A Flexible Approach for Assessing Service Compatibility at Feature Level

Marcelo Yamashita, Karin Becker, Renata Galante

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

Abstract. 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 paper 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 paper also reports an experiment based on a real service, which explores the versioning model to assess the scope of implicit and explicit changes.

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

Keywords: Service compatibility, service evolution management, service versioning

1. INTRODUCTION

Services are subject to constant changes and variations, 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 2008]. 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 [Frank et al. 2008].

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. 2011]. Typically, despite many service features remaining unchanged (e.g. types, operations, message calls), the whole description document is versioned. This 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 documents that hopefully will help clients to adjust to changes (e.g. eBay, Google, Amazon). Release notes usually describe the direct changes (e.g. changes on schema types of service calls), but fail to properly identify how changes propagate their effect throughout the entire service [Le 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 can also be affected by the change. 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 [Le 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 the changed (or affected) 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 [Le 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. 2011].

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. Works such as [Yamashita et al. 2011][Ponnekanti and Fox 2004][Andrikopoulos et al. 2008] suggest the benefits of assessing compatibility in terms of specific client usage.

The objective of this work is to detect explicit and implicit changes between service versions in order to pinpoint incompatible changes. The contribution of this paper 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.

The approach discussed in this paper 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 change impact (e.g. [Wang and Capretz 2009]). It could also support the automatic creation of more detailed release notes based on usage analysis, as in [Le Zou et al. 2008]. Compatibility assessment at feature level enables service evolution based on usage profiles [Yamashita et al. 2011], reduction of provisioned versions based on proxy redirections [Frank et al. 2008], and load balance management among implemented versions, which precedes the finer grain deployment in [Treiber et al. 2010].

The remaining of this paper is structured as follows. Section 2 describes the feature-oriented versioning model, and Section 3 presents the corresponding compatibility assessment algorithm. Section 4 reports an experiment that explores the feature-oriented versioning model to quantify the impact of changes in a real case scenario. Section 5 describes related work. Conclusions and future work are addressed in Section 6.

2. FEATURE-ORIENTED VERSIONING MODEL

This model suits a number of different applications, such as [Frank et al. 2008] [Le Zou et al. 2008] [Frank et al. 2008] [Wang and Capretz 2009] [Treiber et al. 2010] [Yamashita et al. 2011]. Nonetheless, in this paper we focus on its contribution for measuring the scope of change and qualifying its compatibility on a finer grain manner. For this purpose, we characterize a service as a composition of operations that can be accessed by client applications. These operations are bound to a particular format (order and type of input/output parameters), which is usually defined by schema

Fig. 1. Abstract service representation

elements (referred to as *types*), which in turn may depend on other schema elements. We refer to a service and each of its operations and types as a service *feature*. The abstract representation of features and their relationship is described in Figure 1.

Each feature corresponds to a portion of a WSDL/XSD service description document. We propose to version features separately, and to maintain their dependency relationships at version level. The idea of feature versioning is to provide an abstract management to different parts of the interface description in order to version only the changed features, rather than the entire service. Hence, when a new service interface document is exposed, we convert it to an abstract internal representation, compare the descriptions of the features with regard to previously existing ones, as well as their relationship with other features, and create new versions only when changes occurred. As a consequence, a new service is represented by interdependent features associated to new versions, or previously existing ones, according to the changes. This finer-grain of service representation allows us to control the actual modified pieces of a service description, and version only the modified ones.

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 paper 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 more details in Section 3.

2.1 Feature Level Representation

The feature-oriented version model has 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 can not be referenced elsewhere, are not versioned.

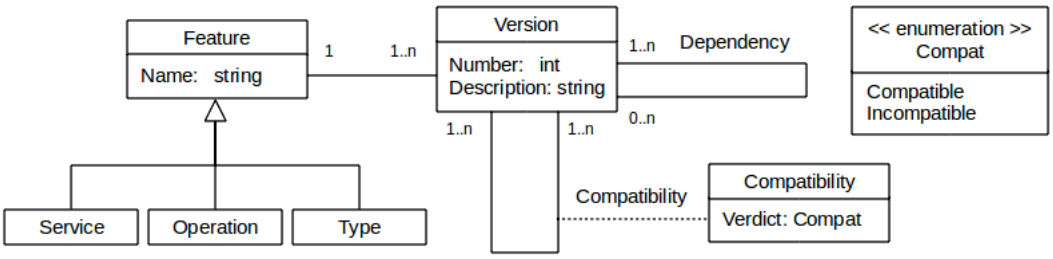
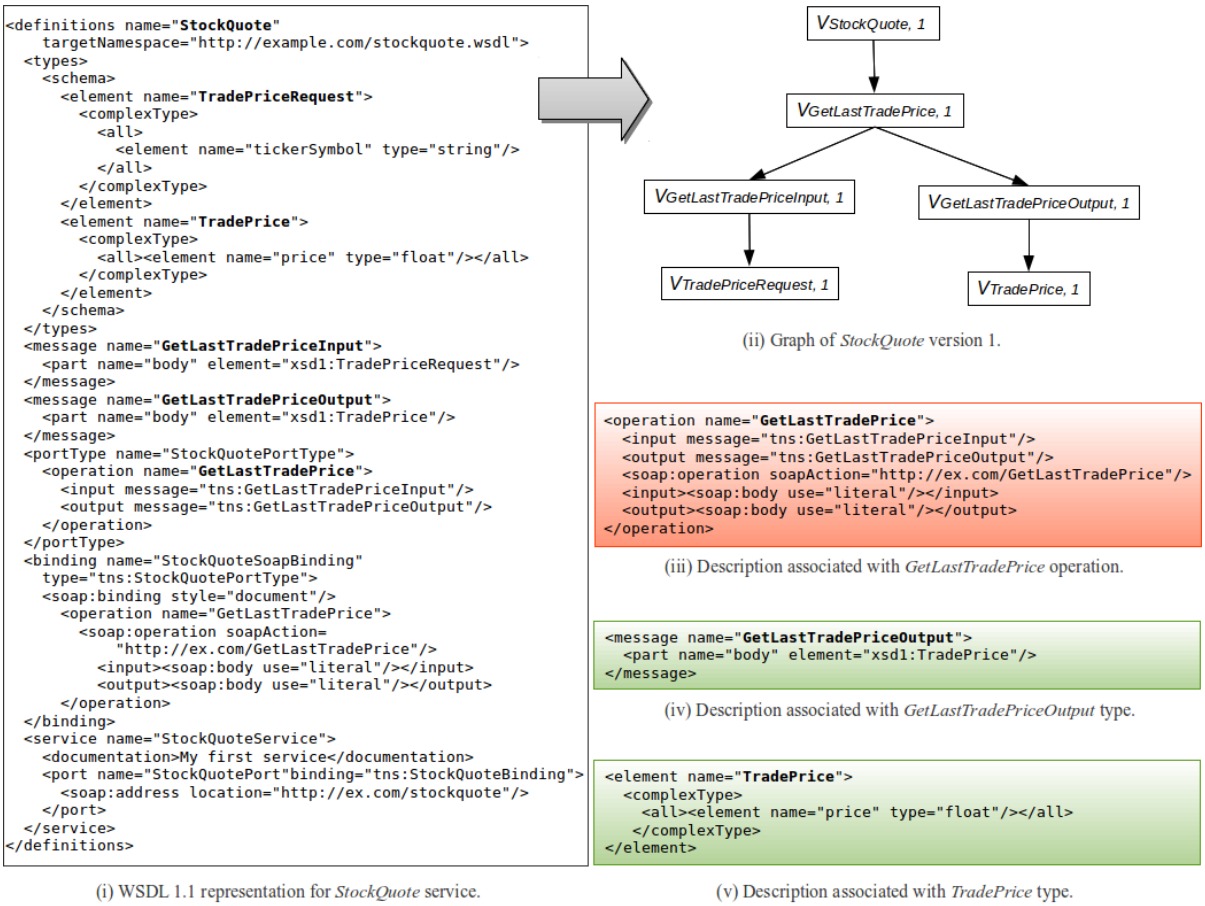


Fig. 2. Versioning Model at Feature Level



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

(ii) Graph of *StockQuote* version 1.

(iii) Description associated with *GetLastTradePrice* operation.

(iv) Description associated with *GetLastTradePriceOutput* type.

(v) Description associated with *TradePrice* type.

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

—*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.x, but it could be easily adapted to the more recent WSDL 2.0 with smaller changes. Figure 3 illustrates the mapping of the fragments of a WSDL 1.6 description to the proposed representation, using the *StockQuote* service[1]. We separate the description of Figure 3(i) into fragments to represent the service, its operation and types. The result of this fragmentation is a rooted graph containing the features versions (Figure 3(ii)). Each feature version is associated with a corresponding description, such the examples in Figures 3(iii), 3(iv) and 3(v).

2.2 Versioning the features of an interface description document

In order to version the features of a WSDL description document, we need to identify the features within the document, relate them to the appropriate versions, possibly by creating new versions in this process, and store this abstract representation in a repository. The feature-oriented versioning requires two steps: a) the conversion of the interface description document to the finer granulated perspective of features, and b) the analysis of the features in order to discover whether they have changed regarding all their previous representation in the repository.

[1] `http://www.w3.org/TR/wsdl#_wsdl`

The model considers to version only explicitly changed features, or features that are indirectly affected by the changes. A feature is *changed* if either 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 did not explicitly changed, but depends on a feature that has changed.

First, the interface description document (e.g. Figure 3(i)) is converted to the feature level representation, which results in a graph representing the features' versions and their dependency relationships (e.g. Figure 3(ii)). Then, all features are analyzed with regard to their correspondent in the repository in order to compare to existent versions. The analysis is done in a bottom-up manner regarding the graph of features in order to properly verify dependencies changes. The 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). We convert the interface description to the feature level representation and version them, 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 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

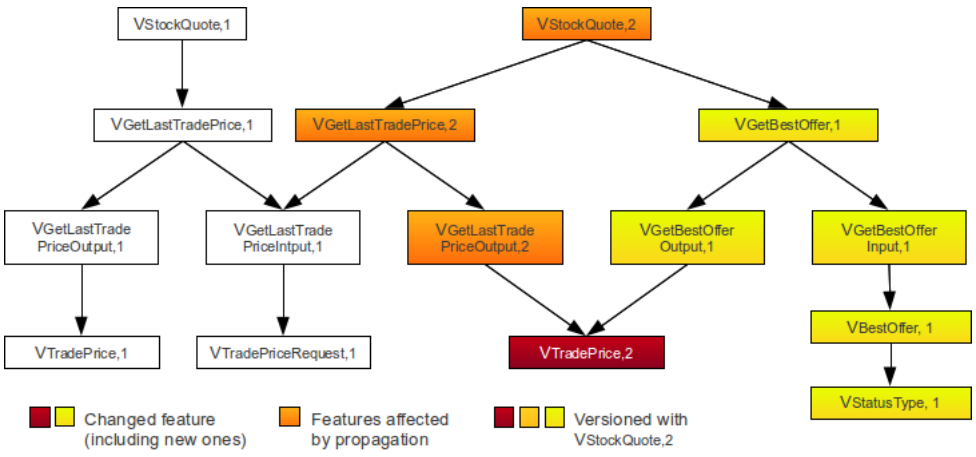


Fig. 4. Example of versioning StockQuote

features, which either previously existed (*TradePrice*) or need to be created (*BestOffer*, *StatusType*). The resulting version graph is depicted in Figure 4.

3. COMPATIBILITY ASSESSMENT ALGORITHM

The algorithm proposed in this paper aims to assess compatibility between any two versions of a service, which implies examining the compatibility considering all the features that describe the service. The assessment of compatibility at feature level relies on the compatibility detection algorithm proposed in [Becker et al. 2008], which assesses compatibility on object-oriented service descriptions. We have adapted this algorithm to address the compatibility on smaller fragments of the WSDL interface description, as represented by our feature-oriented versioning model.

Very few changes are backward-compatible, basically, the addition of new operations, and types that are not referenced by existing operations and types ([Andrikopoulos et al. 2011][Becker et al. 2008][Fang et al. 2007][Brown and Ellis 2004]). For this paper we consider the change cases depicted in Table I, which translate the core cases found in the literature into change operations in our versioning model.

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, with the corresponding verdict (Figure 2). 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 later 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), compare the description fragment associated with the compared versions (line 3), and then recursively evaluates the compatibility of all corresponding dependent feature versions (lines 4-9), when it finally sets the compatibility relationships and verdict (line 10). The version graph rooted $v_{feature,q}$, is traversed in a depth-first manner, which enables the propagation of detected incompatibilities to dependent versions.

The first step of the algorithm (line 2) aims to evaluate whether feature dependencies were removed from $v_{feature,q}$ compared to $v_{feature,p}$. The function *evaluateRemovedDependencies* verifies if all features in the set of dependencies of $v_{feature,p}$ still exist in the set of dependencies of $v_{feature,q}$. If it detects a removed dependency, then versions $v_{feature,p}$ and $v_{feature,q}$ are incompatible due to cases 5 and 6 of Table I. The algorithm also evaluates the textual description associated with versions $v_{feature,q}$ and $v_{feature,p}$. The description evaluation is detailed in Listing 2.

Then, 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 5), then the algorithm is called recursively to assess the compatibility of these two versions (line 6). If any dependency is incompatible, then the algorithm updates the verdict to incompatible due to propagation effects. 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 $v_{feature,p}$.

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*

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

```
1  boolean compat ← true;
2  compat ← evaluateRemovedDependencies(v_feature,p , v_feature,q);
3  compat ← compat ∧ evaluateDescription(v_feature,p , v_feature,q );
4  foreach v_depQ,j ∈ setOfDependencies(v_feature,q) do
     // If there is a dependency feature version with the same name but different version
5    if exists v_depP,i ∈ setOfDependencies(v_feature,p) ∧ (depP = depQ) ∧ (i ≠ j) then
       // Verify recursively if these two versions of a same feature are in turn compatible
6      compat ← compat ∧ compatibilityAssessment(v_depP,i, v_depQ,j)
7    end if
8  end foreach
9  setVerdict(v_feature,q, v_feature,p , compat);
10 return compat;
```

Notice that the algorithm could stop at any point where incompatibility is detected, but we have opted by continuing the assessment as a placeholder for later documenting all incompatible changes found.

Function *evaluateDescription* (line 3 in Listing 1), detailed in Listing 2, aims at verifying the remaining incompatible cases of Table I, namely cases 3 and 4. It receives as input the versions of a same feature $v_{feature,p}$ and $v_{feature,q}$ and check if their descriptions are different (line 2), assuming incompatibility, except in the case of type features, when a more detailed examination of the description field is performed (lines 3-16). In the future we plan for studying less restrictive compatibility. For instance, adding a new optional type at the end of a complex type sequence can be compatible if it is an input message [Becker et al. 2008][Andrikopoulos et al. 2011].

In order to extract the description elements and their properties from the WSDL fragments we use the function *setOfElements*, which parses the excerpt of WSDL corresponding to the description field. The algorithm verifies if there is any added element in $v_{feature,q}$ regarding $v_{feature,p}$ (line 5), which

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

```
1  boolean compat ← true;
2  if v_feature,p(description) ≠ v_feature,q(description) then
3    if v_feature,p(Feature.Type) = type then
4      foreach e_j ∈ setOfElements(v_feature,q) do
5        if not exists e_i ∈ setOfElements(v_feature,p) ∧ e_i(name) = e_j(name) then
6          compat ← false;
7        else if (e_i(order) ≠ e_j(order) ∨ e_i(type) ≠ e_j(type) ∨ e_i(cardinality) ≠ e_j(cardinality)) then
8            compat ← false;
9        end if
10     end foreach
11     foreach e_i ∈ setOfElements(v_feature,p) do
12       if not exists e_j ∈ setOfElements(v_feature,q) ∧ e_i(name) = e_j(name) then
13         compat ← false;
14       end if
15     end foreach
16   else
17     compat ← false;
18   end if
19 end if
20 return compat;
```

leads to the verification of case 3 of Table I. Next, if the element exists within the two versions we compare their properties (line 7) in order to verify if they have changed. The comparison of element properties refers to case 4 of Table I. Removed description elements are verified in lines 11 to 15. Finally, it returns the compatibility verdict for $v_{feature,q}$ regarding $v_{feature,p}$.

4. EXPERIMENTS

We have implemented a prototype for versioning services at feature level, which identifies explicit changed features and affected ones in order to version them. The prototype also generates a report listing the changed and affected features. Our goal is to compare this report with the provider's release notes in order to understand the actual lack of information clients' developers deal with when adapting their applications.

For this experiment we chose eBay *Trading* service [2]. 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.

We used for this experiment the 41 versions of *Trading* service currently supported by eBay. As our goal was to analyze structural changes, we removed all semantic information (e.g. documentation tags) from the service interface documents. We used the oldest available version (653) as baseline, and quantified the explicit and cascaded changes of each new service version with regard to the previous one by counting the newly created versions. Recall that each new version corresponds to a feature that was changed or affected. The results are displayed in Figure 5.

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 [3]. 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 also verified that 99% of explicitly changed features are done to types, whereas the propagation effects reach an average of 26% for operations and 74% for types. This indicates that a change is typically done to a type, and in most cases it is not directly propagated to 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, and in which way, in order to rapidly being able adequate their applications to evolutionary changes of services. We conclude that the proposed versioning model supported the efficient identification and quantification of changes impact. Moreover, the further implementation of the compatibility assessment algorithm can support a more efficient analysis of change impact by qualifying changes and their effect in each feature.

[2] `https://www.x.com/developers/ebay/products/trading-api`
[3] `http://developer.ebay.com/Devzone/XML/docs/WebHelp/ReleaseNotesArchive.html#659`

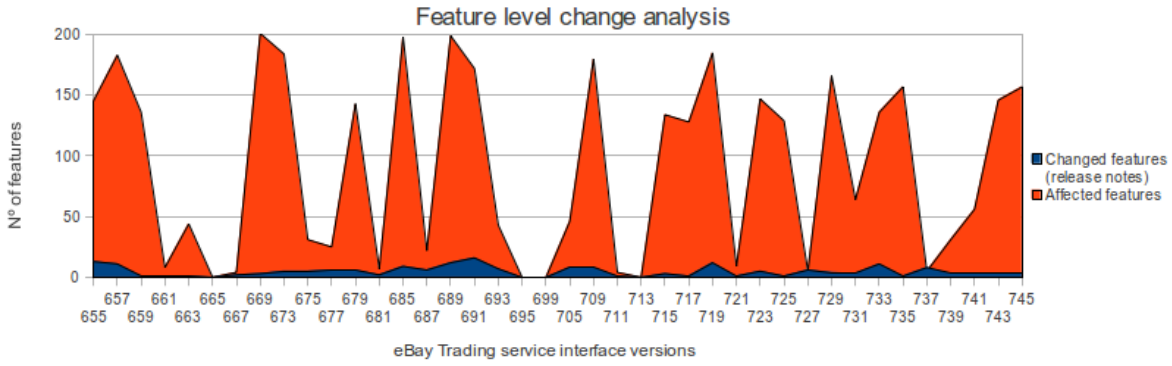


Fig. 5. Change analysis of eBay Trading service interface versions at feature level

5. RELATED WORK

Current approaches on service evolution address service versioning and service compatibility, in order to achieve some degree of conformance and/or transparency during evolution. 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. 2011]. 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 between versions [Le Zou et al. 2008][Fokaefs et al. 2011], particularly regarding compatibility. [Becker et al. 2008] proposes a finer-grain versioning model based on the object-oriented paradigm, but which does not fit W3C current standards. [Andrikopoulos et al. 2011] proposes an abstract model for the description of services, which details all its components, but it is targeted at verifying formally the compatibility of service interface versions, and which does not address versioning itself.

Different works express the need for the easy change impact. [Le Zou et al. 2008] proposes a service delta analyzer to detect explicit changes between service versions, in order to produce customized release notes. [Wang and Capretz 2009] proposes a dependency model to detect and quantify explicit and implicit changes in order to measure change impact, but does not enable the assessment of compatibility. [Fokaefs et al. 2011] presents an alternative method for change detection based on edition distance and clustering. However, current approaches do not address the assessment of compatibility in a finer-grain.

Works such as [Brown and Ellis 2004][Fang et al. 2007][Becker et al. 2008][Andrikopoulos et al. 2011] discuss compatibility rules, i.e. change operations that do not break client application. [Becker et al. 2008] addresses the automatic assessment of compatibility but it targets on classes that represent service descriptions. [Ponnekanti and Fox 2004] addresses compatibility on a finer-grain but restricted to the perspective of a particular client application usage. [Yamashita et al. 2011] suggests the benefits of compatibility assessment on a finer-grain by considering compatibility against specific usage profiles.

The work described in this paper contributes to the above work by the versioning of services in a finer-grain manner, i.e. at feature level. In doing so, we can identify explicit changes, analyze their propagation effects within the feature level proposal and assess compatibility on specific feature versions. Consequently, we can both quantify the scope of changes and qualify their impact. The result of this work supports various applications, such as the service evolution based on usage profiles [Yamashita et al. 2011], reduction of provisioned versions based on proxy redirection [Frank et al. 2008], and enables load balance management among implemented versions, which precedes the finer-grain deployment in [Treiber et al. 2010].

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 feature-oriented versioning model using a real service, and were able to demonstrate its usefulness for pointing out the changes performed in a service description, 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. 2010][Le Zou et al. 2008][Yamashita et al. 2011].

We have prototyped an implementation for versioning service descriptions according to our model, and currently we are concluding the implementation of the compatibility assessment algorithm. Future work will integrate the versioning model and compatibility assessment algorithm into the framework designed for usage profiles analysis presented in [Yamashita et al. 2011].

REFERENCES

Andrikopoulos, V., Benbernou, S., and Papazoglou, M. On the evolution of services. *Software Engineering, IEEE Transactions on* (99): 1–1, 2011.

Andrikopoulos, V., Benbernou, S., and Papazoglou, M. P. Managing the evolution of service specifications. In *Proceedings of the 20th international conference on Advanced Information Systems Engineering*. CAiSE '08. Springer-Verlag, Berlin, Heidelberg, pp. 359–374, 2008.

Becker, K., Lopes, A., Milojicic, D., Pruyne, J., and Singhal, S. Automatically determining compatibility of evolving services. In *Web Services, 2008. ICWS'08. IEEE International Conference on*. IEEE, pp. 161–168, 2008.

Brown, K. and Ellis, M. Best Practices for Web services Versioning, 2004.

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

Fokaefs, M., Mikhaiel, R., Tsantalis, N., Stroulia, E., and Lau, A. An empirical study on web service evolution. In *Web Services (ICWS), 2011 IEEE International Conference on*. IEEE, 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 *Services Computing, 2008. SCC'08. IEEE International Conference on*. Vol. 2. IEEE, pp. 325–332, 2008.

Le Zou, Z., Fang, R., Liu, L., Wang, Q., and Wang, H. On synchronizing with web service evolution. In *Web Services, 2008. ICWS'08. IEEE International Conference on*. IEEE, pp. 329–336, 2008.

Papazoglou, M. The challenges of service evolution. In *Advanced Information Systems Engineering*. Springer, pp. 1–15, 2008.

Ponnekanti, S. and Fox, A. Interoperability among independently evolving web services. *Middleware 2004*, 2004.

Treiber, M., Andrikopoulos, V., and Dustdar, S. Calculating service fitness in service networks. In *Service-Oriented Computing. ICSOC/ServiceWave 2009 Workshops*. Springer, pp. 283–292, 2010.

Wang, S. and Capretz, M. A dependency impact analysis model for web services evolution. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*. IEEE, pp. 359–365, 2009.

Yamashita, M., Becker, K., and Galante, R. Service evolution management based on usage profile. In *Web Services (ICWS), 2011 IEEE International Conference on*. IEEE, pp. 746–747, 2011.