A Feature-based Versioning Approach for Assessing Service Compatibility

Marcelo Yamashita, Karin Becker, Renata Galante

Universidade Federal do Rio Grande do Sul, Brazil
{marcelo.yamashita, karin.becker, galante}@inf.ufrgs.br

Abstract. The ability of dealing with changes is one of the driving forces behind the adoption of service-oriented computing. Service evolution requires sound strategies to appropriately manage versions resulting from changes during service lifecycle. Compatibility addresses the graceful evolution of services by considering the effects of changes on client applications. However, providers cannot always guarantee that the necessary changes yield compatible service descriptions. In practice, providers describe the changes in release notes, focusing on the explicit changes, very often disregarding their cascading effects. Thus, typically it is the responsibility of client's developers to assess the extent of the change and their impact in their particular usage scenario. This work addresses service evolution on a finer grain manner, referred to as features. It describes a versioning model and a compatibility assessment algorithm at service feature level, which allows the identification of changes impact points, and propagation effects, as well as the assessment of changes' compatibility. The article also reports an experiment based on a real service, which explores the versioning model to detect implicit and explicit changes, and validates the algorithm to assess the compatibility of these changes.

Categories and Subject Descriptors: H. Information Systems [**H.2 Database Management**]: Miscellaneous - Service versioning

Keywords: Service evolution, service versioning, service compatibility

1. INTRODUCTION

The adoption of service-oriented computing is highly motivated by the ability of dealing with the inevitable changes. SOA (Service-oriented architecture) and web services have become *de facto* industry standards for the development of loosely-coupled, platform-independent enterprise applications. Like traditional software, services are also subject to constant changes, requiring appropriate strategies to support and manage multiple versions throughout their lifetime. Service evolution management encompasses the creation, maintenance and decommission of different versions in a service provider environment, which leads to the maintenance of several concurrent versions [Papazoglou et al. 2011]. In order to minimize the impact on clients, a common approach to manage service versions from the provider perspective, is the versioning of service interface description [Andrikopoulos et al. 2012].

Service interface description exposes the service version as a *contract* established by the provider, which guides clients on how to access the service functionality. However, current notations for service interface description, including the standard WSDL/XSD, do not properly handle versioning [Andrikopoulos et al. 2012]. Typically, the whole interface description document is versioned, despite the fact that many service features (e.g. types, operations, message calls) remain unchanged. This practice leads to difficulties on recognizing and measuring the actual impact of a change, especially regarding each particular usage scenario. In the absence of proper support, very often providers publish new versions using unique version numbers or timestamps, together with release notes that hopefully will help clients to adjust to changes (e.g. eBay, Google, Amazon). Release notes usually describe

This research is financially supported by FAPERGS, CNPq and CAPES - Brazil.
Copyright©2011 Permission to copy without fee all or part of the material printed in JIDM is granted provided that the copies are not made or distributed for commercial advantage, and that notice is given that copying is by permission of the Sociedade Brasileira de Computação.

the explicit changes (e.g. changes on schema types of service calls), but fail to properly identify how changes propagate their effect throughout the entire service [Zou et al. 2008] [Fokaefs et al. 2011]. For instance, if a change is applied to a type that is referenced (directly or indirectly) by an operation, then this operation is implicitly changed due to the ripple-effect. As interface description versions (and corresponding release notes) are traditionally large documents, the task of finding whether the introduced changes impact client applications is hard, labor-intensive and error-prone [Zou et al. 2008] [Becker et al. 2008].

Service change management requires mechanisms for the identification and classification of changes in order to plan for compensatory actions for their side effects. Thus, service stakeholders need to *quantify* the scope of changes and *qualify* their impact. In other words, they need to easily identify explicitly and implicitly changed features in a new version, if compared to previous ones, and whether these features were changed in a way clients are not broken. The need for a smaller grain of change representation is highlighted in different works, for purposes such as client synchronization [Zou et al. 2008], change impact quantification [Wang and Capretz 2009], accurate recognition of changes [Fokaefs et al. 2011], and usage oriented compatibility assessment [Yamashita et al. 2012].

Compatibility is a central issue on service evolution, because its assessment can provide valuable information regarding the effects of changes on client applications [Becker et al. 2008]. However, traditional compatibility approaches (e.g. [Fang et al. 2007]) are also document-oriented, which means that the assessment of compatibility among different versions focuses on the worst-case of total service compatibility. Establishing compatibility relationship between service versions does not necessarily capture the (in)compatibility impact of the change, because client applications are not bound to the whole service as described by the interface, but rather to specific features they provide. Related work [Yamashita et al. 2011] [Yamashita et al. 2012] [Ponnekanti and Fox 2004] [Andrikopoulos et al. 2008] suggests the benefits of assessing compatibility in terms of specific client usage.

The goal of this work is to detect explicit and implicit changes between service versions in order to pinpoint incompatible changes. The contribution of this article is twofold: a) it describes a feature-oriented versioning model that allows to version services, operations and types individually, whilst maintaining their relationships and b) a compatibility assessment algorithm for services versioned according to this model. The versioning model handles the structural service description from the abstract perspective of service features, which correspond to fragments of a service interface description document. This small-grained versioning model enables to locate and quantify the impact of changes, whereas the assessment of compatibility at feature level enables to qualify it. An experiment was developed using a real eBay web service, and it revealed that less than 5% of detected changes correspond to explicit changes. The remaining 95% of changes is a consequence of the ripple-effect, and they are difficult to track because most of them are not covered by the corresponding release notes. The compatibility of these changes were also successfully assessed, and we realized eBay compatibility rules are more flexible than the ones suggested in the literature [Brown and Ellis 2004] [Frank et al. 2008] [Papazoglou et al. 2011] [Fang et al. 2007].

The approach discussed in this article lays foundation for a wide spectrum of applications in the context of service evolution. The analysis of change impact propagation, which is a straightforward consequence of feature-oriented versioning, allows the quantification of the ripple effect of changes [Wang and Capretz 2009]. It could also support the automatic creation of more detailed release notes based on usage analysis [Zou et al. 2008]. Compatibility assessment at feature level enables service evolution based on usage profiles [Yamashita et al. 2012], reduction of provisioned versions based on proxy redirections [Frank et al. 2008], and versions load balance management [Treiber et al. 2009].

The remaining of this article is structured as follows. Section 2 describes related work. Section 3 presents the feature-oriented versioning model and Section 4 details the corresponding compatibility assessment algorithm. Section 5 reports experiments in a real case scenario. Conclusions and future work are addressed in Section 6.

2. RELATED WORK

The concept of versioning was initially proposed for controlling design evolution and co-authoring on computer aided design [Katz 1990] and software engineering [Westfechtel et al. 2001]. In those environments, versioning is applied to files, such that different alternatives or revisions of a document (e.g. source code, electronic models, product descriptions) are stored on different files. On the context of database systems, versioning is applied for managing the evolution of different elements of the data. A distinguished functionality of the version concept appears when applied to structured data and managed by database systems. In this context, it is possible to control the evolution of different elements of the data depending on the respective data model (e.g. relations, tuples, columns for relational model and classes, attributes, relationships, methods for object oriented model). The versioning concept allows the existence of several alternatives for the evolution of database. With the migration of such database applications to a Web environment, they adopt the XML language as a standard format for representation and exchange of their internal data. Such applications require a mechanism for the representation and manipulation of the evolution of the content of a XML document that goes through modifications along time. Recent examples are XML version management applications [Mozafari et al. 2010][Brahmia et al. 2011][Cavalieri et al. 2011].

While most related work focuses on different aspects of the concepts, designing models, and processing of versions efficiently, there is currently a lack of notations and mechanisms for effective version management in the services domain [Papazoglou et al. 2011]. Current notations for service description (i.e. WSDL/XSD) do not properly handle versioning. Common approaches for service versioning include the use of XML namespaces for each version that potentially breaks the client; version identifiers for unambiguously naming versions; or a combination of these [Brown and Ellis 2004] [Andrikopoulos et al. 2012]. However, all these approaches address the versioning of the entire interface description document, which leads to difficulties on measuring the actual impact of a change, either for locating the change [Zou et al. 2008] or assessing its compatibility [Fokaefs et al. 2011]. A finer-grain versioning model based on the object-oriented paradigm is introduced in [Becker et al. 2008], but it does not fit W3C current standards. An abstract model for the description of services is described in [Andrikopoulos et al. 2012], which details all components of a WSDL service description, but it is targeted at the formal verification of the compatibility of service interface versions, not addressing the service versioning problem.

Different works highlight the need for the easy detection of changes and assessment of their impact. An approach for generating customized release notes is described in [Zou et al. 2008], which includes a service delta analyzer for detecting explicit changes between service versions represented as WSDL documents. This approach is based on the automatic detection of differences in XML descriptions, and it does not address the ripple effect of changes (implicit changes). A service dependency model to quantify the ripple effect of changes in SOA is reported in [Wang and Capretz 2009]. It assumes that changes are explicitly annotated in every new WSDL service description, such that they can be detected and their ripple effect in other services can be calculated. For developing an empirical study of web service evolution, a change detection method based on edition distance and clustering was proposed in [Fokaefs et al. 2011]. This method measures the similarity of a service operation in two versions of a service. None of these approaches address the compatibility of changes.

Previous works [Brown and Ellis 2004] [Fang et al. 2007] [Papazoglou et al. 2011] discuss backward compatibility rules, i.e. changes that can be applied on service descriptions that do not break client applications. However, the set of backward compatible changes is very restricted: a) addition of new operations to an existing WSDL interface, and b) addition of new XML schema types that are not contained within previously existing types. These are considered the core cases for compatibility in most works that address functional components for dealing with effects of changes in the SOA (e.g. [Frank et al. 2008] [Treiber et al. 2009] [Leitner et al. 2008] [Fang et al. 2007]). More recent works propose less restrictive rules, such as input compatibility [Becker et al. 2008] or T-Shape changes

[Andrikopoulos et al. 2012]. An algorithm that automatically detects the compatibility of service descriptions is discussed in [Becker et al. 2008] [Becker et al. 2011], but it assumes object-oriented service descriptions. A conceptual framework for formal compatibility assessment is discussed in [Andrikopoulos et al. 2012], but this work does not include an algorithm for automatic verification. Another algorithm for automatic compatibility detection is described in [Ponnekanti and Fox 2004], which examines the interoperability of a specific client with regard to the aspects of a service it uses, in order to find substitutable services. This approach is restricted to the perspective of a particular client application, but it is one of the pioneers is mentioning the advantages of a finer-grain unit for service versioning. Similar benefits are mentioned for service contract compliance analysis [Andrikopoulos et al. 2008] and discovery of usage profiles for usage-oriented assessment of change impact [Yamashita et al. 2011] [Yamashita et al. 2012].

The work described in this article complements the above mentioned works as follows: a) it proposes a feature-oriented versioning model for the management of service evolution, which is adapted to the W3C service description standards, b) it discusses how to automatically detect differences in WSDL descriptions and version these differences according to this finer-grained versioning model, and c) it adapts the compatibility assessment algorithm reported in [Becker et al. 2008] [Becker et al. 2011] to the feature-oriented version model. In doing so, explicit changes are automatically detected, as well as their ripple effect, and the specific compatibility of each individual change can be assessed. The result of this work supports various applications, such as service evolution management based on usage profiles [Yamashita et al. 2011] [Yamashita et al. 2012] [Silva et al. 2012], reduction of provisioned versions based on proxy redirection [Frank et al. 2008], and load balance management among implemented versions, which is proceeded by a finer-grain deployment [Treiber et al. 2009].

3. FEATURE-ORIENTED VERSIONING MODEL

This section describes a feature-oriented versioning model, which aims at supporting service version management. This model suits a number of different applications [Frank et al. 2008] [Zou et al. 2008] [Frank et al. 2008] [Wang and Capretz 2009] [Treiber et al. 2009] [Yamashita et al. 2012] [Silva et al. 2012], but this article focuses on its contribution for measuring the scope of changes and assessing compatibility on a fine-grained manner. For this purpose, a service description is abstracted as a composition of operations that can be requested by client applications. Through these operations, clients and service exchange messages that are bound to a particular format, which is recursively described by schema elements, i.e. data *types*. Because *services*, *operations* and *types* represent the relevant aspects that describe service functionality [Fang et al. 2007], these three concepts are referred to as service *features* [Lee et al. 2002]. This abstract representation of services as features and their relationship is described in Figure 1 using a UML class diagram.

The idea of feature versioning is to provide the abstract management of the different parts of the interface description in order to version only the changed features, rather than the entire service. Each feature corresponds to a portion of a WSDL/XSD service description document. Features are versioned separately, and their dependency relationships are maintained at version level. Hence, when a new service interface document is exposed, it is converted into this abstract internal representation. The features are extracted from the document, their respective descriptions and relationships are compared to the ones of the existing versions of the corresponding features, and new versions are created only when changes occurred. As a consequence, a new service is represented by interdependent features associated to existing or new versions, according to the changes. This finer-grain of service representation allows us to control the actual modified pieces of a service description.

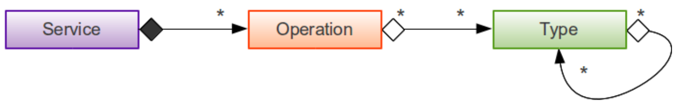


Fig. 1. Abstract service representation

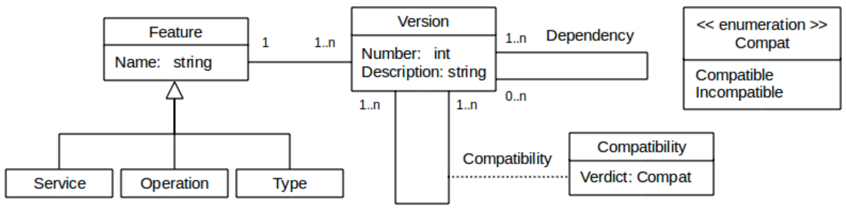


Fig. 2. Versioning Model at Feature Level

The versioning model is depicted in Figure 2 using a UML class diagram. A feature is generalization of service, operation and type. Each feature has at least one version, which in turn can depend on other versions of different features (for instance, an *operation* may depend on a set of *types*). Versions are uniquely identified by a pair <*Feature.Name*, *Version.Number*>, referenced throughout the remaining of this article as $v_{featureName,versionNumber}$. The *Version.Description* attribute corresponds to the textual description of the WSDL document, according to the feature type. The compatibility between two versions of a same feature is also maintained and assessed, as described in Section 4.

3.1 Feature Level Representation

The feature-oriented version model establishes the following correspondences to the WSDL/XSD service representation:

—*Operation*: related to the content of the tag <*operation*> within both <*portType*> and <*binding*> tags;

—*Type*: related to the content of tags <*element*>, <*complexType*> or <*simpleType*> within <*schema*> tag or the content of <*message*> tag. When addressing types, versioning is applied only to those defined outside the context of XSD complex elements, which means that only types meant for reuse are versioned. Hence, both primitive types (e.g. string, double, etc) and complex types, which cannot be referenced elsewhere, are not versioned.

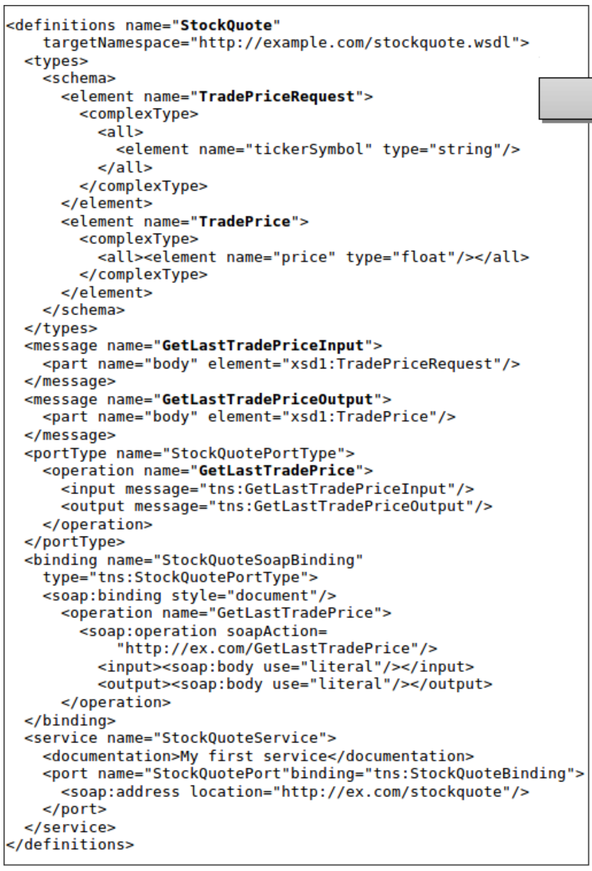
—*Service*: related to all the remaining content of the interface description document, such as the <*service*> tag and the remaining content of <*schema*>, <*portType*> and <*binding*> tags.

This mapping was developed considering WSDL 1.6, but it could be easily adapted to the more recent WSDL 2.0 with minor changes. Figure 3 illustrates the mapping of the fragments of a WSDL 1.6 description to the proposed representation, using the *StockQuote* service (Figure 3(i)). The description is separated into fragments to represent the service, its operation and types, resulting in a rooted graph containing the features versions (Figure 3(ii)). Each feature version description is associated with the corresponding WSDL fragment, as examplified in Figures 3(iii), 3(iv) and 3(v).

3.2 Versioning the features of an interface description document

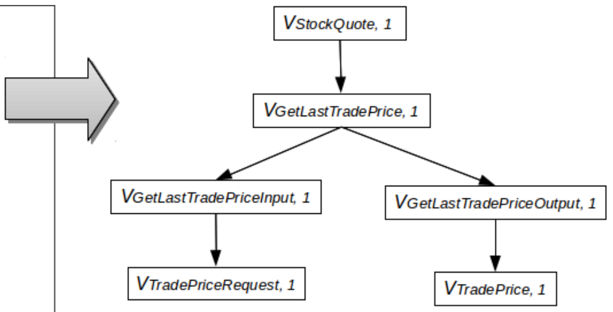
In order to version the features of a WSDL description document, it is necessary to identify the features within the document, relate them to the appropriate versions, possibly by creating new ones in this process, and store this abstract representation in a repository. The feature-oriented versioning process requires two steps: a) the conversion of the interface description document into the fine-grained perspective of features, and b) the analysis of each feature in order to discover whether it has changed with regard to existing versions in the repository.

The model assumes the creation of a new version for a feature only if it has been explicitly changed, or indirectly affected by the ripple effect of a change. A feature is *changed* if it has its description fragment changed somehow, depends on feature it did not previously depended on, or, conversely, no longer depends on a feature it previously depended on. A feature is *affected* when it was not explicitly changed, but it depends on a feature that has changed.

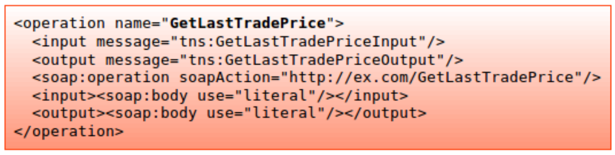


```
<definitions name="StockQuote"
    targetNamespace="http://example.com/stockquote.wsdl">
  <types>
    <schema>
      <element name="TradePriceRequest">
        <complexType>
          <all>
            <element name="tickerSymbol" type="string"/>
          </all>
        </complexType>
      </element>
      <element name="TradePrice">
        <complexType>
          <all><element name="price" type="float"/></all>
        </complexType>
      </element>
    </schema>
  </types>
  <message name="GetLastTradePriceInput">
    <part name="body" element="xsd1:TradePriceRequest"/>
  </message>
  <message name="GetLastTradePriceOutput">
    <part name="body" element="xsd1:TradePrice"/>
  </message>
  <portType name="StockQuotePortType">
    <operation name="GetLastTradePrice">
      <input message="tns:GetLastTradePriceInput"/>
      <output message="tns:GetLastTradePriceOutput"/>
    </operation>
  </portType>
  <binding name="StockQuoteSoapBinding"
    type="tns:StockQuotePortType">
    <soap:binding style="document"/>
      <operation name="GetLastTradePrice">
        <soap:operation soapAction=
          "http://ex.com/GetLastTradePrice"/>
        <input><soap:body use="literal"/></input>
        <output><soap:body use="literal"/></output>
      </operation>
  </binding>
  <service name="StockQuoteService">
    <documentation>My first service</documentation>
    <port name="StockQuotePort"binding="tns:StockQuoteBinding">
      <soap:address location="http://ex.com/stockquote"/>
    </port>
  </service>
</definitions>
```

(i) WSDL 1.1 representation for *StockQuote* service.

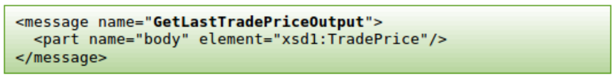


(ii) Graph of *StockQuote* version 1.



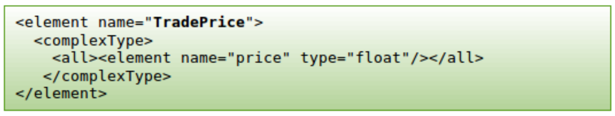
```
<operation name="GetLastTradePrice">
  <input message="tns:GetLastTradePriceInput"/>
  <output message="tns:GetLastTradePriceOutput"/>
  <soap:operation soapAction="http://ex.com/GetLastTradePrice"/>
  <input><soap:body use="literal"/></input>
  <output><soap:body use="literal"/></output>
</operation>
```

(iii) Description associated with *GetLastTradePrice* operation.



```
<message name="GetLastTradePriceOutput">
  <part name="body" element="xsd1:TradePrice"/>
</message>
```

(iv) Description associated with *GetLastTradePriceOutput* type.



```
<element name="TradePrice">
  <complexType>
    <all><element name="price" type="float"/></all>
  </complexType>
</element>
```

(v) Description associated with *TradePrice* type.

Fig. 3. Example of a WSDL 1.1 description and the proposed representation

This process is illustrated in Figure 3. First, the interface description document (e.g. Figure 3(i)) is converted into the feature level representation, which results in a graph representing the features' versions and their dependency relationships (e.g. Figure 3(ii)). Then, each feature is analyzed with regard to its corresponding feature in the repository, in order to compare to existing related versions. The analysis is done in a bottom-up manner regarding the graph of features in order to properly verify all dependencies changes. This analysis leads to four possibilities:

—If the feature does not exist, then it is created together with its first version.

—If the feature already exists (it was previously versioned) and its description differs from all existing versions of this particular feature, then it is marked as changed in the graph and a new version for this feature is created.

—If the feature already exists and its description is equal to an existing version:
  —If it depends on another feature that has been already marked as changed, then a new version is created due propagation effects.
  —If it does not depend on any changed feature, then every other feature that depends on this one is referenced to an already existing (equal) version.

To illustrate the idea of feature versioning, suppose a provider exposes the interface description for the first version of *StockQuote* service as depicted in Figure 3(i). The interface description is converted into the feature representation and versioned, such that each feature is associated with its first version (Figure 3(ii)). Suppose now that the provider exposes a new interface for this service that has two

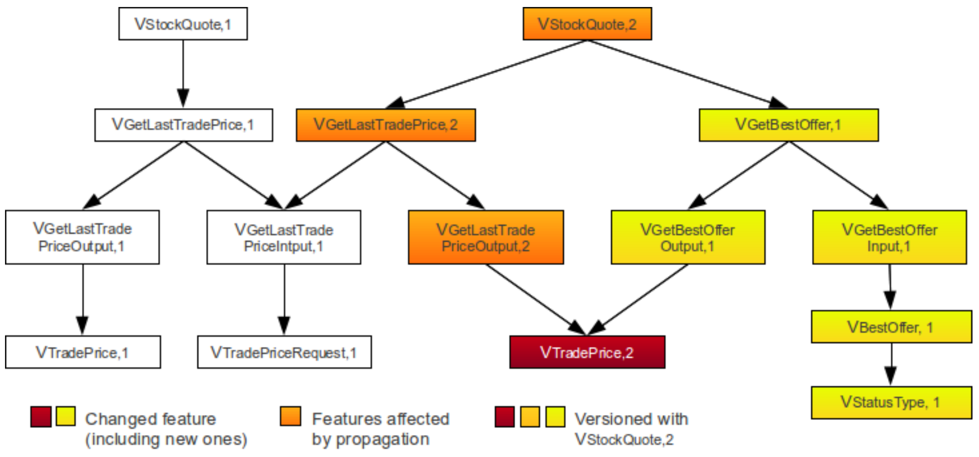


Fig. 4. Example of versioning StockQuote

major changes: a) a new operation with related types exchanged in messages, and b) changes the type of an existing feature. For the latter, suppose that the primitive type associated with *TradePrice* is changed from *float* (Figure 3(v)) to *double*. This new description is converted to the feature representation, and the change in *TradePrice* description is identified. So a new version is created, and associated with this feature. By propagation, features *GetLastTradePriceOutput*, *GetLastTradePrice* and *StockQuote* are affected, and hence equally versioned. In addition, features, together with the respective versions, are created for the new operation *GetBestOffer*, which in turn depends on the newly created features *GetBestOfferInput* and *GetBestOfferOutput*. These in turn depend on other features, which either previously existed (*TradePrice*) or need to be created (*BestOffer*, *StatusType*). The resulting version graph is depicted in Figure 4.

4. COMPATIBILITY ASSESSMENT ALGORITHM

The process described in the previous section transforms an existing standard WSDL/XSD service description into an internal representation, in which only the portions of the service explicitly changed or affected by the change are related to new versions. Otherwise, previously existing versions are associated to the features that constitute the service. In this way, any service description corresponds internally to a rooted graph of versions, where each version is related to a feature. The graph also defines the dependencies between features' versions (e.g. a service with regard to its operations, an operation with regard to the types that describe its messages). In addition to identifying which aspects of a service interface have changed, it is necessary to assess if each change is backward compatible with regard to previous versions.

The algorithm proposed in this article aims to assess compatibility between any two versions of a service, which implies examining recursively the compatibility of all the features that describe the service. The assessment of compatibility at feature level is an adaptation of the algorithm described in [Becker et al. 2011], which assesses compatibility on object-oriented service descriptions. We have adapted this algorithm to address the compatibility checking on smaller fragments of the WSDL/XSD interface description, as represented by our feature-oriented versioning model.

As mentioned in Section 2, most work considers a very restricted set of backward compatible change operations [Brown and Ellis 2004][Fang et al. 2007][Papazoglou et al. 2011][Brown and Ellis 2004][Fang et al. 2007]: a) addition of new operations, and b) addition of new types that are not contained within previously existing types. These are also the compatible changes considered in this work. Table I describes the cases addressed by the algorithm, in terms of change operations that can be applied over the versioning model. Any other change not mentioned in Table I is considered incompatible.

Table I. Change cases for version compatibility

Cases	*Change*	*Feature Type*	*Description*	*Compatibility Verdict*
1	Add	Operation	Add new operation to a service	*Compatible*
2	Add	Type	Add new type as dependency of a new operation/type	*Compatible*
3	Add	Type	Add new type as dependency of an existent operation/type	*Incompatible*
4	Update	Type	Change in description due to order, cardinality or type	*Incompatible*
5	Remove	Operation	Remove operation dependency	*Incompatible*
6	Remove	Type	Remove type dependency	*Incompatible*

The algorithm aims to recursively evaluate the compatibility relationship between two feature versions according to the rules of Table I, and to establish the compatibility relationship between them (Figure 2), with the corresponding verdict. The pseudo-algorithm is presented in Listing 1. It receives two feature versions as input, $v_{feature,p}$ and $v_{feature,q}$, and assess the compatibility of the latter with regard to the former. We assume that both versions relate to the same feature (i.e. have the same name). The algorithm verifies if dependencies of features present in $v_{feature,p}$ have not been removed from $v_{feature,q}$ (line 2), compares the description fragment associated with the compared versions (line 3), and then recursively evaluates the compatibility of all corresponding dependent feature versions (lines 4-11), when it finally sets the compatibility relationship and verdict (line 12). The version graph rooted $v_{feature,q}$, is traversed in a depth-first manner, which enables the propagation of detected incompatibilities to dependent versions.

Initially, the algorithm (line 2) evaluates whether feature dependencies were removed from $v_{feature,q}$ compared to $v_{feature,p}$ (cases 5 and 6 of Table I). Function *evaluateRemovedDependencies* verifies if all features in the set of dependencies of $v_{feature,p}$ still exist in the set of dependencies of $v_{feature,q}$. In line 3, cases 3 and 4 of Table I are verified, by analyzing the textual description associated with versions $v_{feature,q}$ and $v_{feature,p}$, as described in Listing 2.

Finally, the algorithm traverses all features upon which $v_{feature,q}$ depends in order to assess their compatibility against the corresponding ones in the dependency set of $v_{feature,p}$. For all dependencies of $v_{feature,q}$ (line 4), if there is a dependency in $v_{feature,p}$ with the same feature name and different version number (line 6), then the algorithm is called recursively to assess the compatibility of these two versions (line 8). If any dependency is incompatible, then the algorithm updates the verdict to incompatible due to the ripple effect. If there is a dependent version of $v_{feature,q}$ that does not exist in *setOfDependencies*($v_{feature,p}$), then this situation is related to the compatible cases 1 and 2 of Table I. Finally, the algorithm returns the compatibility assessment of $v_{feature,q}$ regarding $v_{feature,p}$. Notice that the algorithm could stop at any point where incompatibility is detected if the goal were to detect the compatibility of the whole service. We continue the assessment to assess each feature individually.

Function *evaluateDescription*, detailed in Listing 2, currently evaluates true if: a) descriptions are exactly the same (line 2); and b) in the case of *Type* features, interpret the XML for the rules of

Listing 1 : compatibilityAssessment($v_{feature,p}$, $v_{feature,q}$)

```
 1: boolean $compat \leftarrow true$;
 2: $compat \leftarrow evaluateRemovedDependencies(v_{feature,p}, v_{feature,q})$;
 3: $compat \leftarrow compat \land evaluateDescription(v_{feature,p}, v_{feature,q})$;
 4: foreach $v_{dep,Qj} \in setOfDependencies(v_{feature,q})$ do
 5:     if exists $v_{dep,Pi} \in setOfDependencies(v_{feature,p}) \land (depP = depQ) \land (i \neq j)$ then
 6:         $compat \leftarrow compat \land compatibilityAssessment(v_{depP,i}, v_{depQ,j})$;
 7:     end if
 8: end for
 9: $setVerdict(v_{feature,q}, v_{feature,p}, compat)$;
10: return $compat$;
```

Listing 2 : evaluateDescription($v_{feature,p}$,$v_{feature,q}$)

```
1: boolean $compat \leftarrow true$;
2: if $v_{feature,p}(description) \neq v_{feature,q}(description)$ then
3:     if $v_{feature,p}(Feature.Type) = type$ then
4:         foreach $e_j \in setOfElements(v_{feature,q})$ do
5:             if not exists $e_i \in setOfElements(v_{feature,p}) \wedge e_i(name) = e_j(name)$ then
6:                 $compat \leftarrow false$;
7:             else if $(e_i(order) \neq e_j(order) \vee e_i(type) \neq e_j(type) \vee e_i(cardinality) \neq e_j(cardinality))$
   then
8:                 $compat \leftarrow false$;
9:             end if
10:        end for
11:        foreach $e_i \in setOfElements(v_{feature,p})$ do
12:            if not exists $e_j \in setOfElements(v_{feature,q}) \wedge e_i(name) = e_j(name)$ then
13:                $compat \leftarrow false$;
14:            end if
15:        end for
16:    else
17:        $compat \leftarrow false$;
18:    end if
19: end if
20: return $compat$;
```

case 4 of Table I (lines 3-18). It relies on function *setOfElements*, which parses the excerpt of WSDL corresponding to the description field in order to extract the description elements and their properties for examining this case. In the future we plan to adopt less restrictive compatibility rules (e.g. [Becker et al. 2008][Andrikopoulos et al. 2012]).

5. EXPERIMENTS

To develop experiments with the versioning model and the compatibility assessment algorithm, we built a prototype that follows the architecture displayed at Figure 5. The *WSDL/Feature Converter* module transforms a new WSDL description into the internal versioning model as described in Section 3.2, and the *Compatibility Analyzer* module implements the compatibility assessment algorithm detailed in Section 4. The *Version Evolution Repository* persists the resulting features and versions in an XML file, according to the versioning model described in Figure 2. The prototype was developed using Java, DOM Parser and JGraphT library for handling the versions graph.

The eBay *Trading* service [3] was selected for the experiments. eBay introduces a new version of this service every two weeks and supports each version for at least 18 months. For each version, there is a release notes entry on eBay website that reports the explicit points of change with regard to the previous version. However, there is no information on how these changes affect other parts of the service. The manual analysis of propagation effects is a hard task since the interface document is huge (e.g. the most recent version of Trading service has almost 130.000 lines). Thus, client developers are responsible for detecting whether changes affect their applications. The release notes also report wversions are backward compatible with regard to the previous one.

We have developed two experiments examining 40 versions of *Trading* service, using version 653 as baseline. The first experiment compares the advertised changes in respective the release notes with the actual changes detected by the *WSDL/Feature Converter* module. The second experiment assesses

[3]`https://www.x.com/developers/ebay/products/trading-api`

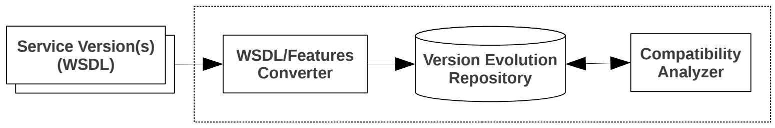


Fig. 5. Prototype Architecture

the compatibility of each feature changed with regard to the corresponding ones in the immediately previous version. We were not able to compare our approach with existing work that automatically detects changes and assess compatibility due to the following reasons: a) the algorithm we extended [Becker et al. 2011] assumes an object-oriented description of service, thus incompatible with WDSL descriptions; b) results reported in [Fokaefs et al. 2011] do not allow comparison, because these results are limited to the presentation of percentages of changes categorized as inclusion, deletion and update; c) [Zou et al. 2008] [Ponnekanti and Fox 2004] focus on detecting changes between a specific client and a service, and do not report results that compare the changes on service versions and their compatibility; and d) code is unavailable for reproducing the results of all aforementioned works.

5.1 Experiment 1 : Quantifying Changes

The goal of this experiment is to compare the updates reported in the provider's release notes and the ones detected by our prototype. For this purpose, we quantified all the explicit and cascaded changes of each new service version with regard to the previous one by counting the newly created versions, and classifying them as changed and affected. The results are displayed in Figure 6. eBay identifies versions using odd numbers, and because version 653 is the baseline, the graph starts at version 655.

We verified that all changed features made explicit due to our versioning model are described in the release notes, but none of the affected ones are mentioned. Explicitly changed features correspond in total to less than 5% of the overall detected changes, which means that more than 95% of changes are not addressed in the release notes. For instance, version 659 has introduced a single explicit change on a type feature, which is reported in the release notes [4]. However, this change affects 36 operations and 100 types that depend on it directly or indirectly, and which are not covered by the release notes.

We observed that 99% of explicit changes are done to types, whereas the ripple effect reaches an average of 26% for operations and 74% for types. Hence, typically a change is done to a type, and in most cases it does not directly affect an operation. Instead, its effects cascade through several types until it affects an operation. This experiment confirms that current information provided by release notes is insufficient for client developers to detect which changes affect them. We conclude that the proposed versioning model supported the efficient identification and quantification of changes impact.

[4] `http://developer.ebay.com/Devzone/XML/docs/WebHelp/ReleaseNotesArchive.html#659`

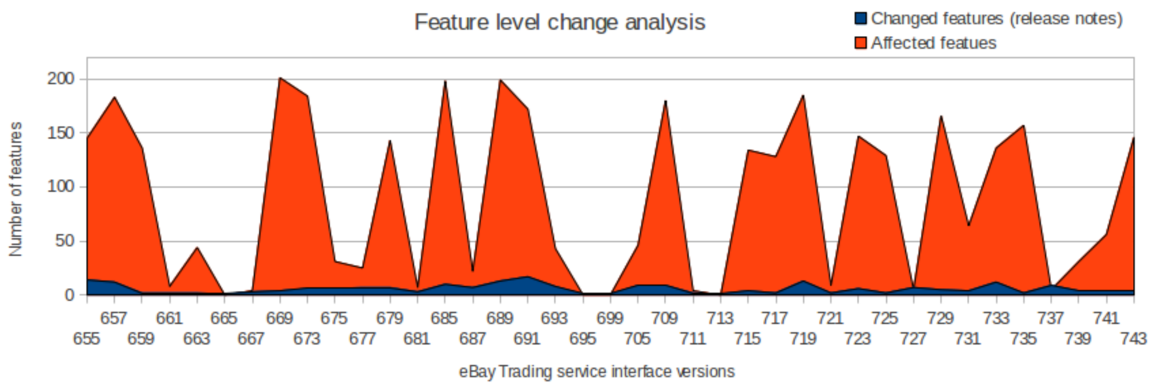


Fig. 6. Change analysis of eBay Trading service interface versions at feature level

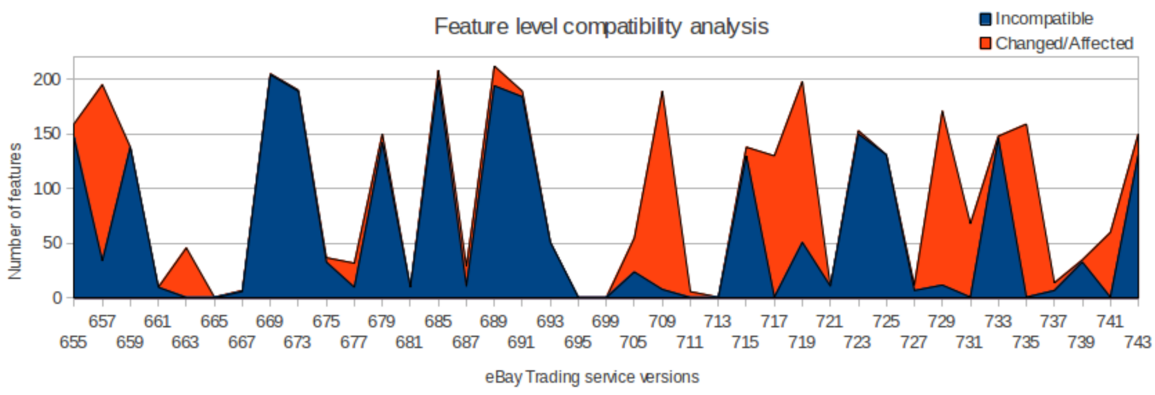


Fig. 7. Trading Service Compatibility Analysis

5.2 Experiment 2 : Qualifying Changes

In this experiment, we analyzed at feature level the compatibility of each service version with the preceding one using our compatibility assessment algorithm. The results are presented in Figure 7. According to the compatibility rules we adopt (Table I), none of the service versions are backward compatible with the previous one. This result differs from the information available in the release notes for these versions, which report only 3 out of the 40 as backward incompatible, namely versions 689, 717 and 719. This discrepancy is explained by eBay evolution policy, which requires that developers build client applications that handle unrecognized data (e.g. non expected additional output arguments). In other words, it assumes as compatible the case 3 of Table I, which is reported as incompatible in related literature [Brown and Ellis 2004] [Fang et al. 2007] [Papazoglou et al. 2011] [Becker et al. 2008]. Hence, clients that do not comply with their evolution policy may be affected in every change, depending whether or not they use the incompatible changed features.

This experiment indicates the usefulness of automatic compatibility assessment algorithms, consistent with compatibility rules adopted by providers. In such a way, providers can easily understand the effects of their intended changes and even of the compatibility rules assumed.

6. CONCLUSION

We presented a versioning model and compatibility assessment algorithm for supporting evolution on a finer grain than the typical service description WSDL/XSD documents. The approach enables to easily identify which are the changed (or affected) features in a new service version, and whether these features were changed in a way client applications are not broken. Hence, we address the identification of impact points, propagation effects, and the qualification of changes. We have experimented our approach in a real service, and were able to demonstrate its usefulness for pointing out the changes performed in a service description and their compatibility, particularly if compared to the release notes. The approach also provides important information for supporting service evolution by either maximizing version reusability (e.g redirecting of requests, overload balancing, etc.) and/or pinpointing the change impact points that basis usage based approaches [Frank et al. 2008][Treiber et al. 2009][Zou et al. 2008][Yamashita et al. 2012].

Future work will integrate the versioning model and compatibility assessment algorithm into the framework designed for usage profiles analysis presented in [Yamashita et al. 2012][Silva et al. 2012].

REFERENCES

Andrikopoulos, V., Benbernou, S., and Papazoglou, M. P. Managing the evolution of service specifications. In *Proceedings of the International Conference on Advanced Information Systems Engineering*. Montpellier, France, pp. 359–374, 2008.

Andrikopoulos, V., Benbernou, S., and Papazoglou, M. P. On the evolution of services. *IEEE Transactions on Software Engineering* 38 (3): 609 – 628, 2012.

Becker, K., Lopes, A., Milojicic, D. S., Pruyne, J., and Singhal, S. Automatically determining compatibility of evolving services. In *Proceedings of the IEEE International Conference on Web Services*. Beijing, China, pp. 161–168, 2008.

Becker, K., Pruyne, J., Singhal, S., Lopes, A., and Milojicic, D. S. Automatic determination of compatibility in evolving services. *International Journal of Web Service Research* 8 (1): 21–40, 2011.

Brahmia, Z., Bouaziz, R., Grandi, F., and Oliboni, B. Schema versioning in txschema-based multitemporal xml repositories. In *Proceedings of the Fifth IEEE International Conference on Research Challenges in Information Science*. Guadeloupe, France, pp. 1–12, 2011.

Brown, K. and Ellis, M. Best Practices for Web services Versioning. Website, 2011. http://www.ibm.com/developerworks/webservices/library/ws-version/, 2004.

Cavalieri, F., Guerrini, G., and Mesiti, M. Updating xml schemas and associated documents through exup. In *Proceedings of the IEEE International Conference on Data Engineering*. Hannover, Germany, pp. 1320–1323, 2011.

Fang, R., Lam, L., Fong, L., Frank, D., Vignola, C., Chen, Y., and Du, N. A version-aware approach for web service directory. In *Proceedings of the IEEE International Conference on Web Services*. Salt Lake City, USA, pp. 406–413, 2007.

Fokaefs, M., Mikhaiel, R., Tsantalis, N., Stroulia, E., and Lau, A. An empirical study on web service evolution. In *Proceedings of the IEEE International Conference on Web Services*. Washington - DC, USA, pp. 49–56, 2011.

Frank, D., Lam, L., Fong, L., Fang, R., and Khangaonkar, M. Using an interface proxy to host versioned web services. In *Proceedings of the IEEE International Conference on Services Computing*. Honolulu, Hawaii, pp. 325–332, 2008.

Katz, R. H. Towards a unified framework for version modeling in engineering databases. *ACM Computing Surveys* 22 (4): 375–408, 1990.

Lee, K., Kang, K. C., and Lee, J. Concepts and guidelines of feature modeling for product line software engineering. In *Software Reuse: Methods, Techniques, and Tools*, C. Gacek (Ed.). Lecture Notes in Computer Science, vol. 2319. Springer Berlin / Heidelberg, pp. 62–77, 2002.

Leitner, P., Michlmayr, A., Rosenberg, F., and Dustdar, S. End-to-end versioning support for web services. In *Proceedings of the IEEE International Conference on Services Computing*. Honolulu, Hawaii, pp. 59–66, 2008.

Mozafari, B., Zeng, K., and Zaniolo, C. K*sql: a unifying engine for sequence patterns and xml. In *Proceedings of the ACM SIGMOD International Conference on Management of Data Conference*. Indianapolis, USA, pp. 1143–1146, 2010.

Papazoglou, M. P., Andrikopoulos, V., and Benbernou, S. Managing evolving services. *IEEE Software* 28 (3): 49–55, 2011.

Ponnekanti, S. and Fox, A. Interoperability among independently evolving web services. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference*. Toronto, Canada, pp. 331–351, 2004.

Silva, E., Vollino, B., Becker, K., and Galante, R. A business intelligence approach to support decision making in service evolution management. In *Proceedings of the IEEE International Conference on Services Computing*. Honolulu, Hawaii, pp. 41–48, 2012.

Treiber, M., Andrikopoulos, V., and Dustdar, S. Calculating service fitness in service networks. In *Proceedings of the ICSOC/ServiceWave 2009 Workshops*. Stockholm, Sweden, pp. 283–292, 2009.

Wang, S. and Capretz, M. A. M. A dependency impact analysis model for web services evolution. In *Proceedings of the IEEE International Conference on Web Services*. Los Angeles, CA, USA, pp. 359–365, 2009.

Westfechtel, B., Munch, B. P., and Conradi, R. A layered architecture for uniform version management. *IEEE Transactions on Software Engineering* 27 (12): 1111–1133, 2001.

Yamashita, M., Becker, K., and Galante, R. Service evolution management based on usage profile. In *Proceedings of the IEEE International Conference on Web Services*. Washington - DC, USA, pp. 746–747, 2011.

Yamashita, M., Vollino, B., Becker, K., and Galante, R. Measuring change impact based on usage profiles. In *Proceedings of the IEEE International Conference on Web Services*. Honolulu, Hawaii, pp. 226 – 233, 2012.

Zou, Z., Fang, R., Liu, L., Wang, Q., and Wang, H. On synchronizing with web service evolution. In *Proceedings of the IEEE International Conference on Web Services*. Beijing, China, pp. 329–336, 2008.