time for services. For example, development-time has been improved between 19% and 26% for scenarios where three or more versions of a service had to be maintained simultaneously. We have also achieved reduced service maintenance and administration effort, improved SOA governance and reduced effort, and improved enterprise architecture.

References


Matjaz B. Juric, Ph.D., is Associate Professor at the University of Maribor and the head of SOA Competency Centre. He has authored several SOA books, such as SOA Approach to Integration, Business Process Execution Language, and BPEL Cookbook (award for best SOA book in 2007). Matjaz has been SOA consultant for several large companies. He has contributed to SOA Maturity Model and performance optimization of RMI-IIOP, etc. He is also a member of the BPEL Advisory Board.

Sasa Ana is a researcher at University of Ljubljana where she is finishing her Ph.D. She has dedicated the last five years to the field of SOA. Ana has worked on a number of IT projects for government agencies and large companies and has specialized knowledge and experience in the technical, business and organizational aspects of SOA.

Bostjan Brumen, Ph.D., is a researcher at the University of Maribor. His research interests are focused on data mining, data classification, and data security. He has published on several conferences and journals.

Ivan Rozman, Ph.D., is the rector of University of Maribor and head of the Institute of Informatics. His research interests include engineering management, software development methodologies, software process, and software quality. He is the leader of COT—Object Technology Centre.