UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

MARCELO CORRÊA YAMASHITA

Service Versioning and Compatibility at Feature Level

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Profa. Dra. Renata Galante
Advisor

Profa. Dra. Karin Becker
Co-advisor

Porto Alegre, June 2013

CIP – CATALOGING-IN-PUBLICATION

Yamashita, Marcelo Corrêa

Service Versioning and Compatibility at Feature Level / Marcelo Corrêa Yamashita. – Porto Alegre: PPGC da UFRGS, 2013.

72 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2013. Advisor: Renata Galante; Co-advisor: Karin Becker.

1. Service versioning. 2. Service compatibility. 3. Service evolution management. I. Galante, Renata. II. Becker, Karin. III. Título.

“Creativity is intelligence having fun.”
— ALBERT EINSTEIN

AGRADECIMENTOS

Inicialmente, gostaria de agradecer às minhas orientadoras Renata Galante e Karin Becker pelo excelente encaminhamento durante todo o mestrado; pelo extenso trabalho quanto ao estresse da nossa proposta; pela exigência quanto a qualidade e produtividade e pela paciência que sempre tiveram.

Aos professores do Instituto de Informática, pela oportunidade de cursar suas disciplinas. Em especial, aos professores Carlos Alberto Heuser e Leandro Krug Wives pelas contribuições durante o Seminário de Andamento que ajudaram a delimitar o escopo deste trabalho.

Aos amigos e colegas de laboratório pela companhia em discussões e aprendizado; pela troca de conhecimento; pelos desabafos e risos compartilhados e churrascos longamente planejados e não executados.

Ao meu pai, que mesmo a mares de distância sempre esteve ao meu lado e cujo caminho de trabalho, esforço e dedicação à familia tenho como exemplo de vida. À minha mãe, que mesmo distante sempre me transmitiu segurança e paz. Às minhas irmãs, com quem pude sempre contar. E por fim, à Deus que me guiou a seu tempo, me iluminando e colocando pessoas maravilhosas no meu caminho.

Um sincero obrigado à todos os que contribuíram direta ou indiretamente para a conclusão deste trabalho.

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

DOM	Document Object Model
CVS	Concurrent Version System
IaaS	Infrastructure as a Service
OWL-S	Ontology Web Language for Services
PaaS	Platform as a Service
SaaS	Software as a Service
SOA	Service-Oriented Architecture
SVN	Subversion
UML	Unified Modeling Language
W3C	World Wide Web Consortium
WSDL	Web Services Description Language
WSDL-S	Web Services Semantics
WSMO	Web Service Modeling Ontology
XML	eXtensible Markup Language
XSD	XML Schema Definition

LIST OF FIGURES

ABSTRACT

Service evolution requires sound strategies to appropriately manage versions resulting from changes during service lifecycle. Typically, a service version is exposed as a description document that describes the service functionality, guiding client developers on the details for accessing the service. However, there is no standard for handling the versioning of service descriptions, which implies on difficulties on identifying and tracing changes as well as measuring their impact, particularly in a finer grain perspective. Compatibility addresses the graceful evolution of services by considering the effects of changes on client applications. It defines a set of permissible change cases that do not disrupt the service external integration. However, providers cannot always guarantee that the necessary changes yield compatible service descriptions. Moreover, the concept of compatibility is often applied to the entire service description, which can not be representative of the actual use of the service by a particular client application. So, it is the client's developers responsibility to assess the extent of the change and their impact in their particular usage scenario, which can be hard and error-prone without proper change identification mechanisms. This work addresses service evolution in a finer grain manner, which we refer to as feature level. Hence, we propose a versioning model and a compatibility algorithm at feature level, which allows the identification and qualification of changes impact points, their ripple effect, as well as the assessment of changes' compatibility in this finer grain of features. This work also reports an experiment based on a real service, which explores the versioning model to assess the scope of implicit and explicit changes and their compatibility assessment.

Keywords: Service versioning, service compatibility, service evolution management.

Versionamento e Compatibilidade de Serviços em Nível de Feature

RESUMO

A evolução de serviços requer estratégicas para lidar adequadamente com a gerência de versões resultantes das alterações ocorridas durante o ciclo de vida do serviço. Normalmente, uma versão de serviço é exposta como um documento que descreve a funcionalidade do serviço, orientando desenvolvedores clientes sobre os detalhes de acesso ao serviço. No entanto, não existe um padrão para o tratamento de versões dos documentos que descrevem o serviço. Isso implica na dificuldade de identificação e localização de alterações, bem como na medição do seu impacto, especialmente em uma perspectiva mais granular. A compatibilidade aborda um estilo mais elegante de evolução de serviços, considerando os efeitos provenientes das alterações nas aplicações cliente. Ela define um conjunto de alterações permissivas, as quais não afetem a integração externa com o serviço. Entretanto, provedores não conseguem garantir que as alterações necessárias ao serviço estarão no conjunto de alterações compatíveis. Além disso, o conceito de compatibilidade é muitas vezes aplicado sobre a descrição do serviço como um todo, o que pode não ser representativo do uso real do serviço por uma aplicação cliente em particular. Assim, é de responsabilidade dos desenvolvedores clientes avaliar a extensão das alterações no serviço a fim de medir o impacto no seu cenário em particular. Esse processo pode ser difícil e propenso a erros sem o uso de mecanismos de identificação de mudanças. Este trabalho aborda a evolução do serviço de maneira mais granular, o que chamamos de nível de *feature*. Desse modo, nós propomos um modelo de controle de versões e um algoritmo de compatibilidade a nível de *feature*, que permite a identificação e qualificação do impacto das alterações, assim como a avaliação da compatibilidade das mudanças neste nível de *feature*. Este trabalho também apresenta um experimento com base em um serviço real, que explora o modelo de controle de versões para avaliar a extensão das mudanças implícitas e explícitas e sua avaliação de compatibilidade.

Palavras-chave: Versionamento de serviços, compatibilidade de serviços, gestão de evolução de serviços.

1 INTRODUCTION

Service Oriented Architecture (SOA) denotes an architectural approach that enables the creation of loosely coupled systems on top of autonomous components, referred to as services. The loose coupling is based on the fact that there is a well defined interface for the service, which exposes the characteristics relevant for its consumption. From the SOA perspective, a service is a set of functionality exposed by a provider to which consumers can bind their applications. In turn, consumers can expose their applications as services performing both the roles of consumer and provider. Organizations are encouraged to include services in their business model in order to implement new business processes using existing or third-party resources, as well as to embrace opportunities such as the SaaS/PaaS/IaaS (Software/Platform/Infrastructure). The lifecycle of decoupled components in SOA encourages the development of autonomous services and allows independence during the phases of development, deployment and maintenance. But services do not escape the necessity of dealing with change. In order to be aligned with new business opportunities, services are subject to constant variations, requiring appropriate strategies to handle multiple versions throughout their lifetime.

Service evolution management encompasses the creation, maintenance and decommission of different versions in a service provider environment (PAPAZOGLOU, 2008), which often leads to the maintenance of several concurrent versions. Service versioning is an overloaded term that may refer to the versioning of service implementation or its interface (BECKER et al., 2008; FRANK et al., 2008). In the context of service evolution, which regards the integration between consumers and providers, the versioning of the service interface is often addressed, particularly the service interface description.

The interface description of a service exposes the service version as a unilateral *contract* established by the provider, which guides clients on how to access service functionality. However, current notations for service interface description, including the standard WSDL/XSD, do not properly handle versioning (ANDRIKOPOULOS; BENBERNOU; PAPAZOGLOU, 2011). Typically, despite many service features remain 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). Typically, release notes describe 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 (FOKAEFS et al., 2011; ZOU et al., 2008). For instance, if a change is applied to a type that is referenced (directly or indirectly) by an operation, then this operation is also 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 (BECKER et al., 2008; ZOU 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 which are 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 (ZOU et al., 2008), change impact quantification (WANG; CAPRETZ, 2009), accurate recognition of changes (FOKAEFS et al., 2011), and usage oriented compatibility assessment (YAMASHITA et al., 2012; YAMASHITA; BECKER; GALANTE, 2011a).

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). For instance, providers can evaluate the trade-offs between the costs of provisioning multiple versions of a service, and the benefits of not disrupting their clients. However, traditional compatibility approaches (e.g. (BROWN; ELLIS, 2004)) are also document-oriented, which means that the assessment of compatibility among different versions is focused 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. By capturing the compatibility impact on a finer-grain, one can estimate more accurately the impact of changes (ANDRIKOPOULOS; BENBERNOU; PAPAZOGLOU, 2008), particularly regarding usage analysis on a specific client (PONNEKANTI; FOX, 2004) or a set of consumer applications (YAMASHITA et al., 2012).

In summary, service stakeholders lack proper mechanisms to easily recognize changes and their impact on evolving services. In this work, we address this deficiency by proposing a versioning model in a finer-grain perspective, referred to as *feature level*, and a compatibility assessment algorithm to automatically qualify versioned features with a compatibility verdict. In doing so, the objective of this work is an approach to version service features separately, along with their relationships, and identify the features directly or indirectly affected. This approach also permits to assess the compatibility between two feature versions in order to verify if the change can possibly affect any external integration of the service. As result, we can *quantify* and *qualify* service changes in a finer grain and thereby, provide invaluable information for service stakeholders with the purpose of helping them cope with changes during service evolution.

This work is part of a framework for supporting service evolution (YAMASHITA; BECKER; GALANTE, 2011a; YAMASHITA et al., 2012). As mentioned, incompatible changes might affect groups of applications very differently, depending on their use of the service. Therefore, the main goal of this framework is to measure change impact based on usage analysis. In this way, service providers have a more reliable understanding of change effects, and are able to develop strategies for service evolution.

The framework is composed by tree main modules, depicted in Fig. 1.1. The work described in this dissertation addresses the *Version Manager* module, which provides a finer-grained representation in order to easily locate and assess the compatibility of changes in service descriptions. Initial results regarding the the *Version Manager* are pre-

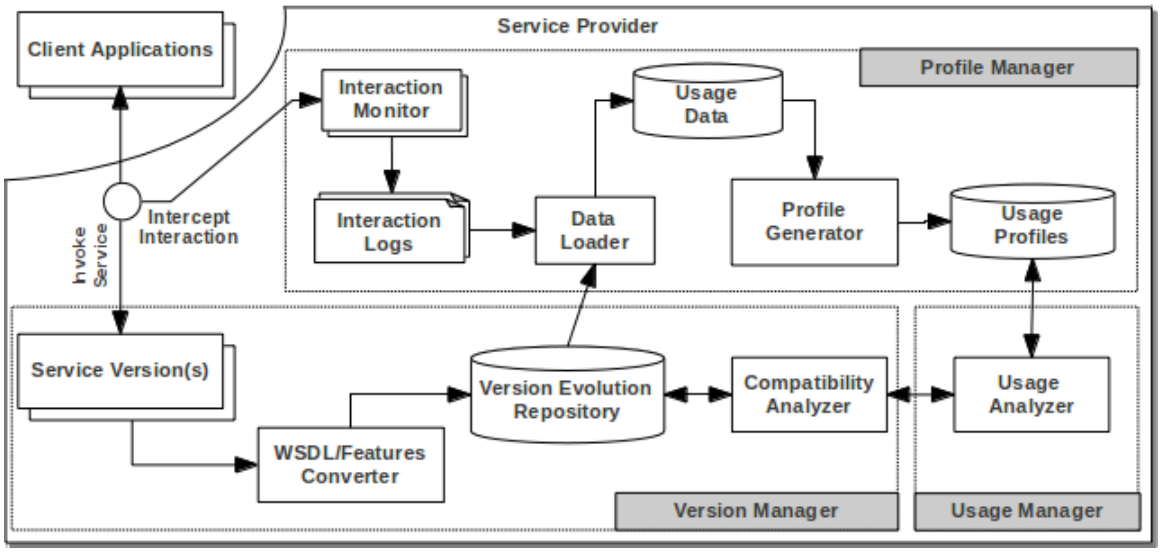


Figure 1.1: Change Management Framework based on Usage Profiles
Source: (YAMASHITA et al., 2012)

sented in (YAMASHITA; BECKER; GALANTE, 2011b) and (YAMASHITA; BECKER; GALANTE, 2012). The *Profile Manager* clusters clients applications based on similar patterns of usage using a knowledge discovery process, and summarizes them in usage profiles of applications enriched with relevant metrics for usage analysis. Finally, the *Usage Manager* enables to analyze the impact of changes with regard to the characteristics of usage profiles. Further details of the framework are discussed in (YAMASHITA et al., 2012).

Despite being part of a framework, the approach discussed in this work lays foundation for a wide spectrum of applications in the context of service evolution. For instance, the analysis of change impact propagation, which is a straightforward consequence of feature-oriented versioning, allows the quantification of change impact (WANG; CAPRETZ, 2009). It could also support the automatic creation of more detailed release notes based on usage analysis, as in (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 load balance management among implemented versions, which precedes the finer grain deployment in (TREIBER; ANDRIKOPOULOS; DUSTDAR, 2009).

The remaining of this work is structured as follows. Chapter 2 presents the main concepts for understanding this work. Chapter 3 presents related work. Chapter 4 describes the proposed feature-oriented versioning model. Chapter 5 describes the manner we propose to version and assess compatibility according to the model presented in Chapter 4. Experimental results are presented in Chapter 6. Finally, Chapter 7 presents the conclusions and future works.

2 CONCEPTS

In this chapter we present the main concepts that surround service evolution, which are core for understanding this work. First, we present the different perspectives regarding service versioning and highlight the one that focuses on the service external integration. Then, we present the most adopted notation for service version description, used throughout this work. As changes are inherent to services as every other software system lifecycle, we also present the concept of compatibility, which acts as a means to establish a verdict for inter-version agreement. Finally, we present an approach to manage service versions with regard their compatibility.

2.1 Service Versioning

Services provide an approach for developing applications as an additive layer on top of existing components and even though they are an evolutionary step beyond component software architecture, they do not escape the necessity of the software lifecycle and the necessity of change (BACHMANN, 2005). Service changes may originate from the modification of the service functionality to improve performance, regulatory constraints that require changes on the service behavior, among others (ANDRIKOPOULOS; BENBERNOU; PAPAZOGLOU, 2008). In this context, service versioning is the consequence of a service being advanced, adapted and adjusted over time.

There are two perspectives regarding service versioning: the *service implementation* and the *service interface* versioning (FRANK et al., 2008). Service implementation versioning concerns every single change in the service source code. These are commonly addressed with usual version control systems (e.g. CVS [1], SVN [2], Git [3], etc). Service interface versioning concerns the changes that can affect the external service integration, such as changes on the service request and response formats. In the context of this work, service versioning is addressed in the perspective of service interfaces. Therefore, we shall use the term *service version* as a synonym for *service interface version*. We formally address service version as follows:

Definition Given two services s_1 and s_2, of a same provider. If $s_1.name$ = $s_2.name$ and $s_1.interface \neq s_2.interface$ and $s_2 > s_1$. Then $s_2.interface$ is a version of $s_1.interface$. By $s_2 > s_1$, assume that s_2 is most recent than s_1.

Service versions are often exposed as interface description documents. An interface description document describes the service functionality, guiding client developers on

[1] `http://savannah.nongnu.org/projects/cvs/`
[2] `http://subversion.apache.org/`
[3] `http://git-scm.com/`

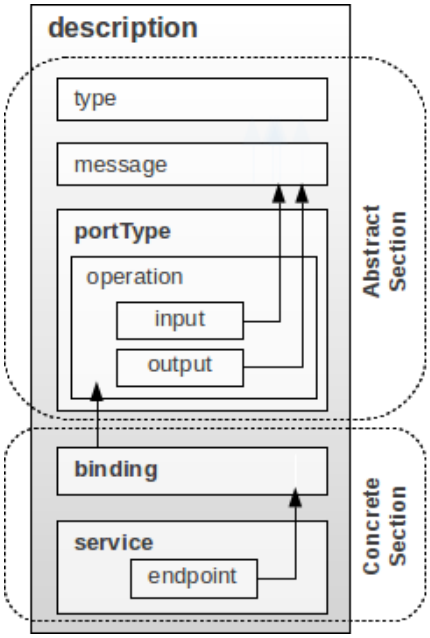


Figure 2.1: WSDL 1.x model
Source: Based on the illustrations in ([4])

the details for accessing the service and handle its response. In this work, we study services that have their versions described using Web Service Description Language 1.x (WSDL) (CHRISTENSEN et al., 2001).

WSDL provides a model based on an XML format to describe Web services in two fundamental sections that encompass the abstract functionality and the concrete details of the services. At the abstract section, the service is described in terms of the exchange of messages a service may receive or respond to. *Messages* are expressed by *type* elements which are described using an independent type system, typically a XML Scheme (XSD) within the WSDL document. *Operations* associate the parameters of its input or output messages to the types previously described. Finally, a *portType* describes the functionality of a service by defining the operations that can be performed and the messages required to perform the operations. At the concrete section, a service is defined as collections of network *ports* that implement a common portType. A port associates a network address with a *binding*, which describes the transport format details for one or more portTypes (CHRISTENSEN et al., 2001).

In (FRANK et al., 2008), it is argued that partitioning the service description into abstract and concrete description, in effect, separates the interface description from implementation. Since clients are bound to services providers through service interfaces, they form the basis for the decoupled lifecycle between service clients and providers. It is an important consequence from the perspective of SOA concept and justifies why Web services are widely adopted to build SOA based systems. The components of the WSDL model are shown in Figure 2.1.

The WSDL is the most common notation for service description and a W3C specification. Large scale providers such as eBay, Amazon and FedEx, are among the providers that expose service versions using WSDL. However, WSDL does not provide methods to systematically describe versions (CHRISTENSEN et al., 2001). Typically, providers

[4] `http://en.wikipedia.org/wiki/Web_Services_Description_Language`

make use of the *documentation* tag within the WSDL document to describe an identifier for the current version, but this tag is not specifically to handle version identifiers, which in turn has no consensus on its form. Alternative service description notations, including semantic ones such as WSDL-S [5], WSMO [6] and OWL-S [7], do not address versioning in a systematically manner either.

During service evolution, it is a common approach to create and expose a new service interface document for every change on the service functionality (FANG et al., 2007; BROWN; ELLIS, 2004; ENDREI et al., 2006). In doing so, consumers can be aware of the change and adapt their code whenever necessary. Also, service providers should ideally expose new service versions regarding new releases *and* maintain the old versions so consumers are not disrupted (ENDREI et al., 2006). However, the maintenance can become cumbersome as the service move through several versions (FANG et al., 2007; BROWN; ELLIS, 2004). To alleviate the problem of maintaining multiple versions of the same service the concept of *compatibility* became widely accepted.

2.2 Service Version Compatibility

Service version compatibility is a means to guarantee that, when introducing a new service version, stakeholders are not affected (PAPAZOGLOU, 2008). Compatibility can have different meanings depending on the context or perspective from which it is viewed. In the context of services, there are two kinds of compatibility to be considered: backward compatibility and forward compatibility (FANG et al., 2007).

Backward compatibility is concerned with how changes in the service versions affect existing service consumers (BECKER et al., 2008; FANG et al., 2007; ENDREI et al., 2006). Assuring backward compatibility means that an evolving service should continue supporting older clients as it changes its interface over time (ENDREI et al., 2006). On the other hand, forward compatibility concerns how service consumers are able to access older versions of a service without been downgraded (FANG et al., 2007).

A service version is defined to be backward compatible with regard to an older version if it delivers at least the same functionality and generates outputs that can be consumed by existing clients, without the necessity for clients to adapt their applications to the changes (ANDRIKOPOULOS; BENBERNOU; PAPAZOGLOU, 2011; FANG et al., 2007). In other words, backward compatibility ensures that current client applications are not affected by the changes (ENDREI et al., 2006). Forward compatibility aims at applying the same constraints on the opposite direction of versions relationship, i.e. from the older with regard to a newer one (FANG et al., 2007). In this work we shall use the term compatibility as a synonym for backward compatibility.

Compatibility is defined by the agreement of the changes occured between two service versions with regard a set of permissible or prohibited change cases, such as Table 2.1. We formally address compatibility as follows:

Definition Given two service versions $s_1.interface$ and $s_2.interface$, in which $s_2 > s_1$. Let $C = \{c_1, c_2, ..., c_n\}$ be the set of changes occurred to $s_2.interface$ with regard to $s_1.interface$ and $P = \{p_1, p_2, ..., p_n\}$ a set of incompatible change cases. Then, $s_2.interface$ is compatible with regard $s_1.interface \iff \forall c_i \in C, c_i \notin P$.

[5] http://www.w3.org/Submission/WSDL-S/

[6] http://www.wsmo.org/

[7] http://www.daml.org/services/owl-s/1.1/

By the compatibility definition it is derived the following proposition, which regards the affects of changes on client applications.

Proposition Given two service versions $s_1.interface$ and $s_2.interface$ and a set of client applications $C = \{c_1, c_2, ..., c_n\}$, which represents all the client applications that consume $s_1.interface$. Let $c_i.s_j$ be a client application i that consumes a service version j. Then, $s_2.interface$ is compatible with regard $s_1.interface \Rightarrow \forall c_i \in C, c_i.s_1 \equiv c_i.s_2$.

Table 2.1 summarizes the change core cases along with their correspondent backward compatibility verdict. For instance, adding an operation to a new version does not represent incompatibility with regard to the external integration of the service, whereas removing an operation can disrupt consumer applications. Notice that in Table 2.1, only a very restricted set of changes are compatible. In fact, only the addition of an operation or the addition of an independent complex data type are compatible (ANDRIKOPOULOS; BENBERNOU; PAPAZOGLOU, 2011). By complex data type it is assumed the WSDL *message* element and the *types* defined within the XSD description. Some change can be compatible or not depending on the approach that discuss this particular change.

Table 2.1: Change core cases for interface compatibility assessment

Cases	Change	Interface Element	Description	Verdict
1	Add	Operation	Add new operation	*Yes*
2	Add	Type	Add new independent complex data type	*Yes*
3	Add	Type	Add optional type as an operation input parameter	*Yes/No*
4	Add	Type	Add optional type as an operation output parameter	*Yes/No*
5	Add	Type	Add mandatory type as an operation parameter	*No*
6	Update	Operation	Rename operation	*No*
7	Update	Type	Rename complex data type	*No*
8	Update	Type	Change input parameter (type) from mandatory to optional	*Yes/No*
9	Update	Type	Change output parameter (type) from optional to mandatory	*Yes/No*
10	Update	Type	Change type from optional to mandatory	*No*
11	Update	Type	Change primitive type with guarantee of not loosing information (e.g. *Integer* to *Float*, etc)	*Yes/No*
12	Update	Type	Change primitive type without any guarantee of not loosing information (e.g. *Float* to *Integer*, etc)	*No*
13	Update	Type	Change the structure of a complex data type regarding the order of its subtypes	*No*
14	Remove	Operation	Remove operation	*No*
15	Remove	Type	Remove type (dependent of another element)	*No*

Table 2.2: Works addressing compatibility cases

Works	**Compatible Cases**
(BROWN; ELLIS, 2004)	1, 2
(PONNEKANTI; FOX, 2004)	1, 2, 3
(ENDREI et al., 2006)	1, 2, 3, 8
(FANG et al., 2007)	1, 2, 3, 4
(BECKER et al., 2008)	1, 2, 3, 8
(ANDRIKOPOULOS; BENBERNOU; PAPAZOGLOU, 2011)	1, 2, 3, 8, 9, 11

In general, an approach considers an incompatible case as compatible by having the service stakeholders in accordance with predefined requirements, which relax compatibility. Table 2.2 relates the works that address compatibility with the change cases in Table 2.1.

Nevertheless, the assessment of compatibility is important in the context of service evolution because it can support providers on the decisions involving service version management (SILVA et al., 2012). The compatibility core cases are also discussed in Section 3.2 and further stressed by our algorithm in Section 5.5.

2.3 Version Management Lyfecycle

The evolution of a service is expressed through the creation and decommission of its different versions during its lifetime (PAPAZOGLOU, 2008). Thus, as services evolve, providers need mechanisms for managing the the different versions of a service.

To introduce change management, (ENDREI et al., 2006) specify a flow script for managing concurrently versions of a service, as shown in Figure 2.2. The first step on introducing a new version is to assess its compatibility with regard to the current one. The result can lead to the decommission of a version or the need to concurrently maintain different versions of the same service. In both cases, the older version should be set

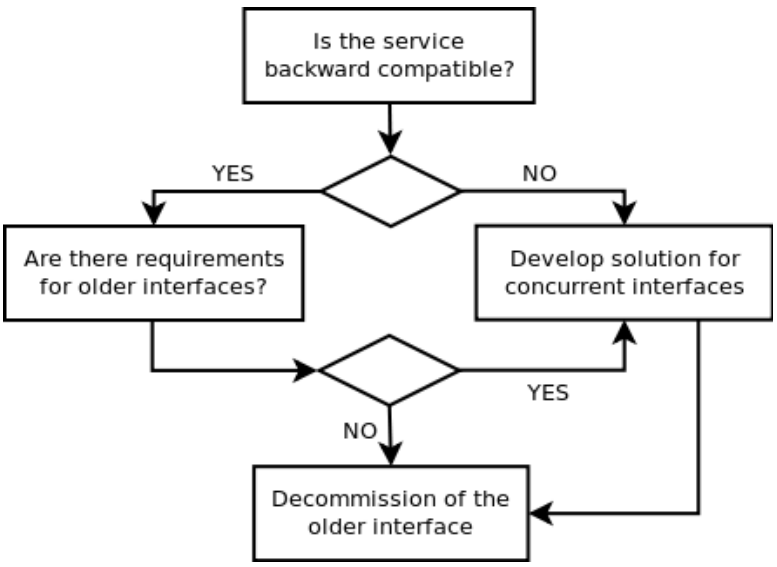


Figure 2.2: Abstract feature representation

as deprecated and it should be determined a grace period for the clients to adapt their applications to the new version before the older one is decommissioned (ENDREI et al., 2006). Alternatively, the provider can choose to support only one version at a time, but taking the risk of breaking clients applications when changes are incompatible.

2.4 Concluding Remarks

In this chapter, we presented the perspectives for versioning services and the most adopted notation for describing service versions. Also, we presented the concept of compatibility, which also acts as a mechanism to help service providers on measuring the impact of changes among versions. Together with the concept of compatibility, we presented a summary of compatibility change operations addressed in related works. Finally, we presented an approach that address the service versioning lifecycle.

The above concepts are inherent to this work, since we propose an approach for versioning services and assess their compatibility on a finer-grain level.

3 RELATED WORK

In this chapter we present the main works regarding service evolution, which are related with this work. As this work proposes a versioning and compatibility approach in a finer-grain, we begin by presenting the main works regarding service versioning and then, we present the main works that address compatibility. We also present a brief comparison of the approaches.

3.1 Service Versioning Approaches

Current approaches on service evolution address service versioning and service compatibility, in order to achieve some degree of conformance and/or transparency during evolution. The main difficulty is that there is *no standard* for handling service versioning, i.e., current interface description notation do not handle versioning.

In order to overcome this deficiency, best practices were proposed as a manner to guide service providers on how to version their services. Common best practices include the use of different XML namespaces for every version that potentially disrupts client applications; version identifiers for unambiguously naming versions; or a combination of these (BROWN; ELLIS, 2004; ENDREI et al., 2006; ANDRIKOPOULOS; BENBERNOU; PAPAZOGLOU, 2011). These techniques can be used to version a service, as well as its information type container, which is commonly an XML Schema Document (XSD) within the WSDL document[1].

The work of (BROWN; ELLIS, 2004) summarizes the best practices for dealing with service versioning. It is presented a set of solutions and basic definitions in order to version services, such as maintaining service descriptions as documentation for compatible versions and using different namespaces for every incompatible one, which are identified by a date or version stamp in accordance with W3C schema naming. As for incompatible versions, it is suggested the implementation of intermediate routers in order to redirect client requests to older implemented versions.

The work in (ENDREI et al., 2006) provides techniques and guidelines for service versioning and change management. It is presented different versioning techniques for service versioning based on the service name, endpoint and operation. An important contribution is the change management flow, which was previously illustrated in section 2.3.

The work in (FANG et al., 2007) addresses the necessity of supporting version representation within existing service standards by proposing the insertion of a version element into the service version description, specifically in the WSDL description document. Then, it is proposed a version-aware model within a service registry directory, which is

[1] `http://www.w3.org/XML/2005/xsd-versioning-use-cases/#p871`

Table 3.1: Related works characteristics regarding versioning

Approaches	Best practices	Versioning Model	Finer-grain perspective	Cope with W3C standards
(BROWN; ELLIS, 2004)	guidelines	–	–	Yes
(ENDREI et al., 2006)	guidelines	–	–	Yes
(FANG et al., 2007)	–	adapt current standards	–	Yes
(BECKER et al., 2008)	–	framework	type level	–
(LEITNER et al., 2008)	–	framework	–	Yes

Note: We use hyphen (–) for works that does not address the topic in each column

able to handle the relationships of different versions. Additionally, by the use of notification components, the approach in (FANG et al., 2007) provides a means for service consumers to be aware of new or deprecated service versions.

Alternatively, (LEITNER et al., 2008) proposes a version-graph model within a custom registry framework, which is able to store and maintain the relationships between service versions. In doing so, this approach provides a degree of transparency for service costumers, which can freely choose among the existing versions the one to bind their applications to. However, it is left to the service provider the responsibility to identify and specify the degree of changes between versions, which can be hard, error-prone and costly if performed manually (BECKER et al., 2008).

The above approaches address the versioning of the *entire* interface description document, which represent the service version. However, even if these approaches are sufficient for the recording and communicating the different versions of a service (ANDRIKOPOULOS; BENBERNOU; PAPAZOGLOU, 2011), the focus on the entire description document leads to difficulties on locating and tracking changes and thereby, measuring the actual impact of a change between versions, particularly regarding compatibility (ZOU et al., 2008; FOKAEFS et al., 2011).

(BECKER et al., 2008) addresses this difficulty by proposing a framework to automatically detect the differences among service versions, which compares the representation of versions description on object-oriented SOA schemes. The proposed framework allows service descriptions to evolve in different granularity levels, by considering a loose dependency between the services and the elements used to describe them. Hence, the framework also considers the possibility of reusing elements across several services. However, the proposed approach is based on the object-oriented paradigm and does not fit W3C current standards.

We summarize in Table 3.1 the approaches strictly regarding the topic of versioning. As highlighted in Table 3.1, early works were targeted in specifying guidelines on how to version services, whereas latter ones proposed alternatives for handling service versions in custom frameworks. Almost all of them, are focused on coping with current service description standards. Also, (BECKER et al., 2008) started pointing out the necessity of a finer-grained representation as a manner to leverage the reuse of unaltered elements across versions. In fact, the lack of a finer-grain perspective also leads to difficulties on managing versions, which has no mechanisms for tracing what have been altered during evolution. With regard to service versioning, we propose in this work a versioning model

that enables a finer grained perspective of services, and which is aligned with W3C current standards.

3.2 Service Compatibility Approaches

As we previously mentioned, compatibility defines a set of change cases that can (or can not) be applied to the service versions in order to maintain the service external integration. Compatibility is discussed in different aspects such as non-disruptive change cases and their verdict assessment.

With respect to compatibility guidelines, many works have discussed change cases as a means to establish or summarize empirical findings and common sense rules (BROWN; ELLIS, 2004; ENDREI et al., 2006; FANG et al., 2007; BECKER et al., 2008; ANDRIKOPOULOS; BENBERNOU; PAPAZOGLOU, 2011). Nevertheless, the change cases that comply with compatibility are very restricted (see Table 2.1), which limits service providers to few types of changes if they look forward to maintaining service versions compatible. Thus, in order to enable a wider range of compatible changes, many works propose means to relax compatibility rules but requiring preconditions to the service provider or other stakeholders.

For instance, providers can relax the format of operation inputs by the addition of optional types and maintain version compatibility *if* their systems are conditioned to handle the new format of messages. Similarly, providers can change some primitive types (ANDRIKOPOULOS; BENBERNOU; PAPAZOGLOU, 2011), such as changing their types from integer to float in order to receive a more detailed value as input.

The above examples represent a degree of flexibility on the service request messages, since the provider can state and ensure the continuity of operation. However, the use of relaxing cases on response messages is not compatible, because there is no guarantee that consumers can adjust their applications to handle the relaxing cases. Even not ensuring compatibility, some providers (e.g. Ebay) make use of relaxed cases on response messages, which can impact on consumers applications. As a manner to soften the impact, the provider establishes a service evolution policy that informs to consumers guidelines on how to build their applications, for instance, expecting and handling unrecognized data. In practice, it is no more than an unilateral contract that does not guarantee compatibility. As alternative, stakeholders can make use of bilateral contracts, as the contract compliant approach proposed in (ANDRIKOPOULOS; BENBERNOU; PAPAZOGLOU, 2009), which delineates a set of permissible changes, but can restrict the evolution of a service.

With respect to compatibility assessment, many works have stated the need for mechanisms to formally of automatically identifying and assessing the compatibility of changes (ANDRIKOPOULOS; BENBERNOU; PAPAZOGLOU, 2011; YAMASHITA; BECKER; GALANTE, 2011a; BECKER et al., 2008; PONNEKANTI; FOX, 2004). For instance, the work of (ANDRIKOPOULOS; BENBERNOU; PAPAZOGLOU, 2011) addresses service compatibility by proposing a service evolution framework based on an abstract model for the description of services, which details all its components such as structural, behavioral and QoS-related. The proposed framework basis a T-shaped compatibility model that address either the compatibility of service versions and the interrelationship of different services. Then, it is described a compatibility check algorithm in order to *formally* assess the compatibility verdict of service versions regarding the T-shape model. With regard to the structural components of the abstract model, an operation

Table 3.2: Related works characteristics regarding compatibility

Approaches	**Change cases**	**Assessment framework**	**Finer-grain perspective**	**Cope with W3C standards**
(PONNEKANTI; FOX, 2004)	guidelines	usage-oriented	identification	Yes
(BROWN; ELLIS, 2004)	guidelines	–	–	Yes
(ENDREI et al., 2006)	guidelines	–	–	Yes
(FANG et al., 2007)	summarization	–	–	Yes
(BECKER et al., 2008)	relaxing cases	automatic	assessment	–
(ANDRIKOPOULOS; BENBERNOU; PAPAZOGLOU, 2011)	relaxing cases	formal	identification	Yes
(YAMASHITA; BECKER; GALANTE, 2011a)	–	automatic/ usage-oriented	identification/ assessment	Yes

Note: We use hyphen (–) for works that does not address the topic in each column

is composed by messages that in turn are aggregation of information types. In this model, operations, messages and information types are elements of the service evolution framework.

A compatibility checking algorithm is proposed in (BECKER et al., 2008), which automatically assesses the compatibility verdict of service versions along with the set of elements that describe the service data types. This approach also assumes flexibility on request messages by verifying input parameters formats. It represents a very useful mechanism that enables the service provider to understand the impact of changes during evolution. However, it targets on classes that represent service descriptions and does not directly cope with W3C current standards.

Another important approach is the work of (PONNEKANTI; FOX, 2004). It addresses service interoperability, which is the ability to replace one service with another one (from the consumers perspective) in terms of client usage. Although not directly related with service evolution, which regards the compatibility of different versions of a *same* service, the work of (PONNEKANTI; FOX, 2004) contributes to the current works by providing a first perspective of compatibility in a finer-grain, together with the perspective of usage. It also presents important results regarding usage, which reveals that consumers applications usually only access a fraction of the set of service functionality. This statement implies that the compatibility assessment of the entire service version is not representative of the actual service usage from a set of consumers applications or an individual usage scenario. The work of (YAMASHITA; BECKER; GALANTE, 2011a) makes use of this statement on suggesting the benefits of compatibility assessment in a finer-grain by considering compatibility against specific usage profiles, which preliminary results are presented in (YAMASHITA et al., 2012).

We summarize in Table 3.2 the approaches regarding service version compatibility.

In Table 3.2, early works were focused on identifying and summarizing change cases and their compatibility verdict, whereas latter ones proposed mechanisms to formally or automatically assess compatibility. Many works discussed flexibility by relaxing compatibility rules, although every relaxed case require a precondition that depending on the affected stakeholder can not be achieved. Almost all of the presented works were focused on coping with current service description standards. Also, many works address a finer grain of change identification in order to identify the nature of the change (structural or behavioral) (ANDRIKOPOULOS; BENBERNOU; PAPAZOGLOU, 2011), verify and reuse the assessment of independent service data types (BECKER et al., 2008) and verify the actual impact of changes based on a particular usage scenario (PONNEKANTI; FOX, 2004) or usage profiles (YAMASHITA; BECKER; GALANTE, 2011a). With regard to service compatibility, we propose an automatic compatibility assessment on a finer grain based on the algorithm of (BECKER et al., 2008), but aligned with W3C standards. As result, we aim to enable the compatibility assessment based on usage profiles proposed in (YAMASHITA; BECKER; GALANTE, 2011a).

3.3 Concluding Remarks

In the above sections, we presented the main works regarding service versioning and compatibility assessment. We compared the related works considering service versioning and compatibility separately. We commented our approach regarding both topics, which in summary, proposes a versioning model and compatibility algorithm in a finer grain that is aligned with current W3C standards and enables the automatic assessment of compatibility on this finer grain of service representation.

4 A VERSIONING MODEL AT FEATURE LEVEL

In this chapter, we propose a versioning model that aims to version the features of a service interface separately. Our final goal is to support the analysis of compatibility on a finer grain. In doing so, it is possible to analyze changes in a finer grain perspective and pinpoint the affected features ripple effect. As result, we can estimate the consumer disruption more accurately.

4.1 Feature-oriented Versioning Model

In order to understand service evolution on a finer grain, we need to emphasize that client applications are not bound to the entire set of functionality exposed by a service, as pointed in (PONNEKANTI; FOX, 2004). In fact, clients are bound to a subset of functionality (e.g. operations) along with their expected and delivered data type structure. For this purpose, we characterize a *service* as a composition of *operations*, through which data is exchanged according to recursively predefined *types* (e.g. messages, schema elements). 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; KANG; LEE, 2002), which is characterized as a versionable element. From an abstract perspective, those elements are addressed in a similar manner in (ANDRIKOPOULOS; BENBERNOU; PAPAZOGLOU, 2011). The abstract representation of features and their relationship is depicted in Figure 4.1.

We propose to version features separately while maintaining their relationships *through their versions*, which is represented by a graph of features. As mentioned, versioning features separately provides a means to identify affected features during versioning, whereas maintaining their dependency relationships enables the consistency with the interface and further analysis to the change propagation effects.

The idea of feature-oriented versioning relies on providing the abstract management of the different parts of the interface description in order to version only changed features, rather than the entire service interface. Hence, when a new service interface document is exposed, we convert it to an abstract internal representation, compare the textual descriptions of the features with regard to previously existing ones, as well as their relationship

Figure 4.1: Abstract feature representation

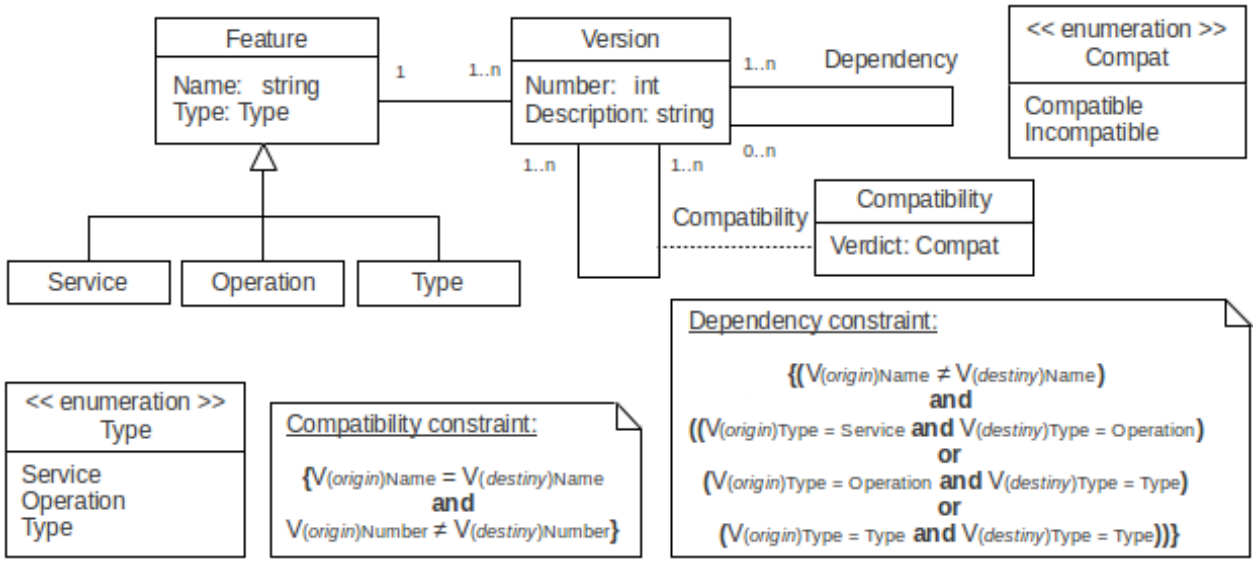


Figure 4.2: Versioning Model at Feature Level

with other features, and create new versions only when changes occurred. Notice that this conceptual view of a service is relatively independent of service description notation.

The versioning model is depicted in Figure 4.2 using a UML class diagram. A *Feature* is generalization of *Service*, *Operation* and *Type*. Each *Feature* is identified by a *Name* and has at least one *Version*, which in turn can have many dependency relationships of versions of another features. In order to comply with the abstract feature representation, a constraint is set on the dependency relationship to assure a) that a feature version does not depend on another version of the same feature and b) the representation hierarchy of feature types (Figure 4.1). Notice that as we propose to version the operations and types of a service separately, the feature service in the version model represents a shell of the former service, which holds the dependencies for its operations and types.

Versions are uniquely identified by a pair <*Feature.name, Version.number*>, referenced throughout the remaining of this work as $v_{featureName,versionNumber}$. The *Version.Description* attribute corresponds to the textual description of the WSDL document, according to the feature type (service, operation or type). The mapping in Section 4.2 defines the fragment of description within the WSDL document that is related to each type of feature. The compatibility relationship between two versions of a same feature is also maintained and assessed, as described in more details in section 5.5. In order to be aligned with the concept of compatibility, a constraint is set on the compatibility relationship to impose the relationship only between versions of a same feature. Notice that while the dependency relationship regards the versions of different features, the compatibility relationship addresses different versions of a same feature.

4.2 Feature Mapping

In order to extract the textual description of WSDL/XSD to the proposed feature-oriented version model, we have established the following correspondence for feature mapping:

- *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, we only consider for versioning the ones defined outside the context of XSD complex elements, which means that we version only types meant for reuse. Consequently, we do not version neither primitive types (e.g. *string*, *double*, etc), nor complex types that cannot be referenced elsewhere.

- *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 because of its high extensive adoption. For instance, services such as eBay Services [1], Amazon EC2 [2], Google Search [3], FedEx Services [4], PayPal Services [5] among others, use WSDL version 1.x to expose their service versions. Although WSDL 1.x do not represent the latest version of the WSDL specification, our mechanism can be adapted to the more recent WSDL 2.0 with small changes.

4.3 Illustration

We illustrate in Figure 4.3 and Figure 4.4 the manner we map the fragments of a WSDL 1.6 description to the proposed representation, using the *StockQuote* service [6]. Figure 4.3 presents the entire interface description for the *StockQuote* service, which has an operation (*GetLastTradePrice*), two complex data types (*TradePriceRequest* and *TradePrice*) and two exchange messages (*GetLastTradePriceInput* and *GetLastTradePriceOutput*) used as a bridge to bind the operations to their types. Hence, we propose to separate the *StockQuote* description into fragments to represent the service, its operation and types separately. The result of this fragmentation is depicted in Figure 4.4, in which every fragment of description is associated with a feature. For instance, *GetLastTradePrice* feature is associated with the description fragment of Figure 4.4b.

Moreover, we identify in the description fragments the dependency of each feature, which in case of *GetLastTradePrice* for instance, are *GetLastTradePriceInput* and *GetLastTradePriceOutput*. We maintain the dependency relationships on a graph containing the features versions, which is rooted in the feature that represents the service. The features graph that represents the *StockQuote* service in a finer-grain is illustrated in Figure 4.5.

[1] `https://www.x.com/developers/ebay`
[2] `http://aws.amazon.com/pt/ec2/`
[3] `https://developers.google.com/soap-search/` (until 2011)
[4] `http://www.fedex.com/us/fcl/pckgenvlp/developer-resources/index.html`
[5] `https://www.paypalobjects.com/wsdl/PayPalSvc.wsdl`
[6] The *StockQuote* service is the default example in the WSDL documentation page (CHRISTENSEN et al., 2001).

```
<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>
```

Figure 4.3: WSDL 1.1 description

```
<definitions name="StockQuote"
    targetNamespace="http://example.com/stockquote.wsdl">
  <types>
    <schema>
    </schema>
  </types>
  <portType name="StockQuotePortType"></portType>
  <binding name="StockQuoteSoapBinding"type="tns:StockQuotePortType">
    <soap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http"/>
  </binding>
  <service name="StockQuoteService">
    <documentation>My first service</documentation>
    <port name="StockQuotePort"
      binding="tns:StockQuoteBinding">
      <soap:addresslocation="http://ex.com/stockquote"/>
    </port>
  </service>
</definitions>
```

(a) Description associated with *StockQuote* service

```
<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>
```

(b) Description associated with *GetLastTradePrice* operation

```
<message name="GetLastTradePriceInput">
  <part name="body" element="xsdl:TradePriceRequest"/>
</message>
```

(c) Description associated with *GetLastTradePriceInput* type

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

(d) Description associated with *GetLastTradePriceOutput* type

```
<element name="TradePriceRequest">
  <complexType>
    <all><element name="tickerSymbol" type="string"/></all>
  </complexType>
</element>
```

(e) Description associated with *LastTradeRequest* type

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

(f) Description associated with *TradePrice* type

Figure 4.4: WSDL 1.1 description and Feature description

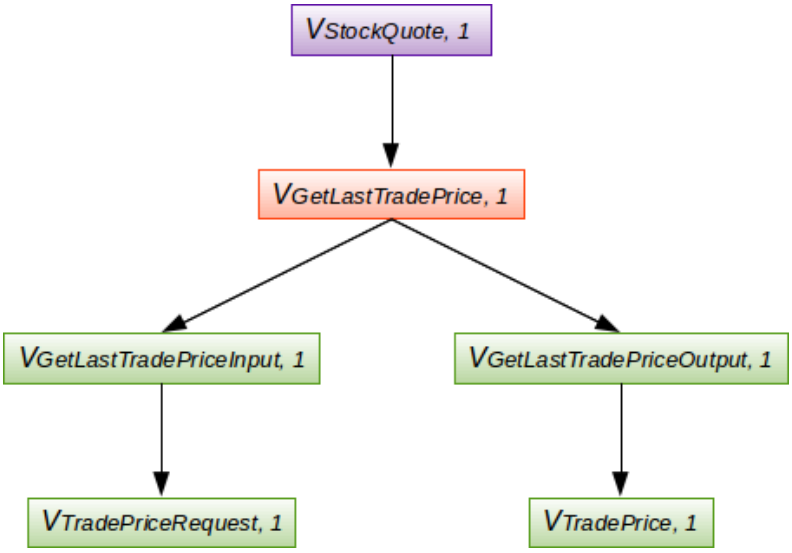


Figure 4.5: Dependency graph representing *StockQuote* version 1

4.4 Concluding Remarks

In this chapter we presented a feature-oriented versioning model, which aims to enable the versioning of the entire service in a finer-grained manner. We presented the model and explained the manner we propose to map the elements of a service interface to the proposed feature level representation. Also, we illustrated the resulting of the service fragmentation as a rooted graph composed by feature versions, which aims to maintain the dependency relationships of features. In summary, this chapter presents the theoretical foundation that basis the WSDL-to-features Converter and the Compatibility Assessment of versions, which are detailed in the next chapter.

5 VERSION MANAGER

In this chapter we present the Version Manager, which is part of the Change Management Framework (Figure 1.1). Hence, we introduce in this chapter the Version Manager components and explain our approach in order to version features and assess their compatibility in a finer grain perspective.

5.1 Version Manager Components

The objective of the Version Manager is to implement the concepts of the feature-oriented model presented in Chapter 4, and thus, the versioning of a service at feature level, as well as the assessment of compatibility for versions in this finer-grain model of features. In summary, the contribution of the Version Manager is twofold: a) the quantification of features affected by the changes between two service versions, and b) the qualification of the change regarding each feature.

The Version Manager, depicted in Figure 5.1, is composed by three components: the *WSDL/Features Converter* that is responsible for extracting the features from the service versions with regard to the feature-oriented model ; the *Compatibility Analyzer* that aims to assess the compatibility of changes between features versions; and the *Version Evolutionary Repository*, which stores features versions, together with their dependencies. The Version Manager components are following described in this chapter.

5.2 WSDL/Features Converter

As mentioned, a service is represented by an interface description document (WSDL). In turn, a feature version corresponds to a fragment of the interface description document together with a reference to its dependencies. Hence, in order to version the features of a service version separately, we need to identify the features within the service interface description document, relate them to the appropriate feature versions, which includes the possibility of creating new versions in this process, and store this abstract representation

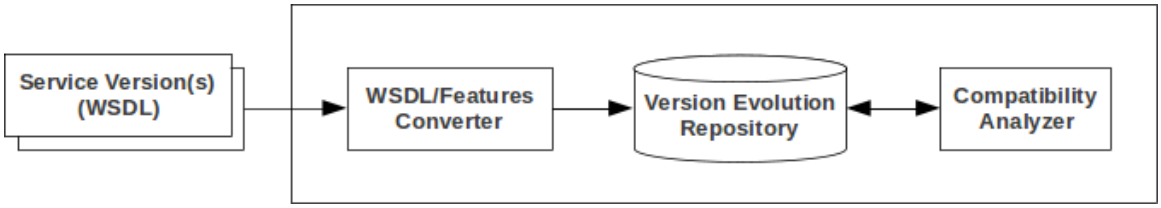


Figure 5.1: Version Manager components

in the *Version Evolution Repository*. This requires two phases: a) the extraction of the features from the interface description document, and b) the analysis of the features in order to discover whether they have changed regarding all their previous representation in the repository.

In versioning features, we intend to version only explicitly changed ones, or features that are indirectly affected by the changes. We refer to *changed* as a feature that either 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. Also, we refer to *affected* as a feature that did not explicitly changed, but depends on a feature that has changed directly or indirectly.

The first phase on versioning features is the extraction of the feature representation from the interface description document. The extraction of features encompasses the following steps:

1. the parsing of the interface description document (e.g. Figure 4.3) in order to identify the features and their relationships;

2. the generation of a graph of features' versions (e.g. Figure 4.5) with regard to the abstract feature representation (e.g. Figure 4.1); and

3. the process of relating the fragments of description to their correspondent feature version (e.g. Figure 4.4), following the mapping presented in section 4.2.

The resulting graph encompasses all the content of the interface description document, but the ability to represented the service as a set of features and their relationships allows us to analyze them separately. Once the graph of features' versions is assembled, the second phase on versioning features is to analyze each feature in the graph in order to discover whether it has changed or affected, and create a new version in the repository when that occur. Thus, for each feature, the *WSDL/Features Converter* analyzes the features' versions with the same name in the *Version Evolution Repository* in order to compare its textual description 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, which we refer to as *versioning cases*:

1. If the feature does not exist in the *Version Evolution Repository*, then the feature is created together with its first version.

2. If the feature already exists (i.e. it was previously versioned) and its description differs from the last version of this particular feature, then it is marked as *changed* in the graph and a new version for this feature is created.

3. If the feature already exists and its description is equal to an existing version, then:

   (a) If it depends on another feature that has been already marked as changed, then a new version is created due propagation effects (i.e. the feature was *affected* by a change).

   (b) 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 in the *Version Evolution Repository*.

5.3 Version Evolution Repository

As mentioned, the *WSDL/Features Converter* needs to compare the new feature versions with the existing ones in order to version only the changed/affected ones. Existing features' versions are maintained within the *Version Evolution Repository*, which is responsible for storing the features' versions along with their dependency relationships. Features' versions are stored following the schema presented in Figure 5.2, which is aligned with the versioning model of Figure 4.2. Relationship constraints are not addressed in the *Version Evolution Repository* schema.

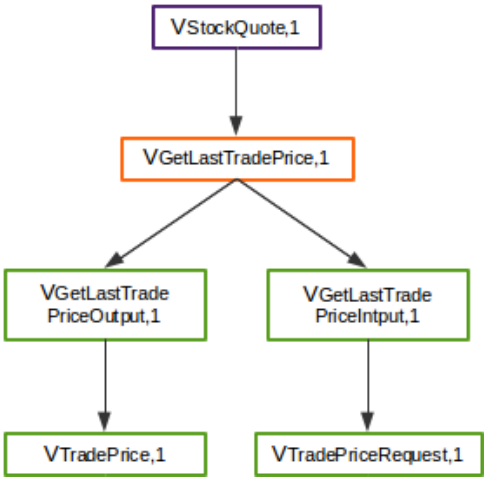
```
<?xml version="1.0" encoding="UTF-8"?>
  <schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified">

    <xs:element name="repository" type="repositoryType" />
    <xs:element name="feature" type="featureType" />
    <xs:element name="version" type="versionType" />
    <xs:element name="dependency" type="dependencyType" />
    <xs:element name="compatibility" type="compatibilityType" />

    <xs:group name="featureTypes">
      <xs:choice>
        <xs:element ref="service" />
        <xs:element ref="operation" />
        <xs:element ref="type" />
      </xs:choice>
    </xs:group>

    <xs:complexType name="repositoryType">
      <xs:sequence>
        <xs:element ref="version" minOccurs="1" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>

    <xs:complexType name="featureType">
      <xs:attribute name="name" type="xs:string" use="mandatory" />
      <xs:attribute name="type" type="featureTypes" use="mandatory" />
    </xs:complexType>

    <xs:complexType name="versionType">
      <xs:restriction base="featureType">
      <xs:attribute name="number" type="xs:integer" use="mandatory" />
      <xs:attribute name="description" type="xs:string" use="mandatory" />
      <xs:sequence>
        <xs:element ref="dependency" minOccurs="0" maxOccurs="unbounded" />
        <xs:element ref="compatibility" minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>

    <xs:complexType name="dependencyType">
      <xs:sequence>
        <xs:element ref="version" minOccurs="1" maxOccurs="1" />
      </xs:sequence>
    </xs:complexType>

    <xs:complexType name="compatibilityType">
      <xs:sequence>
        <xs:element ref="version" minOccurs="1" maxOccurs="1" />
      </xs:sequence>
    </xs:complexType>
</schema>
```

Figure 5.2: Version Evolution Repository Schema

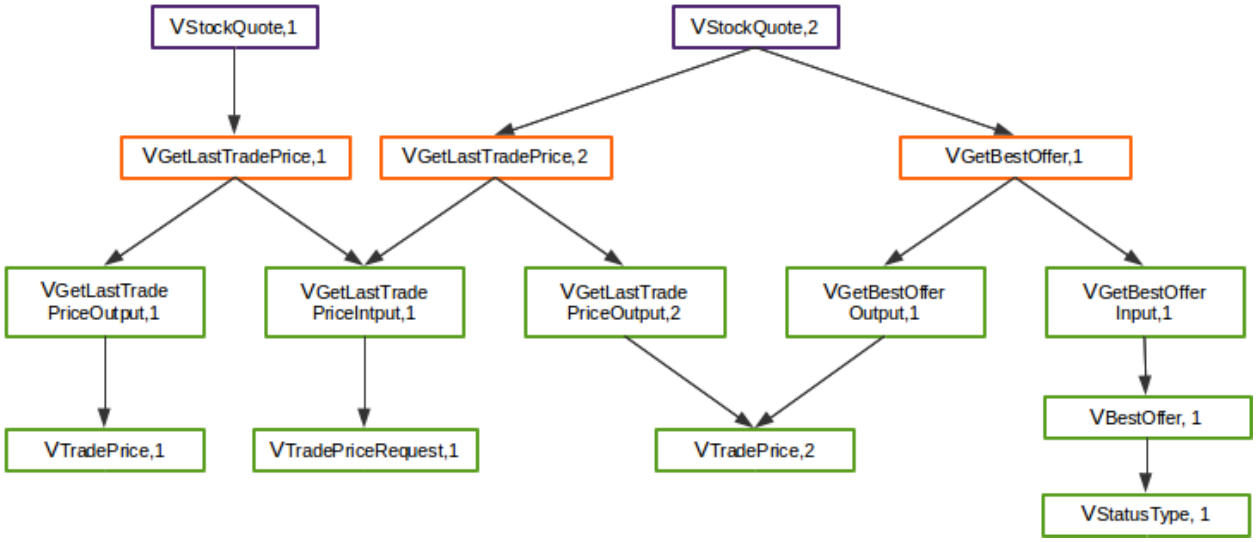
5.4 Versioning Process Illustration

To illustrate the process of feature versioning, suppose a provider exposes the interface description for the first version of *StockQuote* service, depicted in Figure 4.3. So, in order to incorporate the first version of *StrockQuote* into the *Version Evolution Repository*, the *WSDL/Features Converter* parses the WSDL document that represents the *StrockQuote* version for the purpose of identifying the features and their relationships.

After parsing the *StrockQuote* version, the *WSDL/Features Converter* generates a graph of features, in which the vertices of the graph represent the *StrockQuote* features and the edges represent the relationship among features with regard the abstract features representation. Next, the *WSDL/Features Converter* relates every feature in the graph with its correspondent description fragment from the WSDL document resulting in the graph depicted in Figure 5.3a, of which the features are associated with the fragments



(a) Repository state after adding $v_{StockQuote,1}$



(b) Repository state after adding $v_{StockQuote,2}$

Figure 5.3: Repository state through versioning

in Figure 4.4. As none of the features exists in the *Version Evolution Repository*, they are created together with their first version (*versioning case 1*). The state of the *Version Evolution Repository* after the addition of first *StockQuote* version is as depicted in Figure 5.3a.

Suppose now that the provider exposes a new version of the *StockQuote* service, which has two major changes:

1. a new operation together with its related types exchanged in messages, and;
2. changes in the description fragment of an existing feature.

These changes are depicted by the underlined text in Figures 5.4 and 5.5, which illustrate the new *StockQuote* version.

```
<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="double" /></all>
        </complexType>
      </element>
      <element name="BestOffer">
        <complexType>
          <sequence>
            <element name="bestPrice" type="double"/>
            <element name="status" type="StatusType"/>
          </sequence>
        </complexType>
      </element>
      <simpleType name="StatusType">
        <restriction base="xs:token">
          <enumeration value="Active"/>
          <enumeration value="Expired"/>
        </restriction>
      </simpleType>
    </schema>
  </types>
  <message name="GetLastTradePriceInput">
    <part name="body" element="xsd1:TradePriceRequest" />
  </message>
  <message name="GetLastTradePriceOutput">
    <part name="body" element="xsd1:TradePrice" />
  </message>
  <message name="GetBestOfferInput">
    <part name="body" element="xsd1:BestOffer"/>
  </message>
  <message name="GetBestOfferOutput">
    <part name="body" element="xsd1:TradePrice"/>
  </message>
```

Figure 5.4: *StockQuote* version 2 description (first part)

```
  <portType name="StockQuotePortType">
    <operation name="GetLastTradePrice">
      <input message="tns:GetLastTradePriceInput" />
      <output message="tns:GetLastTradePriceOutput" />
    </operation>
    <operation name="GetBestOffer">
      <input message="tns:GetBestOfferInput" />
      <output message="tns:GetBestOfferOutput" />
    </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>
    <operation name="GetBestOffer">
      <soap:operation soapAction="http://ex.com/GetBestOffer" />
      <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>
```

Figure 5.5: *StockQuote* version 2 description (second part)

The first change introduces the operation $v_{GetBestOffer,1}$ along with its associated types $v_{GetBestOfferOutput,1}$, $v_{GetBestOfferInput,1}$, $v_{BestOffer,1}$ and $v_{StatusType,1}$, which have their description depicted separately in Figure 5.6. The second change modifies the primitive type associated with *TradePrice*, which is changed from *float* to *double* as illustrated in Figure 5.7.

Thus, in order to incorporate the second version of the *StockQuote* service, the *WSDL/ Features Converter* transforms this new version to the feature representation, which results in the graph of Figure 5.8. In this process, the change in *TradePrice* description is identified together with all the new features and the affected ones.

So a new version of *TradePrice* type ($v_{TradePrice,2}$) is created in the *Version Evolution Repository* (*versioning case 2*), and associated with this feature. Because of the ripple effect, the features *GetLastTradePriceOutput*, *GetLastTradePrice* and *StockQuote* are affected, and hence equally versioned (*versioning case 3(a)*). 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*).

Notice that the features $v_{GetLastTradePriceInput,1}$ and $v_{TradePriceRequest,1}$ are not versioned with $v_{TradePrice,2}$, but rather have their versions *reused* (*versioning case 3(b)*). As these features are unchanged (Figure 5.8), i.e. neither changed nor affected, the features that depend on these have their dependency relationships updated, in order to reference to the ones already existent in the *Version Evolution Repository*, for instance the feature *GetLastTradePrice, 2*. The resulting state of the repository after adding $v_{StockQuote,2}$ is depicted in Figure 5.3b.

```
<operation name="GetBestOffer">
  <input message="tns:GetBestOfferInput"/>
  <output message="tns:GetBestOfferOutput"/>
  <soap:operation soapAction="http://ex.com/GetBestOfferPrice"/>
  <input><soap:body use="literal"/></input>
  <output><soap:body use="literal"/></output>
</operation>
```

(a) Description associated with *GetBestOffer* operation

```
<message name="GetBestOfferInput">
  <part name="body" element="xsd1:BestOffer"/>
</message>
```

(a) Description associated with *BestOfferInput* type

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

(a) Description associated with *BestOfferOutput* type

```
<element name="BestOffer">
  <complexType>
    <sequence>
      <element name="tickerSymbol" type="string"/>
      <element name="status" type="StatusType"/>
    </sequence>
  </complexType>
</element>
```

(a) Description associated with *BestOffer* type

```
<simpleType name="StatusType">
  <restriction base="xs:token">
    <enumeration value="Active"/>
    <enumeration value="Expired"/>
  </restriction>
</simpleType>
```

(a) Description associated with *StatusType* type

Figure 5.6: New features' description introduced in *StockQuote* version 2

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

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

Figure 5.7: Example of description change

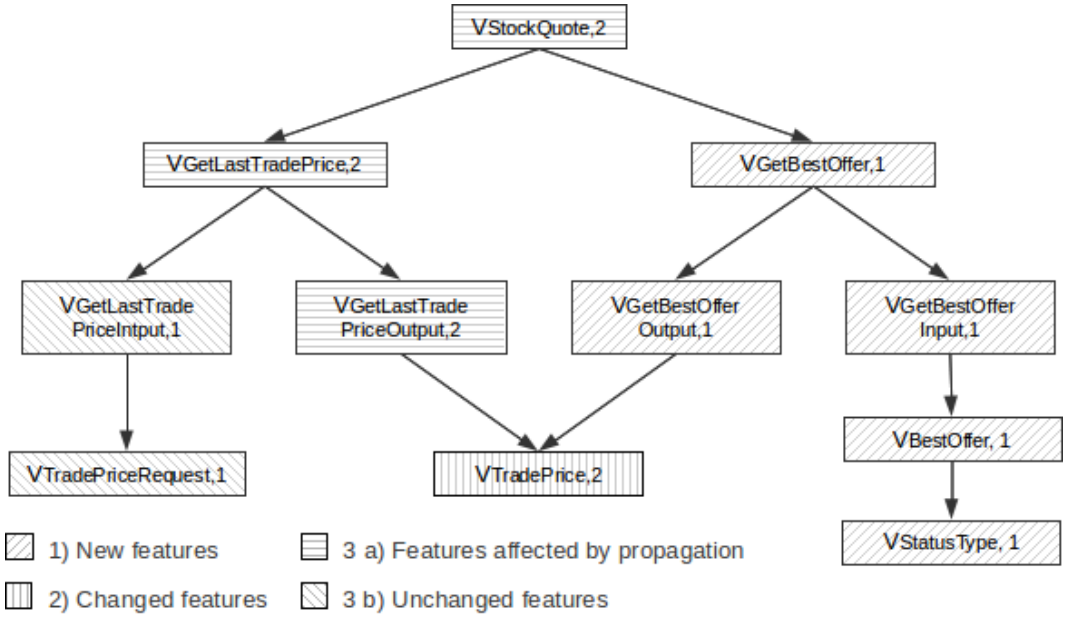


Figure 5.8: Comparing $v_{StockQuote,2}$ with the repository

5.5 Compatibility Analyzer

The process described in the previous section extracts from an existing standard WSDL/ XSD service description an internal representation of features, 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 for instance, a service with regard to its operations or 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 work 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 proposed in (BECKER et al., 2008), which assesses compatibility on object-oriented service descriptions. We have adapted this algorithm to address the compatibility checking on smaller fragments of the WSDL interface description, as represented by our feature-oriented versioning model.

As mentioned in Chapter 3, most work considers a very restricted set of compatible change operations (BROWN; ELLIS, 2004; FANG et al., 2007; ANDRIKOPOULOS; BENBERNOU; PAPAZOGLOU, 2011), which is summarized in a) addition of new operations, and b) addition of new types that are not contained within previously existing types. These are also the only compatible changes considered in this work. Table 5.1 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 5.1 is considered incompatible.

Table 5.1: Change cases for compatibility assessment

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	Operation	Change in description	*Incompatible*
5	Update	Service	Change in description	*Incompatible*
6	Update	Type	Change in description due to order, cardinality or type	*Incompatible*
7	Remove	Operation	Remove operation dependency	*Incompatible*
8	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 5.1, and to establish the compatibility verdict between them, such as the relation between versions depicted in Figure 4.2. The algorithm receives two feature versions as input, $v_{feature,p}$ and $v_{feature,q}$. We assume that both versions relate to the same feature (i.e. have the same name).

Then, the algorithm assesses the compatibility of the latter with regard to the former as the following procedure:

1. the algorithm verifies if dependencies of features present in $v_{feature,p}$ have not been removed from $v_{feature,q}$ (cases 7 and 8 of Table 5.1);

2. compares the description fragment associated with the compared versions (cases 4, 5 and 6);

3. recursively evaluates the compatibility of all corresponding dependent feature versions (cases 1, 2 and 3) , and then;

4. sets the compatibility relationship together with the respective verdict.

The version graph rooted in $v_{feature,q}$, is traversed in a depth-first manner, which enables the propagation of detected incompatibilities to dependent versions. The pseudo-algorithm is presented in Listing 1.

Initially, the algorithm (line 2) evaluates whether feature dependencies were removed from $v_{feature,q}$ compared to $v_{feature,p}$. 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, the algorithm analyzes the textual description associated with versions $v_{feature,q}$ and $v_{feature,p}$, as described in Listing 2.

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:  for all v_dep,Qj ∈ setOfDependencies(v_feature,q) do
5:     // If there is a dependency feature version with the same name but different version
6:     if exists v_dep,Pi ∈ setOfDependencies(v_feature,p) ∧ (depP = depQ) ∧ (i ≠ j) then
7:        // Verify recursively if these two versions of a same feature are in turn compatible
8:        compat ← compat ∧ compatibilityAssessment(v_depP,i, v_depQ,j);
9:     end if
10: end for
11: setVerdict(v_feature,q, v_feature,p, compat);
12: return compat;
```

Then, the algorithm traverses all features upon which $v_{feature,q}$ depends on in order to assess their compatibility against the corresponding ones in the dependency set of $v_{feature,p}$. Hence, 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). Thus, if $v_{feature,q}$ depends on a feature that $v_{feature,p}$ also depends on but not on the same version of that feature, the algorithm is called in order to assess the compatibility between these versions.

If any dependency is incompatible, then the algorithm updates the verdict to incompatible due to the ripple effect. Also, if there is a dependent version of $v_{feature,q}$ that does not exist in the set of dependencies of $v_{feature,p}$, then this situation is related to the compatible cases 1 and 2 of Table 5.1. Finally, the algorithm returns the compatibility assessment result of $v_{feature,q}$ regarding $v_{feature,p}$.

Notice that the algorithm could stop at any point where an incompatibility is detected, but we have opted by continuing the assessment as a placeholder for later documenting all incompatible changes found, as well as for ensuring that each feature will be assessed. We are aware that stopping the algorithm at the point where an incompatibility is found can improve its performance. However, we plan for studying less restrictive compatibility in immediately future work, in which the verdict of every feature can be analyzed against usage.

Function *evaluateDescription* (line 3 in Listing 1), detailed in Listing 2, aims at verifying the cases of Table 5.1 that concerns the analysis of the description fragments, namely cases 4, 5 and 6. 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 1), assuming incompatibility, except in the case of *type* features, when a more detailed examination of the description field is performed (lines 2-15). 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, as proposed in (BECKER et al., 2008; ANDRIKOPOULOS; BENBERNOU; PAPAZOGLOU, 2011).

In order to extract the description elements and their properties from the WSDL fragments of features' versions, we use the function *setOfElements* that 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 4), which leads to the verification of case 3 of Table 5.1. Next, if the element exists within the two versions we compare their prop-

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

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

erties (line 6) in order to verify if they have changed in *order*, *type* or *cardinality*. The comparison of element properties refers to case 4 of Table 5.1. Removed description elements are verified in lines 10 to 14. Finally, the *evaluateDescription* function returns the compatibility verdict for $v_{feature,q}$ regarding $v_{feature,p}$.

5.6 Compatibility Assessment Illustration

To illustrate our algorithm procedure, suppose *StockQuote* versions $v_{StockQuote,1}$ and $v_{StockQuote,2}$ as the algorithm input parameters. In doing so, the algorithm verifies the compatibility of the latter regarding the former by analyzing the dependency graphs of each version. We outline the compatibility assessment of versions $v_{StockQuote,1}$ and $v_{StockQuote,2}$ in Table 5.2, along with the sequence of recursive algorithm calls. As a means of reference, the graph of $v_{StockQuote,2}$ is represented by Figure 5.8, whereas the graph of $v_{StockQuote,1}$ is represented by Figure 5.3a.

The algorithm starts by assessing compatibility of version $v_{StockQuote,2}$ regarding $v_{StockQuote,1}$ (sequence 1 in Table 5.2). Following the compatibility algorithm, the first step (Listing1: line 2) aims to verify if version $v_{StockQuote,2}$ have any removed dependency regarding $v_{StockQuote,1}$. As shown in the comparison of Figure 5.8, $v_{StockQuote,2}$ has no removed dependency. So, the algorithm continues by evaluating the versions description (Listing1: line 3), which has not changed. Therefore, the algorithm continues and recursively assesses compatibility on the features' versions that constitute the set of dependencies of $v_{StockQuote,2}$ (Listing1: lines 4-10), namely $v_{GetLastTradePrice,2}$ and $v_{GetBestOffer,1}$.

Table 5.2: Compatibility assessment outline for $v_{StockQuote,2}$ regarding $v_{StockQuote,1}$

Sequence	Feature Version	Removed Dependency Evaluation	Description Evaluation	Affected by Propagation	Compatibility Verdict
1	$v_{StockQuote,2}$	Compatible	Compatible	Yes	***Incompatible***
2	$v_{GetLastTradePrice,2}$	Compatible	Compatible	Yes	***Incompatible***
3	$v_{GetLastTradePriceOutput,2}$	Compatible	Compatible	Yes	***Incompatible***
4	$v_{TradePrice,2}$	Compatible	Incompatible	No	***Incompatible***

The versions to consider for compatibility are those that exist in $v_{StockQuote,1}$ with different version number from those in $v_{StockQuote,2}$ (Listing1: Line 6). Thus, $v_{GetLastTradePrice,2}$ is recursively assessed by the algorithm, whereas $v_{GetBestOffer,1}$ is not because it has no early version to compare with. The same occurs to $v_{GetBestOffer,1}$ dependencies, which are both not assessed.

Thus, we apply the same procedure of compatibility assessment over $v_{GetLastTradePrice,2}$ with regard to $v_{GetLastTradePrice,1}$ (sequence 2 in Table 5.2). As presented in Figure 5.8, the version $v_{GetLastTradePrice,2}$ has no removed dependencies and maintains the same description as its previous version. So, the algorithm continues in order to verify $v_{GetLastTradePrice,2}$ dependencies, which led to the assessment of $v_{GetLastTradePriceOutput,2}$ (sequence 3 in Table 5.2). Thus, the same procedure occurs to $v_{GetLastTradePriceOutput,2}$ with regard to $v_{GetLastTradePriceOutput,1}$. Feature $v_{GetLastTradePriceInput,1}$ is not compared because it maintains the same version.

Next feature version to be analyzed is $v_{TradePrice,2}$ (sequence 4 in Table 5.2), which has no dependencies and thereby no removed ones. However its description is evaluated following Listing 2. The function *evaluateDescription* verifies that $v_{TradePrice,2}$ has its description changed from $v_{TradePrice,1}$ (Listing 2: line 2). Although $v_{TradePrice,2}$ has no addition of elements (Listing 2: line 5), the element *price* has its datatype changed

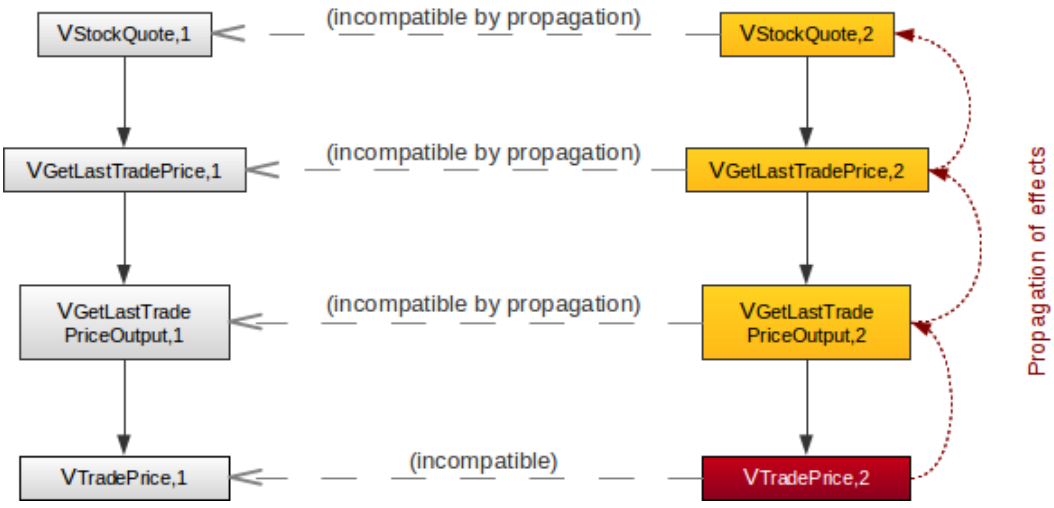


Figure 5.9: Incompatibility verdict propagation of $v_{StockQuote,2}$ regarding $v_{StockQuote,1}$

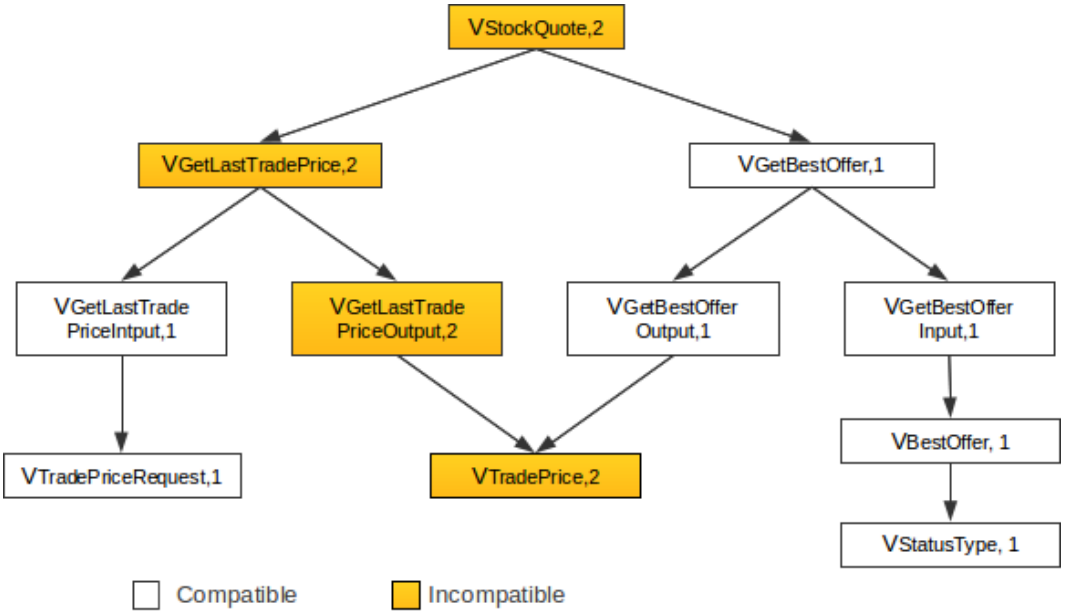


Figure 5.10: Compatibility algorithm result

from *float* to *double* (Listing 2 line 7), as presented in Figure 5.7. Therefore, the feature version $v_{TradePrice,2}$ is assessed ***incompatible*** by the *evaluateDescription* function (Listing 2: line 8) and has thereby its verdict set in line 11 of Listing 1. The verdict is then, propagated (Listing 1: line 8) through the features' versions $v_{GetLastTradePriceOutput,2}$, $v_{GetLastTradePrice,2}$ and $v_{StockQuote,2}$. The propagation effect caused by the incompatibility in $v_{TradePrice,2}$ version is depicted in Figure 5.9.

Finally, the resulting verdict of $v_{StockQuote,2}$ regarding $v_{StockQuote,1}$ is ***incompatible*** and all the versions in the graph are annotated with its correspondent verdict, as shown in Figure 5.10. We summarize the compatibility assessment procedure of $v_{StockQuote,2}$ regarding $v_{StockQuote,1}$ and the sequence of the recursively algorithm calls in Table 5.2.

5.7 Concluding Remarks

In this chapter, we presented the Version Manager, a component of the Change Management Framework that aims to version services and assess their compatibility in a finer grain perspective. We presented the procedure for building the dependency graph of features with regard a service version. Then we presented the manner we use for version only the features that have changed and the ones affected by a change. The graphs are stored in the Version Evolution Repository, which maintains the features versions, together with their dependencies. We also presented the algorithm for assessing compatibility of features versions, which enables the analysis of compatibility on different points of a service with regard its dependencies.

6 EXPERIMENTAL RESULTS

In this chapter, we present an experiment with regard the versioning model and the compatibility assessment algorithm. The main objective is to present the usefulness of our approach by identifying and qualifying the changes occurred on a real case scenario. Hence, we start the chapter by presenting the chosen scenario. Then, we present the experiment by the perspectives of change identification and change qualification. Finally, we present the conclusion for the experiment.

6.1 Experiments

To develop experiments with the versioning model and the compatibility assessment algorithm, we built a prototype that follows the architecture displayed at Figure 5.1. As we mentioned, the *WSDL/Feature Converter* module extracts from a new service version (WSDL description) the internal versioning model as described in Section 5.2, and the *Compatibility Analyzer* module implements the compatibility assessment algorithm detailed in Section 5.5. The *Version Evolution Repository* persists the resulting features and versions in an XML file, according to schema presented in in Figure 5.2. The prototype was developed using Java [1], DOM Parser and JGraphT [2] library for handling the versions graph. The prototype generates a report listing the versioned features and their compatibility according to the rules of Table 5.1.

For this experiment we chose eBay *Trading* service [3], because of the great number of available versions and the frequency on introducing new ones. 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 [4] that reports explicit points of changes with regard to the previous version. However, there is barely any information on how these changes affect other parts of the service. The manual analysis of the propagation effects of a change is a hard task since the interface document is huge, for instance, the most recent versions of *Trading* service have almost 130.000 lines of description. Thus, client developers are responsible for detecting *whether* and *how* changes affect their applications in order to stay in sync with service evolution.

In order to reduce the impact of changes on clients, eBay establishes an evolution policy that assumes as compatible the case 3 of Table 5.1. To comply with this policy, developers need to build client applications that handle unrecognized data (e.g. non expected additional output arguments), a widely adopted practice of which the advantages

[1] `http://www.java.com/`
[2] `http://jgrapht.org/`
[3] `https://www.x.com/developers/ebay/products/trading-api`
[4] `http://developer.ebay.com/devzone/xml/docs/WebHelp/ReleaseNotes.html`

are not a consensus (e.g. it prevents benefiting from strong type checking). In order to adjust to this policy, we adapted our algorithm for considering both the core rules of Table 5.1, and eBay compatibility rules.

We have developed two experiments examining 45 versions of eBay *Trading* service, which correspond to almost two years of the eBay *Trading* versions. We use 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 the compatibility of each feature changed with regard to the corresponding ones in the immediately previous version. Our goal is to analyze structural changes and thereby we removed all semantic information (e.g. documentation tags) from the service interface documents.

We were not able to compare our approach with existing works that automatically detects changes and assess compatibility due to the following reasons: a) the algorithm we extended (BECKER et al., 2008) assumes an object-oriented description of service, which is 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; 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.

6.2 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. eBay identifies versions using odd numbers, and because version *653* is the baseline, the graph starts at version *655*.

The result of this experiment is presented in Figures 6.1, 6.2, and 6.3. Figure 6.1 shows the relation between the changed and affected features for each version. Figure 6.2 depicts for each version the number of changed features regarding their type. Figure 6.3 presents the number of the affected features per version and their type.

We verified that all changed features made explicit due to our versioning model are described in the release notes, but almost none of the affected features are mentioned. Explicitly changed features (Figure 6.1) correspond in total to less than 5% of the 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 [5]. 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 observed that 99% of explicit changes are done to types (Figure 6.2), whereas the propagation effect reaches an average of 26% for operations and 74% for types (Figure 6.3). 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

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

operation. As expected, the service is always affected except for the case of a version that has no feature changed. Note that for each version in Figure 6.2 that has at least one changed feature, there is an affected feature of type *service* in Figure 6.3. Versions that have no changed, thus no affected, features are justified by having changes on semantic information, which as mentioned is out of the scope of this work but can be further addressed.

This experiment confirms that current information provided by release notes is insufficient for client developers to detect which changes affect them. On the other hand, our approach enabled the identification of the affected features not addressed in the release notes, which correspond to 95% of the overall changed/affected features. Thus, we conclude that the proposed versioning model supported the efficient identification and quantification of changes impact.

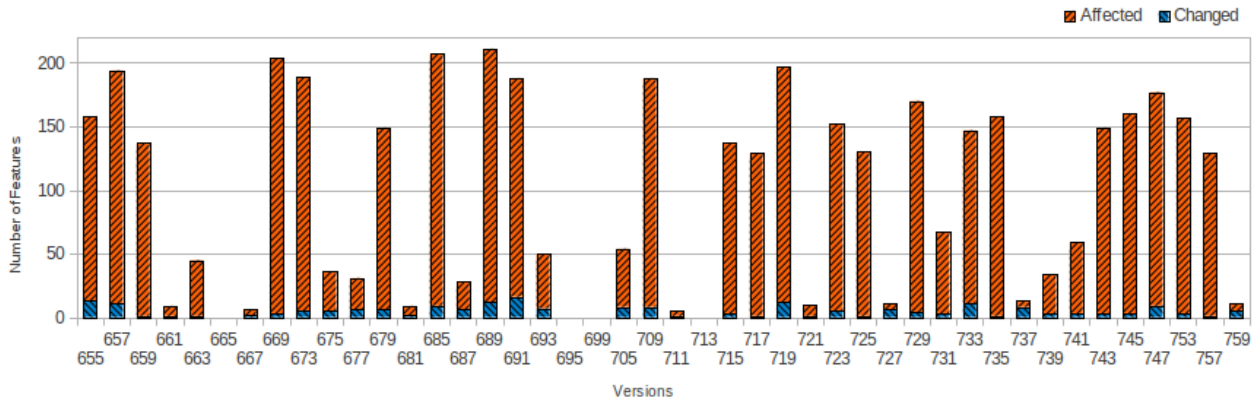


Figure 6.1: Total of features changed/affected per version

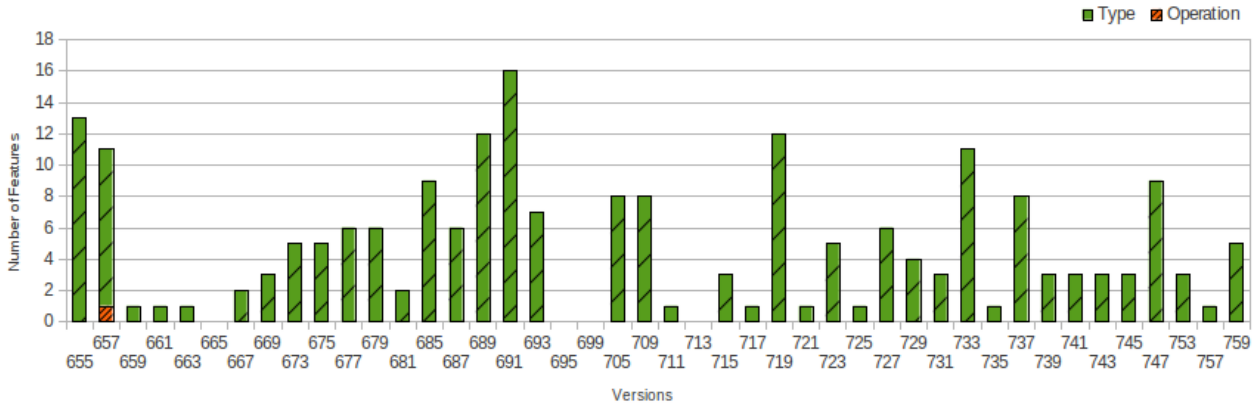


Figure 6.2: Features changed in description

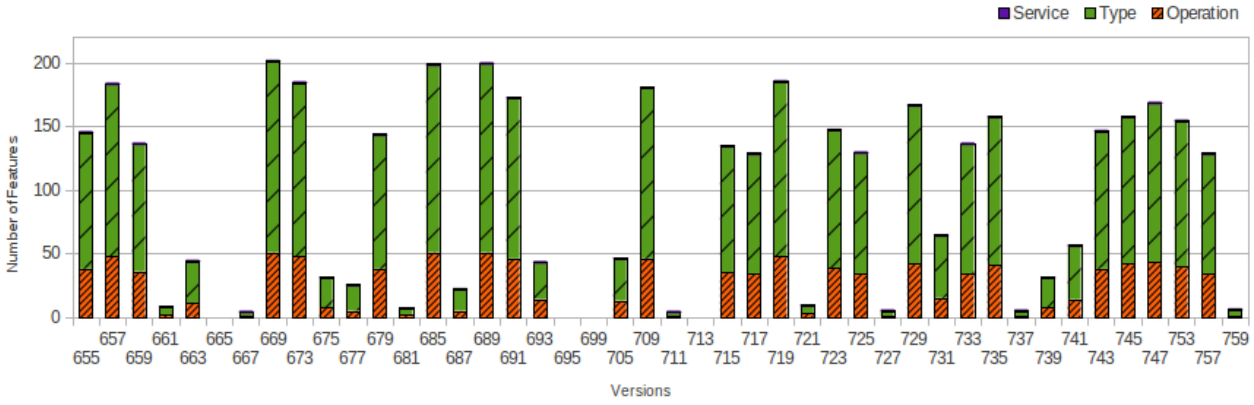


Figure 6.3: Features affected by changes

6.3 Experiment 2 : Qualifying Changes

In this experiment, we analyzed at feature level the compatibility of each service version with regard the preceding one using our compatibility assessment algorithm. We used both the core rules of Table 5.1 and eBay policy compatibility. The results are presented in Figure 6.4, which is divided into two graphs due to improve visualization. The second graph in Figure 6.4 continues the first by comparing version 711 to 709. According to the compatibility rules we adopt (Table 5.1), none of the service versions are backward compatible with regard to the previous one. This result differs from the information available in the release notes for these versions that report only 6 out of the 45 as backward incompatible, namely versions 685, 689, 717, 719, 739 and 757. 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 5.1, which is reported as incompatible in related literature (BROWN; ELLIS, 2004; FANG et al., 2007; ANDRIKOPOULOS; BENBERNOU; PAPAZOGLOU, 2011; BECKER et al., 2008).

We observe in Fig. 6.4 that eBay minimizes the impact of changes on their clients based on their evolution policy. Nevertheless, it is possible to observe that *Trading* service undergoes disruptive changes from time to time (6 versions out of 45). According to more strict compatibility rules, clients that do not comply with their evolution policy may be affected in every change, depending whether or not they use the incompatibly changed features.

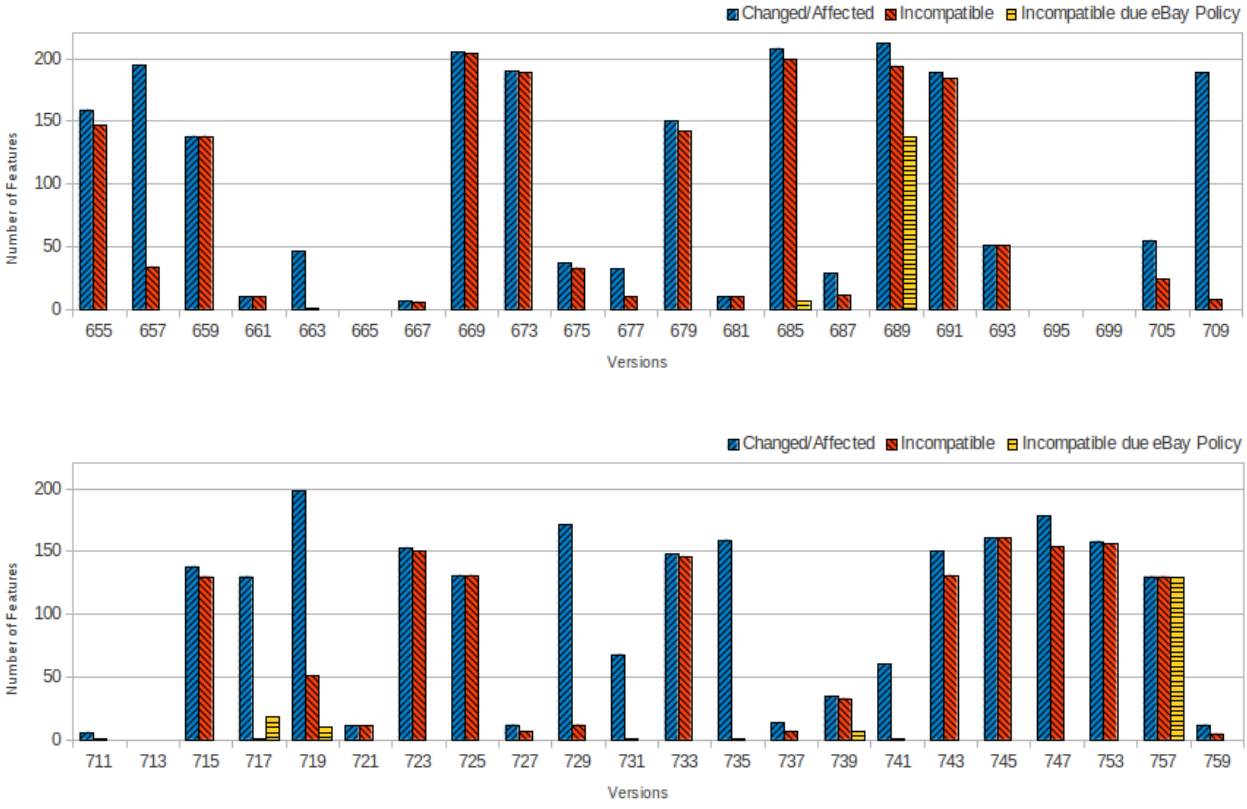


Figure 6.4: Trading Service Compatibility Analysis

6.4 Concluding Remarks

We conclude that the proposed versioning model and compatible assessment algorithm supported the efficient identification and qualification of changes and their impact. Our algorithm was flexible enough for the easy implementation of different compatibility rules. This experiment indicates the usefulness of a finer grain of versioning for locating and assessing compatibility, and 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, and develop mechanisms that support clients to understand how changes affect them, and cope with them.

7 CONCLUSIONS

In this work, we presented a service versioning approach and a compatibility assessment algorithm in a finer-grain, i.e. at feature level, which enabled the scope, quantification and assessment of changes in a finer-grained perspective. The feature-oriented model allowed the representation of an entire service description in a dependency graph of features. By the proposed feature mapping, every feature could be related with a fragment of the service description and incorporated to the feature representation without loosing any information, despite the semantic documentation not addressed in this work. This mapping is compatible with the W3C current standard for service description and although been developed regarding the WSDL versions 1.x, it can be easily adapted to more recent specifications.

With the feature-oriented versioning approach, we presented a manner to version only the features that have changed or been affected by a change, which has as straightforward consequence the easy identification of changes. With regard versioning, the contribution of this work is a version model that covers the lack of a finer-grain perspective as well as providing a mechanism to trace changes during evolution. As depicted in Table 7.1, our versioning model addresses a finer-grain perspective, which is also addressed in (BECKER et al., 2008), but in this work it is aligned with W3C current standards.

With regard to compatibility, this work presents an assessment framework to automatically assess the compatibility verdict of changes. The compatibility assessment algorithm enabled the analysis of change impact on different points of a service, as well

Table 7.1: Comparison between this work and the related ones with regard versioning

Approaches	**Best practices**	**Versioning Model**	**Finer-grain perspective**	**Cope with W3C standards**
(BROWN; ELLIS, 2004)	guidelines	–	–	Yes
(ENDREI et al., 2006)	guidelines	–	–	Yes
(FANG et al., 2007)	–	adapt current standards	–	Yes
(BECKER et al., 2008)	–	framework	type level	No
(LEITNER et al., 2008)	–	framework	–	Yes
This work	–	**framework**	**feature level**	**Yes**

Note: We use hyphen (–) for works that does not address the topic in each column

Table 7.2: Comparison between this work and the related ones with regard compatibility

Approaches	**Change cases**	**Assessment framework**	**Finer-grain perspective**	**Cope with W3C standards**
(PONNEKANTI; FOX, 2004)	guidelines	usage-oriented	identification	Yes
(BROWN; ELLIS, 2004)	guidelines	–	–	Yes
(ENDREI et al., 2006)	guidelines	–	–	Yes
(FANG et al., 2007)	summarization	–	–	Yes
(BECKER et al., 2008)	relaxing cases	automatic	assessment	No
(ANDRIKOPOULOS; BENBERNOU; PAPAZOGLOU, 2011)	relaxing cases	formal	identification	Yes
(YAMASHITA; BECKER; GALANTE, 2011a)	–	automatic/ usage-oriented	identification/ assessment	Yes
This work	–	**assessment**	**identification/ assessment**	**Yes**

Note: We use hyphen (–) for works that does not address the topic in each column

as the incompatibility ripple effect within the feature level proposal. We adopted very conservative rules for compatibility assessment, but have shown that the algorithm can be extended for different rules. The proposed compatibility algorithm enabled the automatic assessment of changes in a finer-grain, which is also proposed in other works, such as the ones depicted in Table 7.2. However, this work differs from those by fitting W3C specifications in case of (BECKER et al., 2008), automatically assessing compatibility in case of (ANDRIKOPOULOS; BENBERNOU; PAPAZOGLOU, 2011), considering relaxed compatibility cases that can address different profiles of clients rather than a particular client in (PONNEKANTI; FOX, 2004) and finally, enabling the usage-oriented approach sketched in (YAMASHITA; BECKER; GALANTE, 2011a).

With regard the usefulness of our approach, we have experimented the versioning and compatibility assessment using a real service. In this experiment, we verified that our approach supported the efficient identification and qualification of changes together with their impact and concluded that our algorithm can be easily adapted to support compatibility relaxed cases, such as eBay evolution policy or relaxed cases found in literature. As commented, service stakeholders lack proper mechanisms to easily recognize changes and their impact on evolving services. Hence, by identifying and qualifying changes at feature level, the approach in this work can provide invaluable information for service stakeholders in order to help them cope with changes during service evolution.

As future work, we plain to improve the compatibility assessment algorithm for the purpose of documenting the detected incompatibile changes. In doing so, the algorithm can provide information on *how* the feature changed rather than only *what* have changed, resulting in a more detailed tracing mechanism. We also plan to aggregate

less restrictive cases of compatibility to the algorithm, such as the T-shaped changes in (ANDRIKOPOULOS; BENBERNOU; PAPAZOGLOU, 2011) and the in/output restriction in (BECKER et al., 2008). Finally, we plan for studying change cases involving the semantics information present in the service interface description, such as the documentation tags that we opted not to address in this work.

The result of this work supports various applications. As mentioned, this work is a component of a framework for measuring change impact according to usage. Hence, this work provides the fundamentals for achieving the usage-oriented compatibility proposed in (YAMASHITA; BECKER; GALANTE, 2011a). Moreover, our approach can contribute to applications, such as SOA ripple effect quantification (WANG; CAPRETZ, 2009), customized release notes (ZOU et al., 2008), the reduction of provisioned versions (FRANK et al., 2008), and enables load balance management among implemented versions, which precedes the finer-grain deployment in (TREIBER; ANDRIKOPOULOS; DUSTDAR, 2009).

REFERENCES

ANDRIKOPOULOS, V.; BENBERNOU, S.; PAPAZOGLOU, M. P. Managing the Evolution of Service Specifications. In: ADVANCED INFORMATION SYSTEMS ENGINEERING, 20., Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2008. p.359–374. (CAiSE '08).

ANDRIKOPOULOS, V.; BENBERNOU, S.; PAPAZOGLOU, M. P. Evolving Services from a Contractual Perspective. In: INTERNATIONAL CONFERENCE ON ADVANCED INFORMATION SYSTEMS ENGINEERING, 21., Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2009. p.290–304. (CAiSE '09).

ANDRIKOPOULOS, V.; BENBERNOU, S.; PAPAZOGLOU, M. P. On the Evolution of Services. **IEEE Transactions on Software Engineering**, Los Alamitos, USA, 2011. in press - early access articles.

BACHMANN, R. **Challenges of web service change management**. Available at: <http://www.sdn.sap.com/irj/scn/index?rid=/library/uuid/4e1d4d29-0801-0010-159b-f8d51a04bbbd/>. Access in: May 2012.

BECKER, K. et al. Automatically Determining Compatibility of Evolving Services. In: IEEE INTERNATIONAL CONFERENCE ON WEB SERVICES, 2008., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2008. p.161–168. (ICWS '08).

BROWN, K.; ELLIS, M. **Best Practices for Web services Versioning**. Available at: <http://www.ibm.com/developerworks/webservices/library/ws-version/>. Access in: May 2012.

CHRISTENSEN, E. et al. **Web Services Description Language (WSDL) 1.1**. Available at: <http://www.w3.org/TR/wsdl>. Access in: Jun 2012.

ENDREI, M. et al. **Moving forward with web services backward compatibility**. Available at: <http://www.ibm.com/developerworks/java/library/ws-soa-backcomp/>. Access in: May 2012.

FANG, R. et al. A Version-aware Approach for Web Service Directory. In: IEEE INTERNATIONAL CONFERENCE ON WEB SERVICES, Salt Lake City, USA. **Proceedings...** IEEE Computer Society, 2007. p.406–413.

FOKAEFS, M. et al. An Empirical Study on Web Service Evolution. In: IEEE INTERNATIONAL CONFERENCE ON WEB SERVICES, 2011., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2011. p.49–56. (ICWS '11).

FRANK, D. et al. Using an Interface Proxy to Host Versioned Web Services. In: IEEE INTERNATIONAL CONFERENCE ON SERVICES COMPUTING - VOLUME 2, 2008., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2008. p.325–332. (SCC '08).

LEE, K.; KANG, K. C.; LEE, J. Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. In: SOFTWARE REUSE: METHODS, TECHNIQUES, AND TOOLS: PROCEEDINGS OF THE SEVENTH REUSE CONFERENCE (ICSR7. **Anais...** Springer-Verlag, 2002. p.62–77.

LEITNER, P. et al. End-to-End Versioning Support for Web Services. In: IEEE INTERNATIONAL CONFERENCE ON SERVICES COMPUTING - VOLUME 1, 2008., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2008. p.59–66. (SCC '08).

PAPAZOGLOU, M. P. The Challenges of Service Evolution. In: ADVANCED INFORMATION SYSTEMS ENGINEERING, 20., Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2008. p.1–15. (CAiSE '08).

PONNEKANTI, S. R.; FOX, A. Interoperability among independently evolving web services. In: ACM/IFIP/USENIX INTERNATIONAL CONFERENCE ON MIDDLEWARE, 5., New York, NY, USA. **Proceedings...** Springer-Verlag New York: Inc., 2004. p.331–351. (Middleware '04).

SILVA, E. et al. A Business Intelligence Approach to Support Decision Making in Service Evolution Management. In: IEEE NINTH INTERNATIONAL CONFERENCE ON SERVICES COMPUTING, 2012., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2012. p.41–48. (SCC '12).

TREIBER, M.; ANDRIKOPOULOS, V.; DUSTDAR, S. Calculating service fitness in service networks. In: SERVICE-ORIENTED COMPUTING, 2009., Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2009. p.283–292. (ICSOC/ServiceWave'09).

WANG, S.; CAPRETZ, M. A. M. A Dependency Impact Analysis Model for Web Services Evolution. In: IEEE INTERNATIONAL CONFERENCE ON WEB SERVICES, 2009., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2009. p.359–365. (ICWS '09).

YAMASHITA, M.; BECKER, K.; GALANTE, R. Service Evolution Management Based on Usage Profile. In: IEEE INTERNATIONAL CONFERENCE ON WEB SERVICES, 2011., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2011. p.746–747. (ICWS '11).

YAMASHITA, M.; BECKER, K.; GALANTE, R. A Flexible Approach for Assessing Service Compatibility at Element Level. In: BRAZILIAN SYMPOSIUM ON DATABASES (SBBD)), Florianópolis - SC, Brazil. **Anais...** Porto Alegre: Sociedade Brasileira de Computação (SBC), 2011. p.8.

YAMASHITA, M.; BECKER, K.; GALANTE, R. A Feature-based Versioning Approach for Assessing Service Compatibility. **JIDM**, Porto Alegre, Brasil, v.3, n.2, p.120–131, 2012.

YAMASHITA, M. et al. Measuring Change Impact based on Usage Profiles. In: IEEE INTERNATIONAL CONFERENCE ON WEB SERVICES, Honolulu, Hawaii. **Proceedings...** IEEE, 2012. p.226 – 233.

ZOU, Z. L. et al. On Synchronizing with Web Service Evolution. In: IEEE INTERNATIONAL CONFERENCE ON WEB SERVICES, 2008., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2008. p.329–336. (ICWS '08).

APPENDIX A VERSIONAMENTO E COMPATIBILIDADE DE SERVIÇO EM NÍVEL DE FEATURE

A.1 Introdução

A Arquitetura Orientada a Serviço (SOA) denota uma abordagem arquitetural que permite a criação de sistemas de baixo acoplamento apoiado sobre componentes autônomos, chamados serviços. O baixo acoplamento é suportado pela existência de uma interface bem definida, que expõe as características do serviço relevantes ao consumo. Da perspectiva do SOA, um serviço é um conjunto de funcionalidades expostas por um provedor as quais consumidores podem vincular suas aplicações. O ciclo de vida dos componentes em SOA encoraja o desenvolvimento de serviços autônomos e permite independência durante as fases de desenvolvimento, implantação e manutenção. Porém, serviços não escapam da necessidade de lidar com mudanças. Para se alinhar à novas oportunidades de negócio, serviços estão sujeitos a constantes variações, o que requer estratégias para lidar com múltiplas versões durante seu ciclo de vida.

A gerência da evolução de serviços engloba a criação, manutenção e desativação de diferentes versões de serviços no ambiente provedor (PAPAZOGLOU, 2008), o que frequentemente leva à manutenção de inúmeras versões concorrentes. No âmbito deste trabalho, entendemos uma versão de serviço por uma versão da descrição da interface que o serviço expõe.

A descrição da interface de um serviço é um contrato unilateral estabelecido pelo provedor, que guia os consumidores no modo como acessar as funcionalidades do serviço. Porém, as atuais notações para descrição de interface de serviço, incluindo o padrão WSDL/XSD, não trata adequadamente o versionamento (ANDRIKOPOULOS; BENBERNOU; PAPAZOGLOU, 2011). Usualmente, apesar de muitas características do serviço permanecerem inalteradas (ex. tipos, operações, mensagens), a descrição inteira do serviço é versionada. Este procedimento gera dificuldades no reconhecimento e medição to real impacto da mudança. Na ausência de um suporte adequado, provedores de serviço (ex. eBay, Google, Amazon) publicam novas versões em conjunto com notas de lançamento na esperança de ajudar os consumidores a se ajustarem às mudanças. As notas de lançamento descrevem as mudanças explícitas, ou seja, mudanças pontuais em determinadas características do serviço, mas falham em descrever o impacto dessas mudanças em suas dependências (outras características do serviço) e por fim, a retrocompatibilidade de características específicas.

Em suma, às partes envolvidas na evolução do serviço, faltam mecanismos adequados para o fácil reconhecimento de mudanças e do impacto dessas na evolução de serviços. Neste trabalho, abordamos essa deficiência propondo o versionamento em uma perspec-

tiva mais granular, ao qual nos referimos como nível de *feature*, e o algoritmo de avaliação de compatibilidade para qualificar automaticamente as *features* versionadas com um veredito de compatibilidade.

Assim, o objetivo deste trabalho é uma abordagem para versionar *features* separadamente, manter seus relacionamentos e identificar as *features* afetadas direta ou indiretamente. Essa abordagem também permite a avaliação de compatibilidade entre duas versões de *feature* de maneira que se possa verificar se a mudança pode afetar qualquer integração externa do serviço. Como resultado, este trabalho permite a quantificação e a qualificação de mudanças no serviço de modo mais granular e assim, provê informações de grande valor às partes que lidam com serviços com o propósito de ajudá-los a lidar com as mudanças decorrentes da evolução de serviços.

A.2 Conceitos

Existem dois conceitos de suma importância para o entendimento deste trabalho: a versão de serviços e sua compatibilidade. Neste trabalho, utilizamos o termo versão de serviço como um sinônimo de versão de interface de serviço. Uma interface de serviço é um documento que descreve as funcionalidades de um serviço, sendo sua interface a representação de um de seus estados ao longo do seu ciclo de vida. Neste trabalho, nós estudamos serviços e suas versões descritas na linguagem WSDL.

WSDL fornece um modelo, com base no formato XML, que descreve serviços em duas seções fundamentais que englobam as funcionalidades abstratas e os detalhes concretos de serviços. Na seção abstrata, o serviço é descrito em termos das trocas de mensagens que o serviço pode receber ou responder. Mensagens são expressas por elementos tipo, que são descritos usando um sistema de tipagem independente, tipicamente o XML Schema. Operações associam os parâmetros de suas mensagens de entrada e saída, previamente descritas. Por fim, um *portType* descreve a funcionalidade do serviço definindo as operações que podem ser realizadas, bem como as mensagens necessárias para realizar as operações. Na seção concreta, um serviço é definido como uma coleção de portas de rede que implementam um *portType*. Uma porta associa um endereço de rede através de um vínculo, que descreve os detalhes de formato de transporte para um ou mais *portTypes* (CHRISTENSEN et al., 2001). Os componentes do modelo WSDL são apresentados na Figura 2.1.

A compatibilidade de versões de serviços é uma garantia tal que, ao se introduzir uma nova versão de serviço, as partes envolvidas na evolução não sejam afetadas (PAPAZOGLOU, 2008). A compatibilidade pode assumir diferentes significados dependendo do contexto ou perspectiva pela qual é vista. Neste trabalho, abordamos a retrocompatibilidade.

A retrocompatibilidade diz respeito à como mudanças nas versões do serviço afetam seus consumidores (BECKER et al., 2008; FANG et al., 2007; ENDREI et al., 2006). Assim, assegurar a retrocompatibilidade significa que um serviço em evolução deve continuar suportando clientes antigos a medida em que haja mudanças ao longo do tempo. Em outras palavras, a retrocompatibilidade deve garantir que aplicações clientes atuais não sejam afetadas por mudanças no serviço (ENDREI et al., 2006). A tabela 2.1 sumariza os casos de mudanças de acordo com o seu veredito de compatibilidade.

A.3 O Modelo de Versionamento em Nível de *Feature*

Para entender evolução de serviços de uma maneira mais granular, precisamos enfatizar que aplicações clientes não são vinculadas ao conjunto de funcionalidades de um serviço como um todo, como também entendido em (PONNEKANTI; FOX, 2004). Na verdade, clientes são vinculados a um subconjunto de funcionalidades (por ex. operações) bem como seus tipos de dados para requisição e consumo. Dessa forma, caracterizamos um *serviço* como uma composição de *operações*, pelos quais dados são requisitados e consumidos de acordo com *tipos* pré-definidos (ex. mensagens, elementos do esquema). Como *serviços*, *operações* e *tipos* representam os aspectos relevantes que descrevem as funcionalidades dos serviços (FANG et al., 2007), esses três conceitos são referenciados como *features* de serviço (LEE; KANG; LEE, 2002), que são caracterizados como elementos versionáveis. A representação abstrata das *features* e seus relacionamentos são apresentados na Figura 4.1.

Neste trabalho, propomos o versionamento separado de *features* enquanto mantemos seus relacionamentos pelas suas versões, representadas por um grafo de *features*. Como mencionado, versionar *features* separadamente fornece uma maneira de identificar *features* afetadas durante o versionamento, ao passo que manter seus relacionamentos de dependência permite a consistência com a interface e a posterior análise de propagação de efeitos das mudanças.

A ideia do versionamento orientado a *feature* implica em prover um gerenciamento abstrato das diferentes partes da interface com o objetivo de versionar somente as *features* alteradas, ao invés de toda a interface. Assim, quando um novo documento de interface é exposto, nosso mecanismo a converte para uma representação abstrata interna, compara a descrição textual das *features* em relação às *features* existentes, assim como seus relacionamentos com outras *features*, e cria novas versões somente quando necessário. O modelo de versionamento proposto é apresentado na Figura 4.2.

A.4 Gerenciador de Versões

Este trabalho resulta em um Gerenciador de Versões que implementa os conceitos do modelo de orientação à *features* e, portanto, o versionamento de serviços em nível de *feature*, bem como a compatibilidade de versões nesse modelo mais granular. Em suma, a contribuição do Gerenciador de Versões é dividida em: a) a quantificação de *features* afetadas pelas mudanças entre duas versões de serviço, e b) a qualificação das *features* da mudança com relação a cada *feature*. O Gerenciador de Versões, ilustrado na Figura 5.1, é composto por três componentes: o Conversor de WSDL/*Features* responsável pela extração das *features* da versão do serviço de acordo com o modelo orientado à *feature*; o Analisador de Compatibilidade cujo objetivo é verificar a compatibilidade das mudanças entre versões de *features*; e o Repositório de Evolução de Versões, que armazena as versões das *features*, bem como suas dependências.

A.4.1 Conversor de WSDL/Features

Como mencionado, um serviço é representado por um documento de descrição de interface (WSDL). Desse modo, uma versão de *feature* corresponde a um fragmento desse documento, juntamente com suas dependências. Assim, para versionarmos as *features* do serviço separadamente, precisamos identificar as *features* dentro do WSDL, relacioná-las à versões de *features*, o que inclui a possibilidade de criar novas versões, e armazenar sua

representação abstrata no Repositório de Evolução de Versões. Esse processo requer duas fases: a) a extração de *features* do WSDL, e b) a análise das *features* para descobrir se sofreram mudanças em relação à todas suas representações prévias no repositório.

A nossa intenção é versionar somente as *features* que sofreram mudanças, ou aquelas que indiretamente são afetadas pelas mudanças. Nos referimos às *features* que sofreram mudanças como aquelas que tiveram seu fragmento de descrição alterado, aquelas que dependem de uma *feature* que não dependiam anteriormente, ou aquelas que não dependem de uma *feature* que antes dependiam. Nos referimos às *features* afetadas, aquelas que não sofreram mudanças, mas dependem direta ou indiretamente de *features* que sofreram.

A primeira fase no versionamento de *features* é a extração de representação das *features* que engloba os seguintes passos:

1. a análise do documento WSDL (ex. Figura 4.3) para identificação das *features* e seus relacionamentos;

2. a geração do grafo de versões de *features* (ex. Figura 4.5) em relação à representação abstrata da *feature* (ex. Figura 4.1); e

3. o processo de relacionar os fragmentos de descrição com sua versão de *feature* correspondente (ex. Figura 4.4).

O grafo resultante engloba todo conteúdo do WSDL e uma vez estruturado, deve-se analisar cada *feature* de maneira a descobrir se essa sofreu mudanças ou foi afetada, criando-se uma nova versão no repositório se ocorrer. Assim, para cada *feature*, o conversor analisa as versões de *feature* com o mesmo nome no repositório comparando seu conteúdo textual. A análise é feita de baixo para cima com relação ao grafo de *features* de maneira a verificar as mudanças e dependências corretamente. A análise é composta de quatro possibilidades, referidas como *casos de versionamento*:

1. Se a *feature* não existe no repositório, então a essa é criada com sua primeira versão.

2. Se a *feature* já existe e sua descrição é diferente à sua ultima versão, então essa é marcada como alterada e uma nova versão é criada.

3. Se a *feature* já existe e sua descrição é igual a uma versão já existente, então:

   (a) Se essa depende de uma outra *feature* marcada como alterada, então uma nova versão é criada para lidar com efeitos de propagação, ou seja, a *feature* foi afetada pela mudança.

   (b) Se essa não depende de nenhuma *feature* modificada, então todas as *features* que dependem dessa terão sua referência apontada para a versão de *feature* existente (igual) no repositório.

A.4.2 Analisador de Compatibilidade

O processo descrito nas seções anteriores extrai de um documento WSDL uma representação interna de *features*, na qual somente as partes do serviço explicitamente alteradas ou afetadas são relacionadas à novas versões. Caso contrário, versões existentes de *features* são associadas ao serviço. Dessa forma, qualquer serviço corresponde internamente a um grafo de versões no qual cada versão é relacionada a uma *feature*. O grafo também define as dependências entre versões de *features*, por exemplo, um serviço em relação

às suas operações ou as operações em relação aos seus tipos de dados. Porém, além de identificar quais os aspectos do serviço sofreram alterações, é necessário verificar se cada mudança é retrocompratível em relação a sua versão anterior.

O algoritmo proposto neste trabalho tem o objetivo de verificar a compatibilidade entre quaisquer duas versões de um serviço, o que implica em examinar recursivamente a compatibilidade de todas as *features* que descrevem o serviço. A análise de compatibilidade é feita de acordo com os casos da Tabela A.1.

O algoritmo recebe como entrada duas versões da mesma *feature*, $v_{feature,p}$ e $v_{feature,q}$. Então, o algoritmo verifica a compatibilidade da última com relação à primeira de acordo com o seguinte procedimento:

1. o algoritmo verifica se dependências de *features* presentes em $v_{feature,p}$ não foram removidas de $v_{feature,q}$ (casos 7 e 8 da Tabela A.1);

2. compara a descrição dos fragmentos associados com as versões (casos 4, 5 e 6);

3. avalia recursivamente a compatibilidade de todas as versões de *features* dependentes (casos 1, 2 e 3), e então:

4. define o relacionamento de compatibilidade, bem como o respectivo veredito.

O grafo de versão com raiz em $v_{feature,q}$ é examinado no modo profundidade-primeiro, que permite a detecção da propagação de incompatibilidades para as versões dependentes. O pseudo-algoritmo é apresentado na Listagem 1.

Table A.1: Casos de mudanças para avaliação de compatibilidade.

Casos	**Mudanças**	**Tipo de *Feature***	**Descrição**	**Veredito de Compatibilidade**
1	Adicionar	Operação	Adicionar nova operação ao serviço	*Compatível*
2	Adicionar	Tipo	Adicionar novo tipo como dependência de uma operação/tipo	*Compatível*
3	Adicionar	Tipo	Adicionar novo tipo como dependência de uma operação/tipo existente	*Incompatível*
4	Atualizar	Operação	Alteração de descrição	*Incompatível*
5	Atualizar	Serviço	Alteração de descrição	*Incompatível*
6	Atualizar	Tipo	Alteração de descrição com relação a ordem, cardinalidade ou tipo	*Incompatível*
7	Remover	Operação	Remover uma dependência de operação	*Incompatível*
8	Remover	Tipo	Remover uma dependência de tipo	*Incompatível*

A.5 Conclusões

Neste trabalho, apresentamos uma abordagem para o versionamento de serviços e a verificação de compatibilidade em uma maneira mais granular, em nível de *feature*, o qual permite delinear, quantificar e verificar as mudanças em uma perspectiva mais granular. O modelo orientado a *feature* permitiu representar a descrição do serviço como um todo em um grafo de dependência de *features*. Com a abordagem de versionamento de *features*, apresentamos uma maneira de versionar somente as *features* que foram modificadas ou afetadas pela mudança, que tem consequência direta na facilitação da identificação de mudanças.

Com relação ao versionamento, a contribuição deste trabalho é um modelo de versionamento que cobre a falta de uma perspectiva mais granular, bem como provê um mecanismo para registrar mudanças durante a evolução.

Com relação à compatibilidade, este trabalho apresenta um *framework* para avaliação automática de compatibilidade das mudanças. A avaliação de compatibilidade permitiu a analise do impacto da mudança em diferentes pontos do serviço, bem como a propagação da incompatibilidade em nível de *feature*.

Com relação à utilidade da nossa abordagem, realizamos experimentos de versionamento e avaliação compatibilidade sobre um serviço real onde conseguimos verificar que nossa abordagem suportou a eficiente quantificação e qualificação de mudanças, bem como o impacto. Dessa forma, nossa abordagem provê um mecanismo para geração de importantes informações acerca da evolução de serviços, ajudando provedores e consumidores durante a evolução dos mesmos.