FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Guidelines for the Development of Resilient Web Services to Enhance Business Process Continuity

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Marco Unterberger
Matrikelnummer 0726018

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Rauber

Wien, 25.09.2014

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Guidelines for the Development of Resilient Web Services to Enhance Business Process Continuity

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Business Informatics

by

Marco Unterberger
Registration Number 0726018

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Rauber

Vienna, 25.09.2014

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Marco Unterberger
Fuchsthallergasse 2/22, 1090 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

I would like to express special gratitude to my supervisor Prof. Andreas Rauber (IFS Research Group, Vienna University of Technology), who gave me the opportunity to do this thesis on the topic of Resilient Web Services. Thank you very much for your scientific guidance and constructive advice. I always benefited from your expertise and holistic view of the problem domain. Furthermore I would also like to thank my colleagues at Secure Business Austria (Vienna), especially Tomasz Miksa and Rudolf Mayer who offered a lot of encouragement during the work on this thesis.

Finally, I take this opportunity to devote my deepest gratitude to my beloved parents Gerda and Hans for their continuous support. You have encouraged me throughout my years of study. Special thanks belong to my grandmother and my aunt Erika for their great support and enthusiasm. Last but not least, I want to express my gratitude to Sabrina. Thank you so much for your understanding during the past year(s), for your love and your support throughout my studies.

Abstract

Processes, either in scientific or business domains, are in general subjected to decay. In general, processes often interact with external components, which are located outside of the process boundaries. Web Services have become the de facto standard for realizing remote software components. Thus, Web Services are often part of a process and therefore can have a strong impact on a correct process execution. However, processes themselves are designed for long lasting, whereas Web Services have a highly dynamic and varying nature. That means, their functional behaviour often changes on demand. That dilemma of volatile third party resources is a major driver for process decay. *Business Continuity Management* (BCM) is a framework for developing and implementing *Business Continuity* within an enterprise including various activities like risk management and process analysis. However, Business Continuity plans and strategies do not cover external artifacts sufficiently. Such external services involve a potential risk, but are hard to address by BCM as they are out of the sphere of influence.

This thesis analysis reasons why Web Services so easily become outdated. Based on a literature survey, the most common service changes scenarios causing the Web Service's dynamic nature are presented. In a first step we investigate challenges regarding Web Service's resilience. Therefore, we observe the Web Service beyond its public interface to identify non apparent challenges. In more detail, we concentrate our effort on two major challenges: *Web Service version management* and *Web Service dependency management*. We organized our work as follows: In the first step, theoretical concepts including requirements and policies addressing the resilience challenges have been developed. In a second step, we provided a reference implementation for that framework to support resilient Web Services. Both, the theoretical and practical contributions lead to the Resilient Web Service Framework. By applying this framework, it supports Web Service providers in offering resilient Web Services.

For the purpose of demonstration, we applied the Resilient Web Service Framework on a selected set of scenarios of the Web Service management domain. The introduced resilience annotations get automatically attach to the service's WSDL. It demonstrates the successfully capturing of Web Service's dependencies at runtime. Furthermore, notifications have been pushed in case of a dependency modification.

Kurzfassung

Geschäfts- als auch wissenschaftliche Prozesse interagieren oftmals mit externen Softwarekomponenten, welche außerhalb der eigenen Prozessgrenzen lokalisiert sind und haben zumeist einen starken Einfluss den Prozess. Web Services sind eine Technologie um solch externe Softwarekomponenten zu realisieren. Prozesse sind im Allgemeinen auf Langlebigkeit ausgerichtet. Im Gegensatz dazu ist die Funktionalität eines Web Services stark an reale Bedarfsanforderungen gebunden ist. Kommt es zu einer neuen Anforderung, wird die Funktionalität des Web Services dahingehend adaptiert. Das hat zur Folge, dass Web Services eine hohe Dynamik und Variabilität in ihrem Verhalten aufweisen. Dies ist hauptverantwortlich dafür, dass die Langlebigkeit von Prozessen stark gefährdet ist.

Betriebliches Kontinuitätsmanagement (im Englischen *Business Continuity Management*) ist ein Framework um Betriebliches Kontinuitätsmanagement innerhalb eines Unternehmens zu entwickeln und zu überwachen. Es beinhaltet unter anderem Aktivitäten wie Risikoverwaltung und Prozessanalyse. Jedoch decken die verfügbaren Aktivitäten externe Artefakte wie Web Services nicht zur Genüge ab. Solche externen Risiken sind mithilfe von Betrieblichen Kontinuitätsmanagement schwer bis gar nicht zu adressieren.

Diese Arbeit untersucht die Gründe, welche die Resilienz eines Web Services beeinträchtigen und entwickelt Richtlinien um die Resilienz von Web Services sicherzustellen. Beginnend mit einer Literaturrecherche werden die häufigsten Änderungsszenarien präsentiert, welche für die dynamische Natur der Web Services mitverantwortlich sind. Darauf aufbauend werden die Herausforderungen an ein *Resilientes Web Service* erforscht. Dafür ist es notwendig das eigentliche Web Service hinter der publizierten Schnittstelle genauer zu betrachten um nicht offensichtliche Anforderungen aufzudecken. Genauer gesagt konzentriert sich diese Arbeit auf zwei Hauptanforderungen. Zum einem ist dies die Web Service Versionierungsverwaltung und zum anderen die Web Service Abhängigkeitenverwaltung. Die Arbeit ist wie folgt aufgebaut: Im ersten Schritt werden theoretische Konzepte und Strategien entwickelt welche die Anforderungen an *Resiliente Web Services* adressieren. In einem weiteren Schritt wird eine Referenzimplementierung dieses Frameworks präsentiert. Durch die Anwendung dieses Frameworks sollen Web Service Anbieter dabei unterstützt werden, Resiliente Web Services zur Verfügung zu stellen.

Um die Fähigkeiten des Frameworks zu demonstrieren, selektierten wir spezielle Szenarien aus der Domäne Web Service Verwaltung. Dabei hat das Framework gezeigt, dass die Umsetzung einer resilienten Versionierungsverwaltung erfolgreich forciert wurde. Das erfolgreiche Erfassen der Abhängigkeiten eines Web Services zur Laufzeit wurde ebenso demonstriert. Des Weiteren wurde gezeigt, dass im Falle einer aufgetretenen Modifikation einer Abhängigkeit eine Benachrichtigung versendet wurde.

Contents

List of Figures

List of Tables

CHAPTER 1

Motivation

1.1 Setting

Digital Preservation, a relatively new research area in computer science, investigates preservation of digital artifacts to ensure their accessibility over the next decades. So far, the main focus of this research area has targeted the preservation of static digital artifacts like text documents and images. A trend out of this research field deals not only with the preservation of digital artifacts itself, but with the preservation of an entire (scientific) process. Preserving such a process includes different activities like preservation planning, custom software tools development, verification-validation for redeployed processes and also maintenance activities.

The Service Oriented Architecture (SOA) paradigm is state-of-the-art in a modern software development to design loosely coupled, dynamic exchangeable, platform independent software components. To guarantee interoperability and re-usability, these components are restricted to open, platform- and development-independent standards like SOAP[1],HTTP, WSDL[2] and many others. *Web Services* as the widely used implementation approach of it, have become the de facto standard for software components offered across organization boundaries as outsourced, reusable services. Such external Web Services are often part of processes to extend process functionality. Processes themselves are designed for long lasting, whereas -by default- Web Services act under a highly dynamic and varying environment. Furthermore, processes and also the included external Web Services always underlay certain quality assurances like throughput, latency, availability and most important security issues. To achieve a long term operating process, it has to be controlled and monitored. In the area of business processes, SOA offers a lot of advantages in gaining flexibility for business processes orchestration. But they also have immense drawbacks like any kind of outsourced software. From the point of view of a process owner such disadvantages are e.g.: threats for security and confidentiality; service quality assurance; service availability. When it comes to process preservation, you have to deal with all different

[1] `http://www.w3.org/TR/soap/`
[2] `http://www.w3.org/TR/wsdl/`

kinds of dependencies of a process. Preserving a process running within an accessible environment (meaning the process owner has direct access to all dependencies) is already a challenging research topic. But if there are also Web Services included, which are by definition outside of the process owners environment, it becomes an even more challenging task. This thesis is a contribution to make Web Services more reliable and sustainable to enhance the availability and simplify preservation of Web Service supported processes.

1.2 Problem Statement

As mentioned above, Web Services not only bring advantages for process owners, but also immense drawbacks. Fulfilling the loosely binding paradigm of SOA, a Web Service offers certain functionality, by computing an incoming request and—in most cases—sending a response. Web Service design principles are in contrast with tracing any consumers interaction, by default. Those principles perfectly satisfy the needs of interchangeability and other key aspects of a SOA environment rather than the requirements of a business process. To satisfy new business needs, Web Service providers are constantly forced to release new updates of an existing service. Thus, Web Services are bound to evolve dynamically over time to address changes in functional and non-functional requirements. The most common approach to manage updates on Web Services is to simply release a new version ([11]). However, a compatible Web Service versioning is necessary, as the provider cannot expect each consumer to change its implementation every time a new update has been released. This leads to the following problem:

What are the features of a resilient versioning policy and how to assist a service provider to stick to a such a policy?

There exists also more complex approaches for managing different Web Service versions ([3], [30], [22]). All these contributions focus on an accurate recognition of what are possible changes in the Web Service interface and what are best practices to avoid those changes. But, there is a lack of attention on possible threats of behavioral (non-functional) service evolution.

It is important to accentuate, that a proper version management for the interface of a service is not sufficient in our context. A Web Service depends on more than just a source code file which gets transformed to a platform independent interface. To ensure sustainability, we have to investigate also changes of service's dependable software components, data files, executables and hardware. Hence, we have to detect what are the dependable artifacts for a correct Web Service execution?

How to identify the Web Service dependencies and monitor its modifications?

How to detect and evaluate changes in the Web Service execution behaviour in case of a dependency modification?

In case a modification of a Web Service's dependency happens, all its consumers have to be notified about that. But since it is a fundamental part of SOA to fulfil the loosely binding paradigm, a Web Service is by default not aware of its consumers.

What notification approaches are applicable in the domain of SOA and what information has to be included?

1.3 Aim of the work

In this thesis we will develop guidelines and strategies for enhancing Web Service sustainability along with the aim to enhance process continuity. Therefore we first have to investigate what are typical pitfalls for processes caused by the SOA design paradigm. We will depict critical issues when it comes to Web Service sustainability. Furthermore, potential sources of problems responsible for a process break will be named and analyzed. Under consideration of released Web Services standards and specifications, we will investigate what requirements Resilient Web Services (RWS) need to fulfil and how these can be met. In the first case, we will show how current efforts can be applied to enhance Web Service resilience. In the second case, further research on how to tackle the left open challenges is necessary. We first develop theoretical concept and formulate policies to address those challenges.

The introduced RWS framework encompasses all policies. Additionally, we implement software prototypes to demonstrate the applicability of the developed framework. The target of the framework is to provide policies and a tool set supporting the Web Service provider in the process of a resilient Web Service development;

Please note: It is not the aim of work to build an out-of-the-box software product covering all possible aspects of resilient Web Services. Rather, the framework should support both, consumer and provider to overcome the major problems caused by the dynamically nature of Web Services.

1.4 Structure of the work

The content of this thesis is structured as follows:

- In chapter 2 we introduce important basic and also more advanced concepts concerning SOA and Web Services in general. That chapter gives an insight on what are processes and especially on what artifacts it depends on. We introduce the basic terminology of the Web Service stack and common Web Service quality factors. Followed by an introduction on SOA Governance, Service Level Agreements and provenance according to our domain. We also present the concept of a View Path for describing digital artifacts. In the last section, we give a short introduction on OWL.

- Chapter **??** surveys the different types of changes which can occur in Web Service evolution. Furthermore, we investigate the arising challenges of service updates, what are common practices and research contributions to deal with those challenges. Related efforts are presented and discussed.

- Chapter 4 discuses and presents requirements for a resilient Web Service design. What are the challenges to address? To meet these requirements, policies for a resilient Web Service design are presented. Among others, major requirements like the dependency identification process and a monitoring strategy are presented.

- The Resilient Framework, as a prototype solution is presented in chapter 5. That chapter focuses on the practical implementations of the concepts introduced in this thesis. It sum-

marizes our contributions to enhance Web Service sustainability. A diagram including the various framework components is presented.

- To demonstrate the capabilities of our resilient framework we have chosen various real life scenarios. Chapter 6 first presents the steps that are necessary to setup the resilient framework. Subsequent two update scenarios are presented. First, the service provider introduces an update of the service functionality. Second, the service's hosting system updates a software dependency of the service. The chapter closes with some critical remarks about the service limitations.

- The last chapter concludes this thesis by summarizing the roadmap to resilient enhanced Web Services. What are important cornerstones? How to get rid of common pitfalls? A brief outlook captures aspects and issues which are not covered in this effort, due to time constraints. *...otherwise, this thesis would never ever be finished.*

CHAPTER 2

Introduction

This chapter introduces basic and advanced topics related to the domain of this thesis. It encompasses information about architecture models, definitions, technologies, specification and standards related to Web Services and processes.

2.1 Service-Oriented Architecture (SOA)

Service-Oriented Architecture is an architectural model which has been arising in the last ten years. The aim of this paradigm is to break down logical functionality of a software system into smaller units of logic, also named *services*. Services are designed to encapsulate a specific task, rather than fulfilling a large amount of features. But that does not exclude services to make use of logic provided by other services to encompass a broader functionality. In such a scenario one or more services build a composed collection (of services). By separating concerns into smaller parts, large processes can be divided into subparts enhancing process agility and flexibility. To achieve this paradigm, services communicate using a defined message exchange pattern supported by standardized communication protocols and a standardized interface description language. We will present that in more detail in chapter 2.2. A bunch of different SOA definitions related to different contexts exists, either in relation to a more technical view or with a major focus on business and enterprises. In the following we present three common definitions of SOA:

- Definition by Gartner [1]

  > *Service-oriented architecture (SOA) is a design paradigm and discipline that helps IT meet business demands. Some organizations realize significant benefits using SOA including faster time to market, lower costs, better application*

[1] `http://www.gartner.com/it-glossary/service-oriented-architecture-soa`

consistency and increased agility. SOA reduces redundancy and increases usability, maintainability and value. This produces interoperable, modular systems that are easier to use and maintain. SOA creates simpler and faster systems that increase agility and reduce total cost of ownership (TCO).

- Definition by an article of the IBM press [7]

  A service-oriented architecture is a framework for integrating business processes and supporting IT infrastructure as secure, standardized components-services-that can be reused and combined to address changing business priorities.

- Definition by the Open Group [2]

  Service-Oriented Architecture (SOA) is an architectural style that supports service-orientation. Service-orientation is a way of thinking in terms of services and service-based development and the outcomes of services.

Concluding, the rise of SOA neither has been driven by software engineering nor by the business needs. It is much more the idea to separate concerns in a service-orientation style, which brings a lot of benefits for both communities. Although both terms, SOA and Web Services, are strongly related consider not to mix them. It has to be stated that just by the use of Web Services we do not have automatically established a SOA: SOA is an architecture style; Web Services are a set of standard that enable platform independent, interoperability in heterogeneous networks. Hence, Web Services represents one way to realize a SOA.

2.2 Services (as Web Services)

As already mentioned above, a service is an enclosed unit of logic. Literature often does not differentiate between the term *Service* and *Web Service*. To be conform with the definition of a Web Service from W3C[3],

> *A Web service is a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols.*

It has to provide a public, platform independent service interface description and has to support a standardized messaging protocol. Former will be realized with the *Web Service Description Language* (WSDL), latter is by default in most cases the *Simple Object Access Protocol* (SOAP). Deriving from the SOA paradigm, Erl [17] associates following principles to services in general, nevertheless those will be also valid for Web Services:

[2] `http://www.opengroup.org/soa/source-book/intro`
[3] `http://www.w3.org/TR/ws-arch`

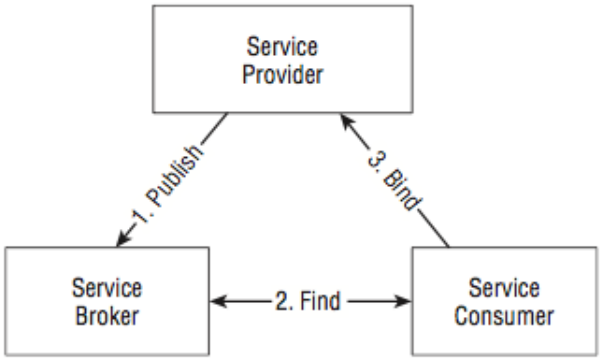

Figure 2.1: The famous *publish-find-bind* triangle [39]

- **Loose coupling** - Services maintain a relationship that minimizes dependencies and only requires that they retain an awareness of each other.
- **Service contract** - Services adhere to a communications agreement, as defined collectively by one or more service descriptions and related documents.
- **Autonomy** - Services have control over the logic they encapsulate.
- **Abstraction** - Beyond what is described in the service contract, services hide logic from the outside world.
- **Reusability** - Logic is divided into services with the intention of promoting reuse.
- **Composability** - Collections of services can be coordinated and assembled to form composite services.
- **Statelessness** - Services minimize retaining information specific to an activity.
- **Discoverability** - Services are designed to be outwardly descriptive so that they can be found and assessed via available discovery mechanisms.

Basic Web Service Architecture

A basic Web Service architecture models interactions between three different roles (Service Consumer, Service Provider, Service Registry/Service Broker). In figure 2.1 the interaction patterns for each role are presented. A Web Service can act in one or multiple roles at the same time. Those basic *publish-find-bind* architecture is also called the *SOA-Triangle*, which can be found in nearly every introductory SOA literature.

Rosen *et al.* [39] describe this architecture as follows: By registering a service description (WSDL) at the a public registry, the service provider publishes (=release) a service (for public use). A service consumer searches at the service registry/broker for an adequate service and

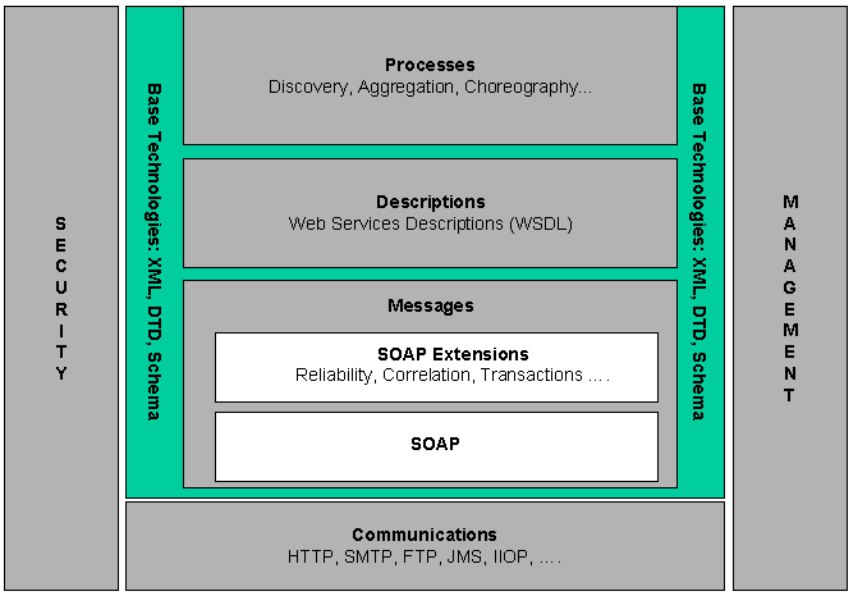

Figure 2.2: Web Service Architecture Stack [9]

dynamically binds to the Web Service. In theory that approach fits the needs of a SOA perfectly. Although it is very solid, in practices it is insufficient just knowing the logical syntax of a Web Service operation. Beside that, policy concerns and *Quality-of-Service* (QoS) attributes are of interest.

Web Service technology stack

Fulfilling Web Service principles mentioned above, a Web Service architecture involves different standards and technologies on different layers. Figure 2.2 [9] presents an overview of the most important technologies and its relations among each other.

- **XML** - The *eXtensible Markup Language*[4] is a simple, but highly flexible structured text format derived from SGML[5] optimized for automated machine processing, but also human-readable. Originally designed to for large-scale electronic publishing, XML has become the de-facto standard for a variety of data exchange patterns on the web and also between Web Services. Schema files constraint the appearance of elements in an XML file. The most common schema types are Document Type Definitions (DTD)[6] and the W3C XML Schema Definition (XSD)[7].

[4] http://www.w3.org/XML
[5] http://www.w3.org/MarkUp/SGML
[6] http://www.w3.org/TR/REC-xml
[7] http://www.w3.org/TR/2012/REC-xmlschema11-1-20120405

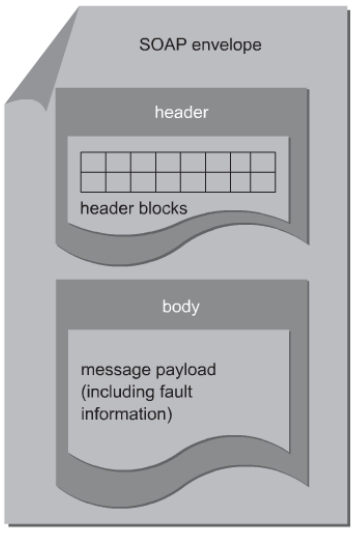

Figure 2.3: Standard SOAP envelope [17]

- **SOAP** - The *Simple Object Access Protocol* (in its current version 1.2[8]) is a lightweight XML based protocol intended for exchanging structural information in distributed environments. Through its simplicity and extensibility it perfectly fulfills requirements of message exchange patterns in Web Service communication. A SOAP message is build up of an *Envelope* elements, which mainly consists of two parts: An optional *Header* element consisting of one or more header blocks containing application-specific information like processing instructions, security, authentication or routing. So to say all WS-* extensions elements are implemented within those header blocks. The second part is the required so called *Body* element which contains the actual payload information intended for the ultimate message recipients. Figure 2.3 illustrates a SOAP envelope.

- **WSDL** - The *Web Service Description Language* (in the current version 2.0)[9] provides a model for describing a service endpoint and its operations in an abstract reusable manner. In previous specifications the endpoint was described independent of concrete network protocols or data types. Only the binding element relates the abstract definition of services to network protocols and data format specifications. In version 2.0 the model was redesigned, to put even more emphasis on re-usability and the separation of independent design concerns. In the introduction of the version specification [14] it says at *abstract level* a Web Service is described in terms of *messages* it sends and receives; *operations* which associates a message exchange pattern with one or more messages; an *interface* groups together operations without any commitment to transport or wire format. At a *concrete level*, a *binding* specifies transport and wire format details for one or more in-

[8] `http://www.w3.org/TR/soap12-part1/`
[9] `http://www.w3.org/TR/wsdl20`

terfaces. An *endpoint* associates a network address with a binding. And finally, a *service* groups together endpoints that implement a common interface.

A sample WSDL 2.0 document can be found in the appendix A.1.

2.3 Web Service classification

On the roadmap to design a guideline for Resilient Web Services we have to investigate the different behavior types of services. In general, Web Service can be divided into *stateful* and *stateless*. A stateful Web Service keeps state information between one invocation and another. This means that the response of a stateful Web Service not only depends on the request data, but also on the internal state of the Web Service. In contrast to that, the response of stateless Web Services only depends on the request data. Dranidis *et al.* [16] dig deeper and present further distinction factors for stateful Web Services. Their effort results in three major criteria how to distinguish stateful Web Services:

conversational/non-conversational By *conversational* it is meant, that Web Service operations can only be accepted in a specific sequence. This has to do with the state of the service itself and the state-manipulating operations. To give an example: In case of a shopping cart service it would not make any sense to accept the 'order' operation call before once the 'add item' operation has been called. In *non-conversational* Web Services all operations can be accepted at all states.

private-state/shared-state A stateful Web Service always holds a certain state. We can differentiate this state in a so called *private-state* or *shared-state*. The distinguish mark of a private-state is that in such a Web Service state changes depends exclusively on the sequence of operation invocations. In contrast to shared-state, in which state changes can be triggered by other services and applications in the environment.

transient-state/persistent-state If a Web Service operates in a *transient-state*, state changes will be reset after the session has been completed and will be initialized at the beginning of a new session respectively. Web Services with a *persistent-state* outlasts the duration of a session and so to say 'remembers' the state.

Thus, a stateful Web Service can be classified with one option out of each criteria. To give an example, we choose the *Amazon Shopping cart*, which is of course *stateful* and additionally: contains *conversational* methods, holds a *shared-state* and operates on a *persistent-state*.

2.4 Web Service Life-Cycle (from a software engineering perspective)

From the service provider view, the Web Service Life-Cycle encompasses at least four phases [29]: *Build*, *Deploy*, *Run*, and *Manage*. In the *build* phase the core service development happens. Collecting requirements, implementing functionality and testing are the main activities. The

deployment phase includes the publication of the service and making it public available. During the *run* phase, the service interface can be accessed by consumers. The service is ready for processing incoming requests. In the *manage* phase ongoing administration and maintenance activities are executed. Such a life-cycle is nothing new in software engineering. At a first sight, it does not really differ from traditional software application development. But there is one big difference: (Traditional) software applications are mostly tailored products considering and satisfying previously contracted needs. That does not apply to for Web Services. It is common practice that Web Services are not build upon specific consumers needs. Web Services are mainly offered by enterprises to make certain functionality or data available to a public audience (e.g.; FedEx shipment tracking). It is also very common to wrap legacy system functionality and make it available via a Web Service, than to develop a custom adapter for that purpose. Such considerations often result in a more abstract and less concrete (tailored) service design. That means that services are rather designed for a general purpose, then for a specific consumer. That fact definitely supports the SOA paradigm, but adds additional challenges for Resilient Web Services. Among others, two challenges are: Due to changes in the underlying legacy system, the Web Service behaviour gets unknowingly affected; Legacy systems often use an outdated data format internally, which must be transformed to a current usual data formats firstly. The Resilient Web Service framework—developed within this thesis—will cover aspects of all life-cycle phases.

2.5 Business Processes (BP)

A *business process* (BP) performs various tasks in a specific order to produce value for an organization. Such processes can have different subbranches including tasks which might or might not be executed. Tasks can be processed fully automatically or need human interaction. Tasks can be executed locally or distributed over a network. It can contain semantic elements for looping, parallel execution or decision making (if/else). Tasks can cross organizational boundaries and interact with other external BPs to produce a common value. The *Business Process Model and Notation* (BPMN) (in its current version v2.0[10]) standard by the *Object Management Group* (OMG) provides a graphical notation to model business processes. The intent of BPMN is to provide an easy understandable notation for all collaborating business parties. It should overcome the gap between the process design and the process implementation. Today, nearly every nameable software vendor provides its own BPMN modeling tool.

As already mentioned in the motivation for this thesis, tasks are often realized by external services. Depending on the logical context of a service, it either performs an individual task or is responsible for performing a group of task, like a subprocess or even the whole process. Figure 2.4 presents the several sample cases.

[10] `http://www.omg.org/spec/BPMN/2.0/PDF`

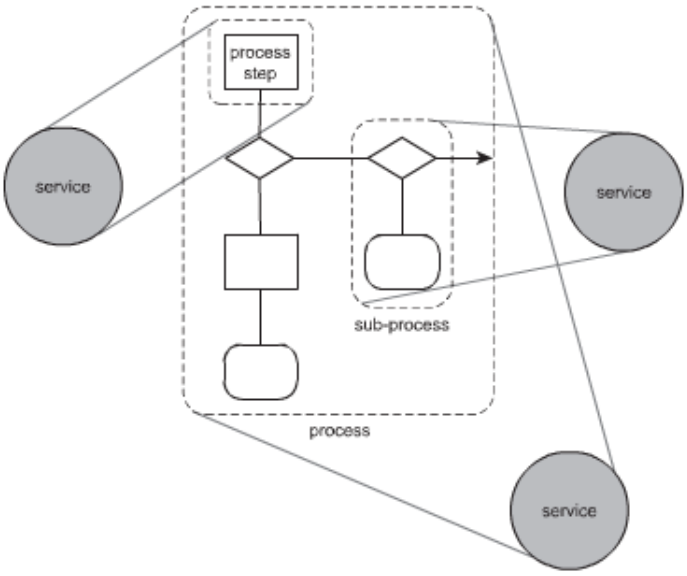

Figure 2.4: Services can encapsulate varying amounts of logic [17]

Process decay

Processes are subject to decay in their ability to re-execute them sometime in future. Belhajjame *et al.* [5] present that standard workflows are suspended to different risks and scenarios which make a workflow become inoperable in a certain way. Typical workflows suffer from decay due external resources or time specific datasets. External resources can become unavailable or change their behaviour. Data-tuples in databases can evolve over time. These are only two issues, which can lead to severe consequences in re-executing a process. To avoid process decay Hettne *et al.* [25] present a guideline of best practices including: *Make it executable from outside the local environment* and *Test and Validate* the correctness of the outcome. Also the usage of meta-data for describing artifacts, behaviour, expected input and output in the process is mentioned. Hence, Web Services are often parts of such processes, their sustainability highly influences process decay.

Process preservation

The TIMBUS project [11], a co-funded EU project focuses on the design and realization of resilient business process (driven by a risk mitigation perspective). Within this project different aspects of process preservation are covered. From a more detailed perspective, a process is more than just the execution of process steps. A process is always executed in specific environment with a certain context. These include legal aspects, liabilities, personal data, infrastructure information,

[11] `http://timbusproject.net`

authentification and rights management and many more. So digital preservation also has to keep an eye on those context and environment related issues to make processes available for a long time span. Therefore, Strodl *et al.* [42] present the three phase approach realized within the TIMBUS project. They additionally show the applicability of this approach for a classification process. The three phases are *Plan*, *Preserve* and *Redeploy*. In the planning phase the original process including all relevant components and dependencies is captured within a context model. During the preserving phase the original business process execution is captured from the source system. Additionally, required preservation actions are preformed to prepare the process for archival storage. In the redeployment phase the process get roll-out on a new environment. Therefore, the access and usage of services, data and legal conditions have to be adjusted to the new environment. Are more detailed presentation of this approach is shown in figure 2.5.

Figure 2.5: TIMBUS three phases approach [42]

2.6 SOA Governance

SOA Governance enforces compliance of services and other artifacts with defined policies and guidelines and is therefore essential to a successful SOA.To manage SOA Governance it is most important to separate policy logic from business logic. Erl *et al.* [18] split the so-called *SOA Governance* life cycle in four different phases (= types of governance):

- **Design-time governance** Policies and procedures to ensure that the right services are built

and used.

- **Deploy-time governance** Policies that affect the deployment of services into production.
- **Run-time governance** Policies that affect the binding of consumers and providers.
- **Change-time governance** Policies and procedures that affect the design, versioning, and provisioning of service enhancements.

What we can derive is that governance is not only attached afterwards on a releasable service. It is present in each phase of a service life cycle. From the first architectural decision to the final service retirement. For this thesis, we keep all four phases in mind, but —in our case— the two most important ones are the Run-time and Change-time governance. What can we provide to support service providers and consumers in this two phases?

2.7 Service-level Agreement (SLA)

Gartner defines a service-level Agreement in general as follows[12]:

> *An agreement that sets the expectations between the service provider and the customer and describes the products or services to be delivered, the single point of contact for end-user problems and the metrics by which the effectiveness of the process is monitored and approved.*

As mentioned above in 2.2, operations of a service endpoint are described in its corresponding WSDL document. With a WSDL interface, a service is defined in a very technical (functional) manner focusing on data types, communication protocols, message exchange patterns, but lacking off semantic (non-functional) information. But those non-functional requirements can become very important in a B2B context.

Before we start with SLAs, we first have to explain what are *Non-functional properties (NFP)*: Other than functional properties (FP), which describe the behaviour of a service, NFPs are requirements describing characteristics of a service. It is not about, what a service DOES (defined by FPs), but HOW a service performs its functionality. In context of web services and also services in general, those are also named Quality-of-Service attributes. Anbazhagan *et al.* [32] present the following popular QoS metrics for Web Services:

- **Availability** is the quality aspect of whether the Web service is present or ready for immediate use. Availability refers to the ratio of time in which the Web Service is up and running.
- **Accessibility** is the quality aspect of a service that represents the degree it is capable of serving a Web service request.
- **Integrity** is the quality aspect of how the Web service maintains the correctness of the interaction in respect to the source.

[12] `http://www.gartner.com/it-glossary/sla-service-level-agreement`

- **Performance** is the quality aspect of Web service, which is measured in terms of throughput and latency.
- **Reliability** is the quality aspect of a Web service that represents the degree of being capable of maintaining the service and service quality.
- **Regulatory** is the quality aspect of the Web service in conformance with the rules, the law, compliance with standards, and the established service level agreement.
- **Security** is the quality aspect of the Web service of providing confidentiality and non-repudiation by authenticating the parties involved, encrypting messages, and providing access control.

Other popular metrics are *Cost-Per-Usage* and *User Rating Feedback* which are not covered in [32].

A SLA is a legal agreement between a service provider and a client based on QoS metrics. With the help of those metrics, so called *Service-level Objectives (SLO)* can be defined, which must be achieved to fulfill the agreement. Additionally for each SLO a validity period and enforcing penalties are defined. Keller *et al.* [28] present a framework called *WSLA (Web Service Level Agreements)*, which can be used for specifying and monitoring SLAs for Web Services. It consists of a flexible and extensible language (*WSLA Language*) based on XML Schema and a *WSLA Runtime Architecture*. It can be used to describe the complete life cycle of a SLA within it various stages. From *SLA Negotiation and Establishment* (stage 1) to *SLA Termination* (stage 5).
WSLA Language - The language specification defines a type system for various SLA artifacts and structures the SLA in three sections as follows: all contractual parties, service description specified by service characteristics and obligations, which defines various guarantees and constraints to be imposed on SLA parameters.
WSLA Runtime Architecture - The runtime architecture is the composition of different elementary services needed to enable the management of an SLA throughout its life cycle. The core of the runtime architecture is the *SLA Compliance Monitor*. It is responsible for deploying, measuring and evaluating interactions. An implementation is part of the IBM Web Service Toolkit (WSTK).
As this thesis is based on a programmatically alignment, SLAs represent the contractual counterpart to it. From the consumers point of view, SLA can be used as a hedging instrument to protect themselves against a Web Service provider, who can not assure the resilience of its Web Service or violates it demonstrably.

2.8 Provenance enhanced Web Services

The term *provenance* original comes from the field of arts and literature. It describes the origin of an artifact and thereby valuates it and helps to prove its authenticity. In the past years this term has been adopted in the IT—especially in e-Science—to refer the origin of data elements, so called *data provenance*. A very popular survey of data provenance in e-Science is presented

by Simmhan *et al.* [41]. In a further effort [40] the same author investigates the applicability of provenance, which architectures are common for various scientific and business domains and presents an entire taxonomy of provenance techniques based on a survey. One such architecture is called *Data Processing Architecture* (DPA) by the author. Furthermore, it is stated that a SOA is a typical instance of a DPA. By each Web Service operation invocation, the service processes input data and responds with transformed output data. Such transformations can be described by a *directed acyclic graph* (DAG) as shown in figure 2.6 [40].

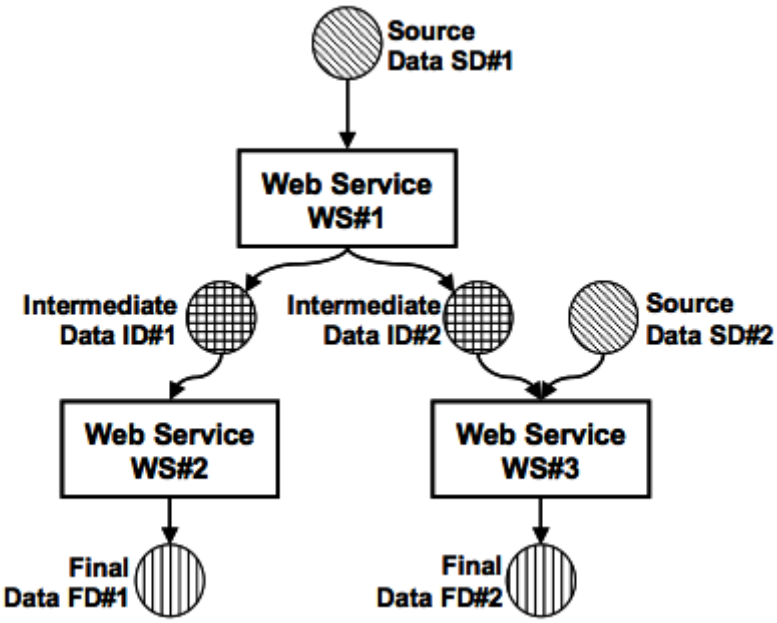

Figure 2.6: Data Process Architecture represented as a DAG from [40]

To retrace the processed data presented in Web Service's response, it is needed to attach provenance information to each particular response. More details on which information should be included is presented in section 4.4.

2.9 Commit Hooks in a Version Control System

In a *Version Control System* (VCS), a commit is the operation in which a set of distinctive changes (e.g. update or add new source code files) gets transferred from a local working copy to the VCS. Advanced VCS systems support so-called pre- and post-commit hooks. A pre-commit hook is executed before the commit operation is performed. In contrast to a post-commit hook, which gets executed after a successfully execution of a commit. In general, a pre-commit hook is used to validate and control changes in the files which are going to be committed. In practices, such hooks are also used e.g. to run formatter tools on the source code files before they get committed. A post-commit hook can be used to notify other tools (in the tool-chain of software

development) about source code changes. A popular use case for that is, to trigger a new build of the software project by notifying a CI server like Jenkins.

By using a pre-commit hook, every new commit of the Web Service gets validated against its backward compatibility. In case of a non successful validation the commit gets aborted. Details about the application of the pre-commit hook are presented in section 5.2.

2.10 Compatible Web Service evolution

The *IEEE Standard Glossary of Software Engineering Terminology* [1] defines the term compatibility as follows:

> *The ability of two or more systems or components to perform their required functions while sharing the same hardware or software environment* and *The ability of two or more systems or components to exchange information.*

When talking about a compatible evolution, Web Services are expected to limit changes to those that are either backward or forward compatible, or both: According to Web Service interface versioning, the WSDL 2.0 specification defines backward and forward compatibility as follows:

Backward compatible - The Web Service behaves correctly if it receives a message in an older version of the interaction language.
Forward compatible - The Web Service behaves correctly if it receives a message in a newer version of the interaction language.

In case of a RWS, a Web Service update must guarantee backward compatibility. Section 3.3 presents various approaches addressing backward compatibility of interface evolution. However, those approaches focus only the Web Service interface itself, but not on its behaviour.

Further details according to a compatible Web Service interface design are presented in WSDL 2.0[13].

2.11 View Path concept

From the viewpoint of a business process owner, a Web Service's interface describes its offered operations, information about the transport protocol and of input and output data types. Additional information beyond the Web Service's interface is non-obvious for interacting parties. Web Services depend on both, software and hardware dependencies. As one way to represent the Web Service's software and hardware dependencies, we want to present the *View-Path (VP)* concept, which simply means that any digital object needs an environment to render it. Van Diessen *et al.* [43] define a VP as follows:

> *A View Path represents a full set of functionality needed to render the information from a digital object.*

[13] `http://www.w3.org/TR/wsdl20-primer`

Furthermore, *van Diessen et al.* define a *basic layer model* including four layers, which are involved in the performance/rendering process of a digital object. In following itemization we describe each layer:

Data format layer defines the structure format of the digital objects' bit stream.

Application layer includes all applications which are needed to create, use, modify and view information.

Operation system layer provides the shared functionality which is needed by every application.

Hardware layer is the platform on which the digital object is rendered into a physical object, like a screen representation or a printed document.

That basic layer model shows the chain of software and hardware dependencies needed for a correct rendering. Guttenbrunner *et al.* [24] add that, depending on the digital object, some layers in the View Path can be missing or additional layers can be present. They demonstrate e.g. in case of a Java application, the *Java Virtual Machine (JVM)* itself as an additional layer between the application and operation system layer. Any change in the View Path can lead to changes in the performance/rendering of the digital object. However, depending on the digital object itself, there can exist more than one valid View Path.

Applying the View Path layer model supports us identifying the Web Service's dependencies (see section 4.7).

2.12 Service Models

The *National Institute of Standards and Technology* (NIST) [38] defines three different types of service models.

> *Software as a Service (SaaS). The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings.*
>
> *Platform as a Service (PaaS). The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment.*

Infrastructure as a Service (IaaS). The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls).

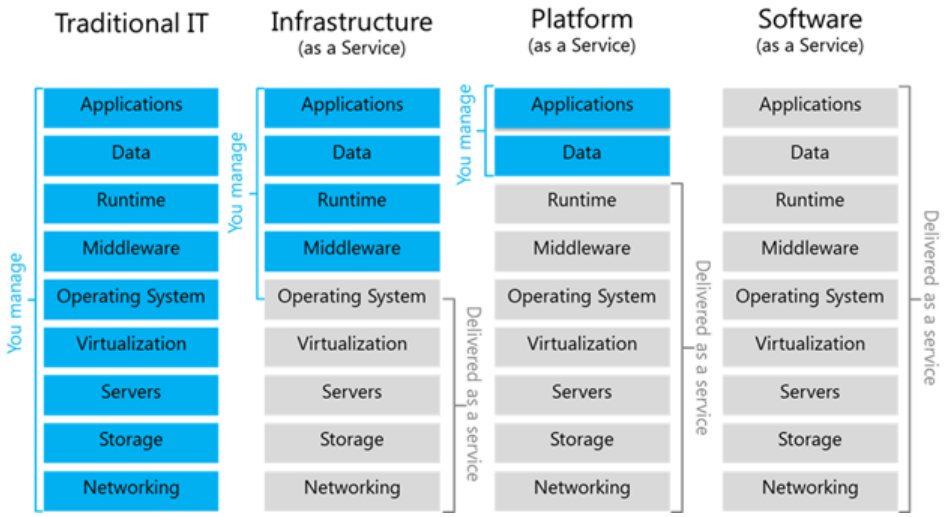

Figure 2.7: Visualization of traditional IT environment in comparison to the various service models in Cloud Computing

Figure 2.7 presents the traditional IT environment approach in comparison to the three different Cloud Computing approaches defined by the NIST. Depending which service model the RWS provider chooses, its responsibilities vary. In a traditional IT environment the RWS provider has to manage—and therefore is responsible for—all layers of the Web Service's View Path. To the purpose of focusing the company's main capabilities on its core business, outsourcing of tasks and responsibilities is common practices in IT. With cloud computing a separation of responsibilities is realizable. In the IaaS model, the provider outsources the complete infrastructure. Thereby, it transfers the hardware responsibilities to another vendor. In PaaS all layers are outsourced, except the Application and Date layer. That common model is offered by notable vendors like Google App Engine, Amazon AWS, Windows Azure. PaaS delivers runtime environment, application server and database instances out of the box. Next to the benefit of outsourcing those responsibilities, PaaS provider mostly guarantees high scalability and elasticity. According to the viewpoint of a RWS provider, next to a traditional IT environment, PaaS and IaaS are usable service models. Since the RWS provider will offer its own functionality to consumers, the SaaS model is not applicable. The responsibilities for collecting the depend View Path vary depending which service model the RWS provider chooses. In section 4.7 further details are provided.

2.13 The Semantic Web

In an article published in the *Scientific American* in 2001 Berners-Lee *et al.* [6] introduced the term *Semantic Web* for the first time and define it as follows:

> *The Semantic Web will bring structure to the meaningful content of Web pages, creating an environment where agents roaming from page to page readily carry out sophisticated tasks for users ... The Semantic Web is not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation.*

It described the evolution of the traditional existing "Web of documents", which largely consists of documents for humans to read, to a "Web of data" that includes data and information for machines to manipulate. That should be realized by a standardized way of expressing the relationships between -more or less- static web pages. It should allow machines to understand the meaning of digital context and of hyperlinked information. Therefore, the need for expressing meaning and knowledge representation have to be satisfied (*Machine-Understandable information*). Instead of publishing information to be consumed by humans, publish machine-processable data and metadata using languages that can be understood by machines.

Todays vision of Semantic Web drifted to a concept of the—so called—Data Activity[14]. As stated on the official website, *The overall vision of the Data Activity is that people and organizations should be able to share data as far as possible using their existing tools and working practices but in a way that enables others to derive and add value, and to utilize it in ways that suit them. Achieving that requires a focus not just on the interoperability of data but of communities.*

2.14 The concept of ontologies

Ontologies are considered one of the pillars of the Semantic Web. An ontology defines the concepts and relationships used to describe and represent an area of concern[15]. In short an ontology is a specification of a conceptualization. Gruber [23] defines the term ontology in the context of computer and information science as follows:

> *An ontology defines a set of representational primitives with which to model a domain of knowledge or discourse. The representational primitives are typically classes (or sets), attributes (or properties), and relationships (or relations among class members). The definitions of the representational primitives include information about their meaning and constraints on their logically consistent application.*

Depending on the application and the according area, ontologies can become very complex. Beside others, two main needs can be satisfied by the use of ontologies. Firstly, it can be used

[14] `http://www.w3.org/2013/data`
[15] `http://www.w3.org/standards/semanticweb/ontology`

to *organize knowledge by using the power of linked data*. Secondly, *complex reasoning procedures* can be applied. W3C offers different formats to describe and define different forms of vocabularies in a standard format. The most important are the Resource Description Framework Schema (RDF Schema) [16] (in its current version 1.1 from February 2014) and the Web Ontology Language (OWL) [17] (OWL 2, an extension and revision of OWL has been released in December 2012). The first one provides a data-modelling vocabulary for RDF data by combining several basic concepts and the abstract syntax of RDF. OWL is described in the next section.

Web Ontology Language (OWL)

The Web Ontology Language is a computational logic-based language such that knowledge (expressed in OWL) can be reasoned by applications. This can be useful either to verify the consistency of that knowledge or to make implicit knowledge explicit (i.e. reason). OWL 2 [26], an extension from best practices and experiences of OWL is a language for expressing ontologies in a declarative (logical) way. It is designed to formulate, define and reason about domains of interest. To represent knowledge in OWL 2 there exists three basic notations: Axioms, the basic statements that an OWL ontology expresses. Entities, are elements used to refer to real-world objects. Expressions, are combinations of entities to form complex descriptions from basic entities. Additionally there exists different syntaxes for OWL 2 serving different purposes. The RDF/XML[18] syntax is more or less the default one and therefore mandatory to be supported by all OWL 2 tools. It defines the mapping between the structural specifications of OWL 2 and RDF graphs. The Manchester[19] syntax is designed to be easier for non-logicians to read. It exists also a OWL XML[20] syntax which defines the XML serialization for OWL 2. There exists tools which can translate between the various syntaxes.

[16] http://www.w3.org/TR/2014/REC-rdf-schema-20140225
[17] http://www.w3.org/TR/2012/REC-owl2-primer-20121211
[18] http://www.w3.org/TR/2012/REC-owl2-mapping-to-rdf-20121211
[19] http://www.w3.org/TR/2012/NOTE-owl2-manchester-syntax-20121211
[20] http://www.w3.org/TR/2012/REC-owl2-xml-serialization-20121211

CHAPTER 3

Challenges of Web Service evolution

Web Services are part of modern businesses and therefore—like any artifact in a rapidly changing environment—are subject to constant functional and behavioural changes. Such changes can be triggered by business needs, regulations and other business-related events. What are common change scenarios are covered by an empirical study on the evolution of real life Web Service like *Amazon EC2* or *PayPal SOAP API*. Fokaefs *et al.* [21] present the most common service change scenarios. To understand how services evolve, they investigate what types of changes are more or less frequent, and whether these changes endanger the stability of the service clients. Table 3.1 shows the evolution profile for well-known Web Services including the Amazon EC2[1], FedEx Package Movement Information and Rate Services[2], PayPal SOAP API[3] and the Bing search service[4]. As shown in the table, most service evolutions are dominated by additions. The authors derive two conclusions thereof: Firstly, the investigated services were in a stage of rapid development and high expansion. Secondly, radical changes like deletions are in most cases avoided. Also in one service (FedEx Package Movement Information) an increasing number of changes have been noticed. This indicates that this service was in a more stable stage and developers performed restructuring and perfective changes.

3.1 Challenges of Web Service evolution

Miksa *et al.* [34] describe possible sources of change in Web Services. The authors classify the changes into two categories: *Internal changes* and *External changes* depending on the view of the process owner. Furthermore, they identify four categories to classify the possible effects when altering Web Services: *Web Service becomes unavailable*, *Web Service changes its communication interface*,*Functional changes*,*Behavioural changes*. Functional changes relate

[1] http://aws.amazon.com/ec2

[2] http://www.fedex.com/us/developer

[3] https://www.paypalobjects.com/en_US/ebook/PP_APIReference/architecture.html

[4] http://www.bing.com/developers

Table 3.1: Evolution profile of Web Services [21]

Service	Version	Changed(%)	Deleted(%)	Inserted(%)
Amazon EC2	2	2.82	0	97.18
Amazon EC2	3	13.33	0	86.67
Amazon EC2	4	50	0	50
Amazon EC2	5	8.82	0	91.18
Amazon EC2	6	16.67	50	33.33
Amazon EC2	7	1.17	0	98.29
Amazon EC2	8	1.40	0	98.60
Amazon EC2	9	3.54	0.88	95.58
Amazon EC2	11	2.67	0	97.33
Amazon EC2	12	5.56	0	94.44
Amazon EC2	13	0.79	0	99.21
Amazon EC2	14	2.70	0	97.30
Amazon EC2	15	10.26	0	89.74
Amazon EC2	16	1.08	0	98.92
Amazon EC2	17	64.90	0	35.10
Amazon EC2	18	31.06	0	68.94
FedEx Rate	2	8.93	21.43	69.94
FedEx Rate	3	9.20	5.75	85.06
FedEx Rate	4	8.11	17.05	74.84
FedEx Rate	5	8.00	20.00	72.00
FedEx Rate	6	1.51	6.67	91.83
FedEx Rate	7	3.05	30.46	66.50
FedEx Rate	8	11.48	12.02	76.50
FedEx Rate	9	11.53	42.88	45.59
Bing	2.1	0	21.33	78.67
Bing	2.2	0	9.38	90.63
Bing	2.3	0	0	100.0
Bing	2.4	0	0	100.0
PayPal	53.0	2.33	0	97.67
PayPal	62.0	0.55	0	99.45
PayPal	65.1	1.35	0	98.65
FedEx Pack.	3	80.00	0	20.00
FedEx Pack.	4	100.00	0	0

to internal changes of the service while the interface stays the same. For example, switching the temperature unit from Kelvin to degree Celsius. In addition to functional changes, we have to consider also behavioural changes. Those are changes, which results in an identical output, but leads to another service quality behaviour. Such changes are mainly concerned with QoS issues like security, availability and traditional performance indicators like response time or throughput.

Li *et al.* [31] present by an empirical approach how service evolution affects clients. They identify 16 common change patterns and also try to bridge the gap to automatic client migration. Although there are a lot of contributions addressing that topic, most of them lack of attention to behavioural challenges.

According to Papazoglou [37] services can evolve by accommodating a multitude of changes along the following functional trajectories:

Structural changes Changes that occur in the service type, messages, interfaces and operations.

Business protocol changes Business protocols help describing the structure and the ordering of the messages that are exchanged between a service and its clients in a certain usage scenario. Changes in policies or regulations can lead to changes in the business protocol.

Policy induced changes Changes in the policy assertions and internal or external constraints on the service. Those changes limit or specify any aspect of a business agreement that is agreed between interacting parties.

Operational behaviour changes Changes, which concentrate on analyzing the effects and side (cascading) effects of changing service operations.

Furthermore, he classifies the nature of service changes depending on the effects and side effects they cause: *Shallow Change*, which is strict localized to a service and therefore *only* effect that service and its clients. Whereas a *Deep change*, are cascading types of changes which extend beyond the clients of a service possibly to entire value-chain. Typical shallow changes are changes on the *Structural level* and *Business protocol* changes. *Policy induced changes* and *Operational behavior changes* are typically deep changes.

Functional and behavioural changes are more or less hard to detect, but can cause severe problems for a correct process execution. Brown and Ellis [11] differentiate between *backward-compatible* changes and *non-backward-compatible* changes. Backward-compatible changes are changes, which do not affect the requester implementation, which means a certain subpart of the interface stays the same. Non-backward-compatible changes affect the requester's implementation. Table 3.2 presents an overview of reasons for possible changes based on [34] , [37], [11]. In addition each issue is classified, its backward-compatibility is checked and information on the impact for a consumer is presented.

Table 3.2: Possible changes during Web Service evolution

Issue	**Description**	**Classification** *[**F**unctional/**B**ehavioural]*	**BW-Compatibility** *[**Y**es/**N**o]*	**Impact** *[**N**o/**C**an have/**H**igh]*	**(Can) lead to** *[**N**o-impact/**U**nexpected **R**esults/**U**navailability]*
Internal source code update	Altering requirements like performance enhancement, bug fixing or code refactoring forces the Web Service development team to update the current stable source code version.	F,B	Y,N	C	R
Changing method parameters	Driven by changing requirements, a service provider is forced to change the service interface. A change can effect the amount of parameters, type of each single parameter or switch an optional parameter to a required one.	F	N	C	U
Dependency changes	Due to an outdated (not available any more, not maintained) dependency, a suitable replacement have to be found and packaged within the current Web Service.	F,B	Y,N	C	R
Licensing of Dependencies changes	Like the Web Service itself, also the dependencies can evolve over time and with it the license policy can change. Dependencies can become fee-based or illegal for commercial usage. In the first cases one can pay the fee or search for a substitute. In the second case one have to search and find a suitable substitute.	B	Y,N	H	U

Web Service Policy changes	WS-Policy (W3C Recommendation[5]) defines a framework for expressing domain-specific capabilities and requirements. By changing such a policy for a deployed Web Service, the request behaviour can be limited or more worse, the requester may be denied from accessing the service. Both can make the Web Service useless for a consumer.	F,B	N	H	U
Web Service Security changes	Due to a successful security exploit of the Web Service backend or new security policies, Web Service providers are forced to adapt the current policy. e.g.: add user authentication for Web Service calls;	F,B	N	C,H	U
Add operation	Add a new operation to the existing Web Service.	F	Y	N	N
Remove operation	Delete an existing operation of the Web Service.	F	N	H	U
Web Service Address changes	Either the address of the WSDL Interface or the Web Service endpoint address itself change.	F	N	H	U
Changing the Web Service transport protocol	Although SOAP enables protocol independence at first sight, it will cause problems when a Web Service Provider decides to change the underlying transport exchange protocol. Without any doubts, the most used protocol is HTTP/S, but nevertheless FTP, SMTP or a message exchange approach like the Java Message Service (JMS[6]) can be used.	F	N	H	U

To get rid of some of the state problems, various authors and vendors of SOA frameworks came up with different approaches for different Web Service Lifecycle phases. There exist several approaches, which are applied on a already deployed service. In contrast to that, there exist approaches developed to be applied at design time. In the subsequent sections we present various versioning approaches.

[5] http://www.w3.org/TR/ws-policy/

[6] http://www.jcp.org/en/jsr/detail?id=914

Table 3.3: A general comparison of the three versioning strategies [19]

	Strategy		
	Strict	**Flexible**	**Loose**
Strictness	high	medium	low
Governance Impact	high	medium	high
Complexity	low	medium	high

3.2 Addressing Web Service evolution by versioning

In this section we will present state-of-the-art approaches tackling the Web Service versioning dilemma. This topic has become very popular in the last years. Various business driven and research driven efforts related to that problem have already been published.

Erl *et al.* [19] define three main versioning strategies for Web Services based on their compatibility properties: *Strict*, *Flexible* and *Loose*.

Strict Any compatible or incompatible changes result in a new version of the service contract. This approach does not support backwards or forwards compatibility.

Flexible Any incompatible change results in a new version of the service contract and the contract is designed to support backwards compatibility but not forwards compatibility.

Loose Any incompatible change results in a new version of the service contract and the contract is designed to support backwards compatibility and forwards compatibility.

Each strategy brings its benefits, but also drawbacks. Table 3.3 presents a general comparison of the three versioning strategies with respect to Strictness, Governance Impact and Complexity. The higher the strictness (of an approach), the lower its complexity. To ensure real backward compatibility, we rather should choose a strict versioning approach, but that will hardly be feasible for complex Web Service environments. It makes no sense to draft a new service contract each time a compatible or incompatible change occurs. Compatible changes (e.g. source code optimization) not necessarily lead to behavioural changes. In case of RWS, a flexible strategy is the most suitable strategy.

Evdemon [20] distinguishes *Message Versioning* (focuses on versioning the schemas used to describe messages processed by the service) and *Contract Versioning* (focuses on versioning the WSDL and contract information used to describe the service). Based on the two approaches, he recommends six design principles for versioning Web Services best practices:

1. Use `targetNamespace` to communicate major version releases.

2. Judicious use of unambiguous wildcards can minimize service versioning.

   That means to provide an extensible schema to meet changing service or consumer needs. The element `<xsd:any>` whicha can be used as wildcard element enables schemas to be extended in a well-defined manner.

3. Extensions must not use the `targetNamespace` value.

4. When adding new data structures, make them optional and add them to the end of service request messages.

   That means that existing consumers remain unaware of the new data structures.

5. Changing service response messages (other than type restrictions) are breaking changes that will require a new version of the service.

6. Adopt a one-to-one relationship between interface versions and UDDI tModels.

   To notify consumers about changes in the Web Service, Evdemon makes use of UDDI (a central Web Service registry) to communicate the changes. The tModel (technical model) is a data structure that represents a Web Service in UDDI. Evdemon suggests to provide a separate tModel for each Web Service version.

Bechara [4] presents different patterns according to the release type (*minor* or *major* release). He presents three different patterns to handle Web Service versioning. The *Consumer Binding* pattern recommends providers to inform consumers on service updates. But, the consumer is responsible for adapting the code to access the new service version. The *Layer of Indirection* pattern allows two minor releases co-exists without changing the consumers' code. A routing component forwards consumers' requests (based on the request content or address) to the service version required by this consumer. To overcome that problem, the author presents the *Adapter Pattern*. By adapting an old version request to new major version request the consumers' code does not need to be changed. This mediation layer for decoupling the consumer from the provider is shown with the help of the *Oracle Service Bus*.

Kaminski *et al.* [27] explore the question: What are desirable properties of an evolving Web Services. They present following six requirements as an outcome:

Backwards Compatibility A new service version must be backward-compatible to the previous one. At least the service interface must fulfill that requirement.

Common Data Store Common service states must be exposed to all clients regardless of which service version they are using. It is recommended to share a single datastore along multiple service versions.

No Code Duplication An important software engineering requirement is to avoid code duplication. That should be also common practice in a proper versioning strategy.

Untangled Versions Each piece of new code should be assigned to a specific service version. That should avoid deadcode-fragments and reducing the overall complexity. A typical cause e.g., is uncontrolled distribution of unfinished code.

Unconstrained Evolution Service evolution should be unconstrained by previous versions (as much as possible). Redesigning of the interface and refactoring of the source code should not be prohibited. But, this requirement is likely unachievable in practice. Especially because it conflicts with other recommended requirements.

Visible Mechanism Web Service frameworks should support the versioning process and keep it visible to the developer. Do not anticipate every possible action and avoid behind the scenes magic.

To fulfill the mentioned requirements, the authors introduce a freeze, adapt and delegate technique, which can be realized by the *Chain of Adapters* pattern. The idea behind: For any service updated which forces an interface evolution, a new seperate interface instance in combination with a tailored adapter is deployed. Every new interface adapts its previous version. That leads to a single service instance offering multiple service interfaces (see figure 3.1). The Chain of Adapters can be applied by service developers to achieve backward compatibility (at least for the service interface). This approach does not validate the backward-compatibility of the service behaviour. The scalability of that approach is also questioned.

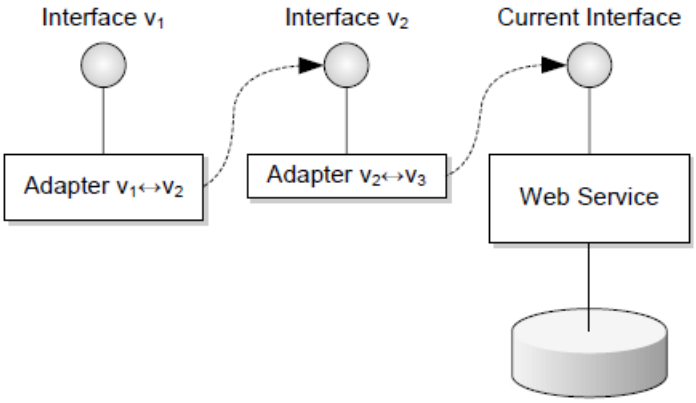

Figure 3.1: Chain of Adapters structure [27] after the second version have been published

WS-Temporal

Banati *et al.* [3] present the so called *WSDL-Temporal (WSDL-T)* approach for Web Service change management. They introduce two new attributes (*Validity* and *Timestamp*). Those attributes are used to extend relevant artifacts in a WSDL 2.0 document, like element, operation, endpoint. As the name says, the validity attribute gives information about the validity of the current element. It can have any value out of the validity set *latest*,*past*,*deleted*,*alwaysTrue*. *AlwaysTrue* denotes that, the artifact will be present in all the versions of the Web Service. The elements with this validity status form the basic functionality of the service and contracted as not changeable. The timestamp attribute captures the current datetime when the last change event affecting this element has happend. Apart from that, the scheme for naming of an element is modified. and version number is appended in the name with the delimiter # e.g. name#x.y.z where x.y.z denotes the version number according to the user defined version scheme. That allows multiple declared versions of e.g. an operation within a single WSDL document. The essence of WSDL-Temporal effort lies in enabling a single Web Service running at a given URI

absorbing changes. By extending the standardized WSDL to WSDL-T eases the management of various versions of a Web Service and allows access to various service versions at the same URL. Appendix (A.1 , A.2) provides a sample of a WSDL 2.0 file and how its instance of a WSDL-T file can look like. In a further effort [2] the authors also extend the BPEL specification for WSDL-T based Web Services.

Generic Web Services to support backward compatibilty

Beside the mentioned versioning strategies above, Borovskiy *et al.* [10] contribute a new method of preserving backward compatibility for the evolution of a Web Services. They introduce the concept of *Generic Web Services* (GWS) which is in its essence a rule for designing backward compatible service interfaces. A GWS consists of at least one generic operation. Such an operation possesses a specific relaxed signature. That means, its parameters are loosing restrictions on its type and value. That can be achieved by redesigning input parameters in contrast to common design style: Developers are encouraged to define method parameters in a highly significant way. (for example: *getPlayer(String firstname, String lastname, int age)*). When it comes to signature relaxation, input parameters are designed in a way that there exists two different kind of parameters: Identifying parameters and value parameters, which are controlled by the identifying parameters (for example: *getPlayer(String[] attributes, Object[] values)*). A generic operation must have at least one input parameter, which defines another. If a service has a generic interface, adding new features does not require any changes in its interface. So the interface will remain stable.

Best Practices for Artifact Versioning in Service-Oriented Systems

Next to Web Service versioning itself, Novakouski *et al.* [36] describe challenges of software versioning in service-oriented architecture environments. They name typical challenges in this domain. Which change types can appear? What are key artifacts which might be not considered during the software life cycle? By which granularity such artifacts should be versioned? Manging those changes become more complex, because of the increasing number of artifacts and the complexity of distributed service environments. However, the presented guidelines and recommendations aim to provide information how to apply version control policies to the various phases within the lifecycle of an SOA infrastructure. The full list of recommendations can be found in table A.4 of the appendix.

3.3 Summary of Web Service versioning approaches

In that chapter we introduce various challenges for a RWS design according to its version management. Based on an empirical study about the evolution of real life Web Service, the most common change scenarios are presented. Followed by the state-of-the-art and best practices of Web Service version management including various contributions and strategies how to handle the dynamic evolution of Web Services. Table 3.4 presents a summary of benefits and drawbacks

of the presented approaches according to our thinking of a proper Web Service versioning strategy which supports Web Service sustainability. As it shows, all approaches focus more or less only on Web Service interface versioning and how to handle the co-existence of various interface versions. Although some strategies consider backward compatibility they are all lacking of versioning the Web Service endpoint itself. Those contributions cover only functional changes to a certain extend, but completely lack of behavioural change detection.

Table 3.4: Summary of presented versioning approaches

	Benefits	**Drawbacks**	**Ref.**
Consumer Binding	Consumers get notified about service updates.	A new service is published for each update. Breaks backward compatibility immediately.	[4]
Layer of Indirection	Request-based message routing. Consumers binding code remains unchanged.	Additional routing component. Maintain consumer-version-mapping.	[4]
Chain of Adapters	Consumers binding code remains unchanged. Avoids code duplication.	Additional adapter needed for each version. Maintain various adapters and interfaces. Not all changes (e.g. operation deletion) can be hidden by an adapter.	[27]
WS-Temporal	Extend service artifacts with validity date. Multiple versions within the same interface (Consumers do not have to adapt their binding).	Multiple versions within the same interface (Interface gets needlessly blown-up which can lead to accidentally wrong consumer binding)	[3], [2]
Generic Web Services (GWS)	Enhance backward and forward compatibility of the interface.	Inaccurate method parameters make consumer binding more difficult. Sub-typing problem can occur.	[10]

Unfortunately, there exists not a single solution covering all aspects of the thesis's problem domain. Therefore, requirements are developed to address challenges according to Resilient Web Services. Various concepts (e.g. test methods, verify behavioural backward compatibility, consumer update notification) are introduced in the following chapter (4) to guide Web Service providers through the development and maintenance process of a Resilient Web Service.

CHAPTER 4

Resilient Web Service Framework

The various related approaches present in chapter 3 have their benefits, but still do not cover all aspects which we think are necessary for a proper RWS design. Following aspects are covered with little or no concern:

Traceability of computations According to ISO 900003 clause 7.5 traceability is stated as follows:

> *Throughout the product life cycle, there should be a process to trace the components of the software item or product. Such tracing may vary in scope according to the requirements of the contract or marketplace, from being able to place a certain change request in a specific release, to recording the destination and usage of each variant of the product.*

To apply that statement in our domain, traceability is the ability to identify and trace the evolution of the Web Service and its depending artifacts. RWS needs to support tracing of changes arising during the Web Service's life cycle.

Reproducibility of computations According to ISO 2004 *Repeatability* is:

> *A measure of variability derived under specified repeatability conditions. i.e. independent test results are obtained with the same method on identical test items in the same laboratory by the same analyst using the same equipment, batch of culture media and diluents, and tested within short intervals of time.*

To establish repeatability in the domain of Web Services, it is not sufficient to control only the Web Service source code. It is also necessary to control the surrounding environment by which a Web Service execution is effected. Both, the traceability and reproducibility of computations should be addressed and achieved by a proper RWS design.

4.1 The PictureService

Before we start discussing the various requirements for a RWS, we introduce an example service named PictureService. It is a Java based Web Service offering various operations applied to images. Each operation can be distinguished, either as *{stateless; stateful}* and *{deterministic; non-deterministic}*. In more detail, the service interface provides the following operations:

- *Image convertJpeg2Png(Image image);*
  {stateless; deterministic}
  Converts a given JPEG image to a PNG image.
  Since the outcome of the operation depends on nothing but the input, that operation is stateless. It is also deterministic, because it contains no randomness in its execution.

- *Collection searchPicturesForLocations(String... locations);*
  {stateless; non-deterministic}
  Retrieves a public collection of pictures according to the provided location parameters. Since the service itself do not maintain any state, the service can be classified as stateless. It is non-deterministic because locations tags can be added and removed from pictures dynamically.

- *Collection retrieveAlbum(String albumName);*
  {stateful; deterministic}
  Retrieves the private collection of the pictures containing in the album.
  Since the picture collection is maintained by the service it has an internal state. Uploading a new picture to the service, will affect both, the internal state and the picture collection. The operation is deterministic, because if the operation is requested with the same input parameter and the service itself is in the same state, its return will be identical.

- *Collection retrieveAlbumWithComments(String albumID);*
  {stateful; non-deterministic}
  Retrieves the private collection including public comments of all images containing in the album.
  Since the picture collection is maintained by the service it has an internal state. In comparison to the previous operation, this operation is non-deterministic. Because the result set also includes public comments by other users. Those comments are fetch by the service on request and are not maintained by it.

Thereby, the service makes use of the *Facebook4J*[1] API to connect to user profiles and retrieves the albums belonging to the user. It also utilizes *ImageMagick*[2] to offer the image convert functionality. In appendix-A.3 the generated WSDL of the PictureService is presented.

[1] `http://facebook4j.org/`
[2] `http://www.imagemagick.org`

4.2 Requirements for a Resilient Web Service

Realizing a RWS, following requirements must be addressed:

- *Versioning strategy* - A resilient versioning strategy and its application has to be ensured.
- *Provenance enriched responses* - The origin of a Web Service's response must be comprehensible for its consumers.
- *Enhance semantics of the Web Service interface* - A RWS must provide more semantic information about its operations.
- *Execute service operations in a sandbox* - A RWS has to provide a demo version for each operation which does not persist any changes in the environment of the service.
- *Dependency management* - A RWS owner must be aware of the dependencies of its service.
- *Notification support* - Any change in the service, either functional or behavioural has to be propagated to other resilient parties (e.g. consumers).

In the following, the various RWS requirements are presented in more detail.

4.3 Versioning policy

As mention in section 2.10, a RWS must guarantee not to break its backward compatibility with consumers. Since we do not focus on forward compatibility in the domain of RWS, we only classify between compatible and incompatible changes, where incompatibility means the absence of backward compatibility. To identify a Web Service's version we use a common version identification pattern like it is used by the Apache group[3]: *major.minor.point release* (e.g. 2.4.0)

- **major release** - incompatible changes are carried out by new major version number (e.g. deprecation of operations) (v1.3.3→v2.0.0)
- **minor release** - used to indicate compatible changes like adding a new service operation, or adding optional parameters to an existing operation (v1.3.3→v1.4.0)
- **point release** - least significant types of changes where no new functionality is added to the service. Such changes includes bug fixes or dealing with performance issues (v1.3.3→v1.3.4)

[3] `http://commons.apache.org/releases/versioning.html`

As a requirement for a RWS, each operation has to declare a *validity period* (see section-4.5). A new major release breaks that requirement. In such a case the RWS provider has to support multiple major releases concurrently, at least until the validity periods (see section 4.5) of the previous major version has been expired. To support multiple service instances in parallel, the major attribute of the version is represented in the service location. In the following sample the service has the following version: 2.3.1.

```
http://www.rws.com/pictureService-v2/PictureServiceService?wsdl
```

Whereas the complete version identification is declared as an attribute in the *service* element of the service's WSDL.

```
<service name="PictureService" version="2.3.1">
```

POLICY I The RWS provider has to follow the resilient versioning policy to ensure a resilient Web Service evolution.

4.4 Provenance aware computation

For Web Service consumers it must be comprehensible with which service version they are interacting. Therefore, the Web Service must provide somehow—among other attributes—its current version number. To provide automated provenance information, we propose to make use of the current source code revision number and the service release version. That is necessary to link from the service's release version to its internal source code version. By adding those information as meta data (e.g. in the header part) to a service response, a consumer can always verify which Web Service's version was used to process its request. The provenance information includes the following attributes:

- `Web Service release version` - Actual Web Service version identification (e.g. version 2.3.1). The version number is defined by the Web Service provider. It represents the Web Service evaluation.
- `VCS revision number` - Revision number of the current Web Service's source code file. In contrast to the Web Service release version, the revision number is generated and defined by the VCS.
- `Timestamp of last update` - Timestamp of the most current commit in the VCS. It presents the last update of the Web Service's source code. The format has to comply with RFC3339[4] (e.g. 2014-04-12T23:20:50.52Z). It can be used to determine the actual time of the Web Service's interaction.

[4] `http://www.rfc-editor.org/rfc/rfc3339.txt`

POLICY II To enable provenance tracking of a service response, the RWS provider has to attach provenance attributes, specifically the *Web Service release version*, *VCS revision number* and the *Timestamp of last update* to each SOAP response header.

4.5 Enhance semantics of the Web Service interface

WSDL describes a service in a functional syntax. It lacks any semantic element describing the service behaviour. There is also no attribute for the Web Service owner to publish its contact information within the service interface (WSDL). That fact makes it more difficult for a consumer to get in contact with a service owner. UDDI, which had been introduced to establish a service discovery platform on the World Wide Web, enables service providers to register its services and share its contact information. In UDDI the *businessEntity*[5] data structure reflects that information. The *contact* data structure is part of the businessEntity data structure and provides a way to imply contact information. Since the registration at a centralized UDDI compatible registry is optional and the maintenance often cumbersome, many providers do not register its services at all. But if that contact information is directly attached to the WSDL and represented in a standardized schema, it would be easier for providers to provide that information. However, we require for RWS owners to provide its contact information. Since it has to be compliant with the contact data structure specification of UDDI, we demand the following attributes to be present:

- personName - name of the person.
- useType - type of contact (e.g.“technical contact“,“hosting contact“).
- email - email addresses for the contact.
- phone - (OPTIONAL) telephone numbers for the contact.

It is recommended to bundle the contact information with the published service. The most common approach will be attaching the contact information directly to the service’s WSDL.

POLICY III A RWS provider has to attach its contact information—compliant to the UDDI contact data structure—to the public WSDL interface of its published services.

Apart from the missing contact information, any semantic information regarding a Web Service operation is also missing. By default, an operation definition is limited to its argument(s), return type and its naming. Any further information like file formats or operation classification is not available. In the scenario of RWSs, we require the following attributes as mandatory information for a Web Service operation:

[5] `http://www.uddi.org/pubs/DataStructure-V2.03-Published-20020719.htm#_Toc25130756`

- *Parameter exchange format* - In case the Web Service's operation uses not only simple arguments and return types (Integer, Double, ...), the exchange formats have to be defined in more detail. A proper definition of the exchanged file format is required to improve the consumers binding to objects returned by the Web Service. More precisely, the file format for string and binary data types has to be defined by making use of common file formats, which are either domain-specific or managed in a public registry. PRONOM, as an example of such a file format registry, encapsulates impartial and definitive information about the file formats required to support longterm access to digital records. The Pronom Unique Identifier (PUID) can be used to present the corresponding file format (e.g. Raw JPEG Stream $\rightarrow$ fmt/41).

- *Validity period* - Each Web Service operation has to be tagged with a so-called validity value representing a future timestamp (RFC 3339 format is proposed). The Web Service provider has to guarantee that the operation's functional and non-functional behaviour remains unchanged within that period of time, except in case of severe security issues or other forms of force majeure. In such a scenario required exceptional measures can be further elaborated in SLAs. Referring to table 3.2, only BW-compatible changes are permitted. That information helps business process owners to plan and choose a suitable operation for their purpose.

- *Operation classification* - In addition to the improved parameter description capabilities, a Web Service operation description must also reflect its operation classification. Referring to Web Service classification a detailed classification must be provided. In section 2.3 we have presented a very detailed Web Service classification schema by Dranidis *et al*. This thesis does not cover the *conversational* and the *transient* classification property. However, we think those two properties do not impact the resilience of a Web Service. The *private state* property is also not directly covered, as it presents no relevant information for a consumer. It is only relevant to distinguish a service behaviour between **{stateless;stateful}** and **{deterministic;non-determinstic}**.

This information has to be provided by the Web Service owner at deployment time. Annotations in the source code file can be a proper solution for providing that information in a structural form. Further details about the annotations and its application are provided in section-5.1.

POLICY IV The RWS provider has to enhance an operation description, by providing information about the parameter exchange format, supported validity period and the classification of each offered operation.

POLICY V The RWS provider has to maintain a Web Service operation for at least the guaranteed *validity period*.

4.6 Sandbox for Web Service operation executions

Testing is a major requirement for all resilient Web Services. Both, the Web Service provider and its consumers want to verify that the service is behaving as expected. Therefore, both parties require an approach to execute the service's operations without persisting any changes in the Web Service and its environment.

First, we want to pay attention to the common source code testing procedure.

Provide proper source code testing methods

Providing a RWS requires proper test methods to support verifying the repeatability of computations. For Web Service providers, it is mandatory to test the correctness of an operation's functionality during the development phase of the Web Service. A common quality metric for software testing is the *code coverage* metric. Code coverage is a measure of systematic software testing describing to which extend the source code is tested. The higher the test coverage value, the more possible forks of the code are accessed. To ensure a high coverage, the test case has to maximize the accessed paths in the operation's source code. Software testing distinguishes in general between white-, grey- and black-box testing, according to the degree of internal source code knowledge of the tester. The more defined a test set is, the greater assurance of software quality and reliability can be made. To improve the code coverage, the Web Service provider can use Random Testing (RT). RT is a fundamental testing procedure, which randomly selects test cases from the whole input domain. That approach can be used to enhance the possibility to pass every possible branch of method body. In theory, only if every possible path has been executed, a method is fully tested. Myers and Sandler [35] criticized that RT is a least effective testing method for using if you have little or no information about the software under test. But if you have that information (e.g. insights in the source code), like in our scenarios, RT is able to evenly spread the test cases over the input domain, which enhances its effectiveness. Chen *et al.* [13] name this approach Adaptive Random Testing (ART). Chen *et al.* [12] also showed that ART can achieve higher coverage on program structures. This is not only applicable for programs with numeric inputs, but also for those with non-numeric inputs. According to Cornett [15], 70-80% is an acceptable code coverage threshold for system tests.

POLICY VI A Resilient Web Service's source code must be tested till it exhibits a code coverage rate in between 70% - 80% or higher. The higher the rate, the lower is the possibility for any bugs or other uncertainties. A stable, faultless implementation will prevent the Web Service provider from ongoing update procedures.

Since testing methods often trigger state changes or modify data tuples, a rollback strategy has to be applied. That means, that after each test method execution, all conducted transactions get discarded and not persisted. Such a rollback action is common practice in modern source code testing frameworks like JUnit.

POLICY VII For each test method, the RWS provider has to implement a rollback strategy to avoid the persisting of any state changes or data modifications.

It is the purpose of those source code testing methods to cover all possible behavioural patterns of an operation. That also includes throwing errors intentionally and test the operation's exception handling. Such testing scenarios are mandatory for the Web Service provider. Those testing methods are usually deposit next the actual Web Service source code file. They are targeted only for internal use during the development phase and therefore they are not published via the WSDL interface. However, for Web Service consumers it is also necessary to test the Web Service behaviour, but within another setting. Consumers typically are not interested in internal operation execution procedure, but in its computational results. Therefore the Web Service provider has to provide additional demo operations which get executed in a sandbox. A sandbox for testing the service behaviour is necessary in order to not trigger any changes in the life system.

Provide operations in sandbox mode

For Web Service consumers, it is important to execute a Web Service's operation at runtime in order to make a decision if the service functional behaviour meets their business needs. Therefore, the RWS provider has to decidedly provide a demo method for each service operation. The purpose of such a *demo operation* is to execute the real service behaviour without persisting neither any internal state changes, nor any state changes in its surrounding system. A major concern to address is to mock the internal state (e.g. operations are often only executable in a certain state) and data write access (e.g. prohibit any modifications in common data stores) of the Web Service. Since the response of a stateless operation only depends on the request data, no additional demo operations is necessary.

POLICY VIII For each stateful Web Service operation, the Web Service provider must offer a demo operation mode, which performs the original functionality, without persisting any state changes. By convention, the signature of the demo operation must be equal to the original operation including a *Demo* suffix for the name. Those additional demo operations must be provided by the RWS interface. The service owner has to provide sample request for each demo operation. Those requests are used in a later point in time create the Web Service View Path (see section 4.7).

For the *retrieveAlbumWithComments* operation, figure 4.1 depicts the additional methods which have to be provided by the RWS owner. The RWS owner has to provide a proper amount of internal testing methods to fulfil POLICY 6. To indicate such an internal test method, it has to start with a *test_* prefix in the name. Those methods are not available through the service interface. Its purpose is to test the service's functionality during the development phase. Additional, the RWS owner has to provide a demo operation for each operation. It is recommended to add a *Demo* suffix to its name. The demo operations have to be available in the service interface. Additionally the service provider has to provide input requests for the demo operations. At a later point in time, those requests are used to collect the Web Service's View Path.

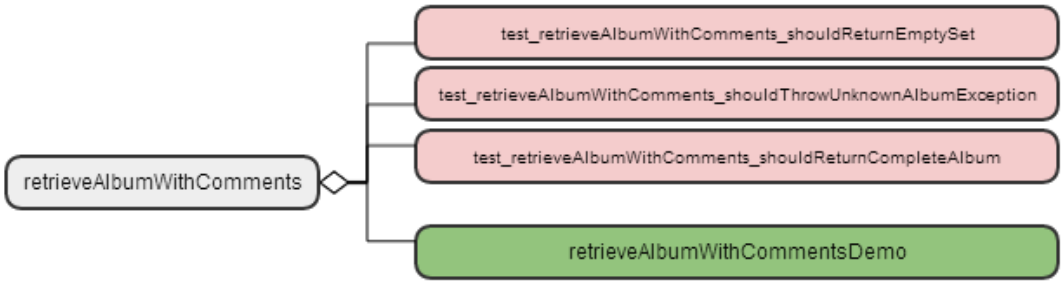

Figure 4.1: A resilient operation *in aggregation with* its multiple test methods and the single demo method

Table 4.1: Two possible View Paths for the *convertJpeg2Png* operation of the PictureService

	View Path #1	*View Path #2*
Data formats	Raw JPEG Stream (fmt/41);Portable Network Graphics (fmt/13)	Raw JPEG Stream (fmt/41);Portable Network Graphics (fmt/13)
Application	ImageMagick 6.8.9-7 Q16 Microsoft Visual C++ 2010	ImageMagick 6.8.9-7
JVM	Java SE 6 Update 45	Java SE 7 Update 10
Operating System	Windows 7 Enterprise SP1	OS X 10.9.4
Hardware	3,3GHz Intel Core i3 8GB 1600MHz DDR3 NVIDIA GT630 2GB	2,3GHz Intel Core i5 4GB 1333MHz DDR3 Intel HD Graphics 3000 384MB

4.7 Web Service dependency management

In case of developing *Resilient Web Services*, we are not only interested in the syntactical description of a Web Service interface (WSDL). A RWS must also provide information about its hardware and software dependencies. That means a RWS must be aware of its View Path. In the following we present an example depicting the effect of two divergent View Paths: According to the *convertJpeg2Png* operation of the *PictureService*, which converts a JPEG file to a PNG image, we present in table 4.1 two different View Paths. The left column represents a typical Windows environment, the right column a common OS X setup.

What can be the result of two varying view paths is presented in figure 4.2. The upper image shows the input file which gets converted from JPEG to PNG by ImageMagick[6]. That process has been executed within both, the execution environment of View Path #1 and View Path #2. Subsequent, ImageMagick is used to compare the two converted images. The resulting *diff* is

[6] `http://www.imagemagick.org`

Figure 4.2: Used two different view paths to convert the upper image from JPEG to PNG and makes a diff (lower image), which is showing the deviations.

presented in the lower image. The red points identify deviations between both converted images.

Since a divergent view path can result in a different computation result, a proper RWS dependency management is required. A RWS dependency management includes the following aspects:

1. Capture dependencies → Create the Web Service's view path
2. Monitor dependencies for changes → Detect changes of the view path
3. Propagate changes to recipient → Notification support

Table 4.2: Responsible party for collecting the dependencies with respect to the chosen service model

	Hardware Dep.	*Software Dep.*	*Source Code Dep.*
IaaS	x		
PaaS	x	x	
In-house hosting	x	x	x

Capture dependencies

To reveal all Web Service dependencies, figure 4.3 presents possible dependencies to its level of insight. The figure is four-folded (minor overlappings included): In the first container a high-level view from the viewpoint of a service consumer is given. As already mentioned, only the WSDL is visible for the consumer. In the second container we present a look underneath the Web Service's interface. It highlights the dependencies of a Web Service itself according to additional included Web Services, software components and shared data resources. The third container details the above view by providing possible environmental dependencies of a service. Those include e.g. reference software libraries, invoked system programs or application server dependabilities. Also hardware dependencies can be fundamental for proper Web Service execution. Therefore, in the last container, we present relevant hardware artifacts. Next to the standard hardware artifacts like CPU, Harddrive and Memory, also not so popular artifacts like GPU or any external devices like sensors can become a hardware dependency.

As shown in figure 4.3, we have to distinguish between *software* and *hardware dependencies*. The software dependencies collection encompasses all software artifacts, which are needed to successfully execute a specific Web Service operation from the platform independent Web Service interface to the core Operating System libraries. The hardware dependencies collection comprises all hardware artifacts, which are used in the processing a Web Service request.

The big challenge of the identification process is to ensure that none of the required dependencies is missing. To assure a preferably complete set, it is recommended to execute the obliged *Demo methods* (see section 7). It is a design requirement to separate the identification process from the actual Web Service to its hosting system, because it can be resource intensive and time consuming (see limitations of the framework in section 6.4). To capture the Web Service's View Path is the task of the platform on which the RWS is hosted. It has to provide tools for automated collecting the software and hardware artifacts of the hosted Web Service.

The responsibility for collecting the specific information depends on which service model has been chosen by the RWS provider. Table 4.2 presents the responsibilities for collecting the dependencies with respect to the chosen service model. In a scenario, in which the service provider outsources the hosting IT infrastructure and consumes it as IaaS, the IaaS provider is the responsible party for collecting the hardware dependencies. If the service provider chooses to consume the hardware infrastructure and software platform as a service, the PaaS provider has to capture all relevant dependencies. In a traditional IT environment (In-house hosting), the service provider has to carry out the dependency management on his own.

As a side note: The automated identification process for software dependencies is a highly dynamic approach. Therefore, it is hard to assure that all dependencies are correctly captured.

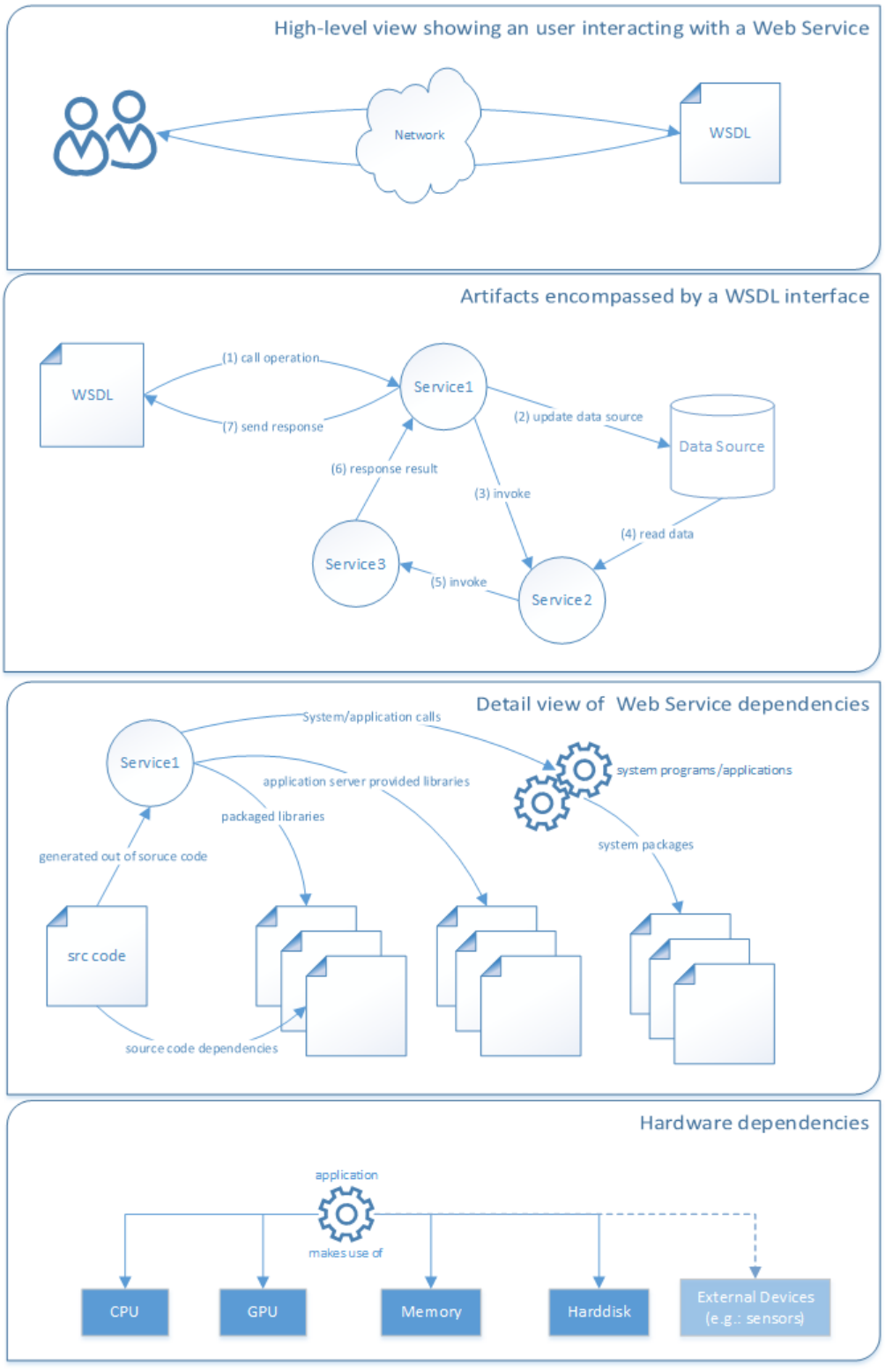

Figure 4.3: Web Service dependencies on different levels

Missing dependencies will lead to an incomplete dependency collection. To complete the dependency collection, one possibility is to add required dependencies manually by expert knowledge.

It is also required to make those dependencies explicitly available. Therefore, we opted OWL as representation format. A major criteria for choosing OWL is that the collected View Path can contain several thousand dependencies which is nothing but a linked data connection. Other benefits using OWL as the representation format are follows: OWL defines convenient mechanisms for reasoning (e.g. SPARQL); Easy to use *Java API and reference implementation*[7] for creating, manipulating and serializing OWL Ontologies; It exists tools for comparing ontologies.
Within the TIMBUS[8] research project a *process context model* has been developed. It defines the generic concepts related to the process context and its dependencies in a domain-independent ontology (DIO). These concepts can be redefined and mapped to a domain-specific ontology (DSO). The context model is one approach to make the Web Service's View Path explicit available.

Binder *et al.* [8] present an approach for capturing the process context model. By utilizing `strace`—which allows an interception of the system calls—they present a tool for capturing the context model for a certain process execution. More details about the application of the tool are present in section 5.5.

POLICY IX The service hosting platform provider is responsible for capturing the Web Service's View Path.

Subsequent to the capturing process, it is required to permanently monitor those artifacts for modification. Such a modification includes file updates, file deletion and also additional created files. A dependency modification during the Web Service's *run phase* can cause sever consequences; including divergent computations, divergent QoS values, and service unavailability. Therefore, it is important to detect changes in the serivce's dependencies in-time. Changes of depended artifacts can be caused by various sources.

- Upgrade or replace hardware resources like CPU or GPU
- Install update packages for the running operating system
- Upgrade the application execution environment
- Falsely removed dependent application
- ...

An in-time detection can only be guaranteed by a permanent monitoring approach. The party which is responsible for the capturing is also responsible to detect changes on those dependencies. However, such a permanent monitoring is resource consuming. Furthermore, it is not necessary to detect every file modification that happens (e.g. write to a logging file, remove

[7] `http://owlapi.sourceforge.net`
[8] `http://www.timbusproject.net`

a temporary file). Therefore, we present an alternative approach. The basic concept can be described as follows: A notable modification of the view path can only be introduced by the platform provider through an update process (e.g. update Java version 1.6 to 1.7, install security batch). Thus, the view path modification detection only has to be triggered in combination with an update event. To test if a certain context model is affected, RDF query languages like SPARQL can be used to detect if the context model contains a certain individual. In case the context model is affected, a notification about the update has to be forwarded to the service owner like discussed in the next section. The service owner has to evaluate, whether the service behaviour has changed. In case of a detected changes, the service owner has to forward a notification to the service's consumers. This concept is also modeled as a sequence diagram presented in figure 4.4.

POLICY X The service hosting platform has to provide a push notification mechanism to inform those events to Web Service users as well as providing an interface to allow them to pull the respective informations.

Propagation of dependency changes

In case of a modification has been detected, it is necessary to propagate that change to the depended RWS parties. To ensure interoperability across various RWS parties, a common exchange format has to be defined. Since the View Path is represented in OWL, it is reasonable to propagate changes by utilizing the PREMIS[9] OWL. It is an ontology based on the PREMIS *Data Dictionary for Preservation Metadata*, which is a digital preservation standard based on the *OAIS*[10] reference model. Mayer *et al.* [33] present an approach for a software update scenario by utilizing the *Event* entity concept of PREMIS. In more details, an update of a software package is presented by so called *SoftwareReplacement* event. It is used to link individuals by *linkingSourceObject* and *linkingOutcomeObject* properties, representing the change. Figure 4.5 depicts such a *SoftwareReplacement* event by means of a Java version update. Since the RWS design also requires to propagate hardware changes, we have designed a *HardwareReplacement* event similar to the *SoftwareReplacement* event. As an example for the *HardwareReplacement* event, figure 4.6 shows the replacement of the current Intel CPU Q9300 with the more powerful Q9650 instance.

POLICY XI The service hosting platform has to propagate occurred changes in the following format. For software updates the *SoftwareReplacement* event has to be used. In case of a hardware update the *HardwareReplacement* event has to be used.

As aforementioned the number of resilient parties can vary depending on the service model chosen by the service owner. Figure 4.7 presents who is responsible for propagating the changes to whom according to the various service models and communication paths.

[9] `http://www.loc.gov/standards/premis`
[10] `http://www.iso.org/iso/catalogue_detail.htm?csnumber=57284`

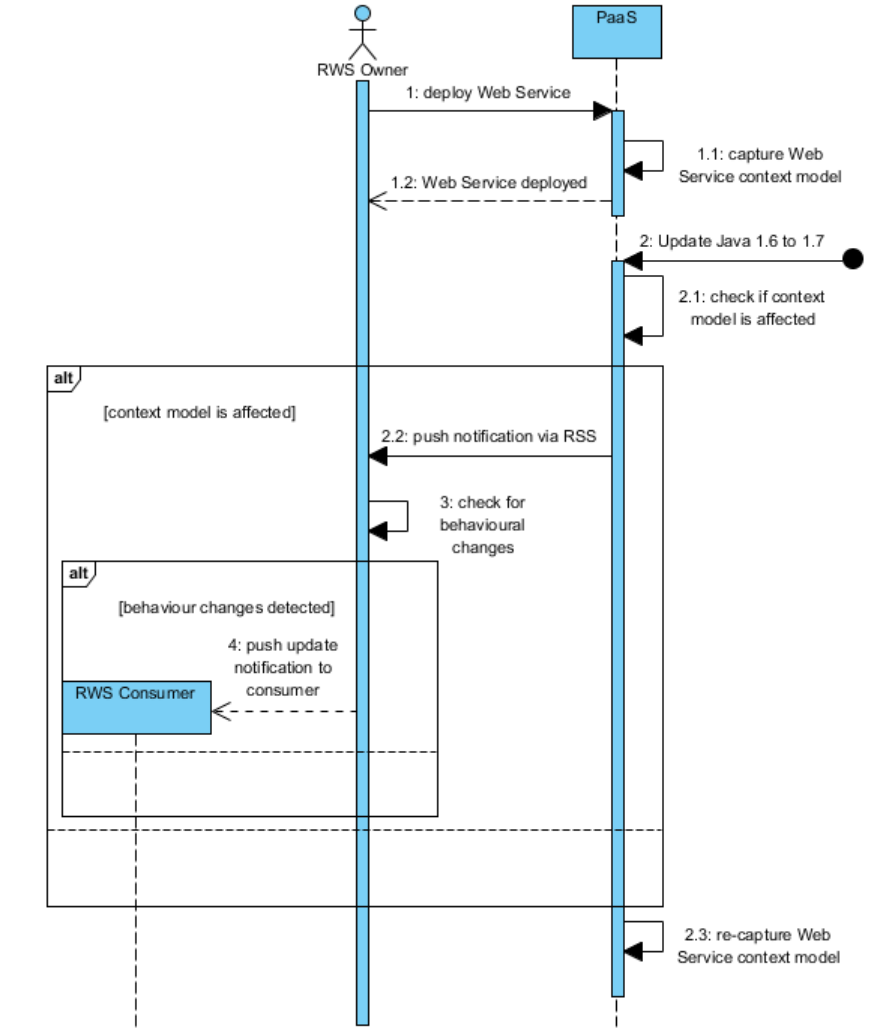

Figure 4.4: Concept for reacting on dependency updates

```
<ClassAssertion>
  <Class IRI="http://id.loc.gov/ontologies/premis.rdf#Event"/>
  <NamedIndividual IRI="[serviceLocation]/[identifier]/SoftwareReplacement"/>
</ClassAssertion>
<ObjectPropertyAssertion>
  <ObjectProperty IRI="http://id.loc.gov/ontologies/premis.rdf#
      linkingSourceObject"/>
  <NamedIndividual IRI="[serviceLocation]/[identifier]/SoftwareReplacement"/>
  <NamedIndividual IRI="[originalModelURI]#OracleJava1.6.u44"/>
</ObjectPropertyAssertion>
<ObjectPropertyAssertion>
  <ObjectProperty IRI="http://id.loc.gov/ontologies/premis.rdf#
      linkingOutcomeObject"/>
  <NamedIndividual IRI="[serviceLocation]/[identifier]/SoftwareReplacement"/>
  <NamedIndividual IRI="[serviceLocation]/[modifiedModelURI]#OpenJDK1.7.u65"
      />
</ObjectPropertyAssertion>
```

Figure 4.5: Description detailing the changes made to the system by replacing *Oracle Java* version 1.6 with 1.7

```
<ClassAssertion>
  <Class IRI="http://id.loc.gov/ontologies/premis.rdf#Event"/>
  <NamedIndividual IRI="[serviceLocation]/[identifier]/HardwareReplacement"/>
</ClassAssertion>
<ObjectPropertyAssertion>
  <ObjectProperty IRI="http://id.loc.gov/ontologies/premis.rdf#
      linkingSourceObject"/>
  <NamedIndividual IRI="[serviceLocation]/[identifier]/HardwareReplacement"/>
  <NamedIndividual IRI="[originalModelURI]#Intel-Core(TM)2-Quad—CPU——Q9300
      —@-2.50GHz"/>
</ObjectPropertyAssertion>
<ObjectPropertyAssertion>
  <ObjectProperty IRI="http://id.loc.gov/ontologies/premis.rdf#
      linkingOutcomeObject"/>
  <NamedIndividual IRI="[serviceLocation]/[identifier]/HardwareReplacement"/>
  <NamedIndividual IRI="[serviceLocation]/[modifiedModelURI]#Intel-Core(TM)2-
      Quad—CPU——Q9650—@-3.00GHz"/>
</ObjectPropertyAssertion>
```

Figure 4.6: Description detailing the change of the CPU by replacing the *Intel Q9300* with *Q9650*

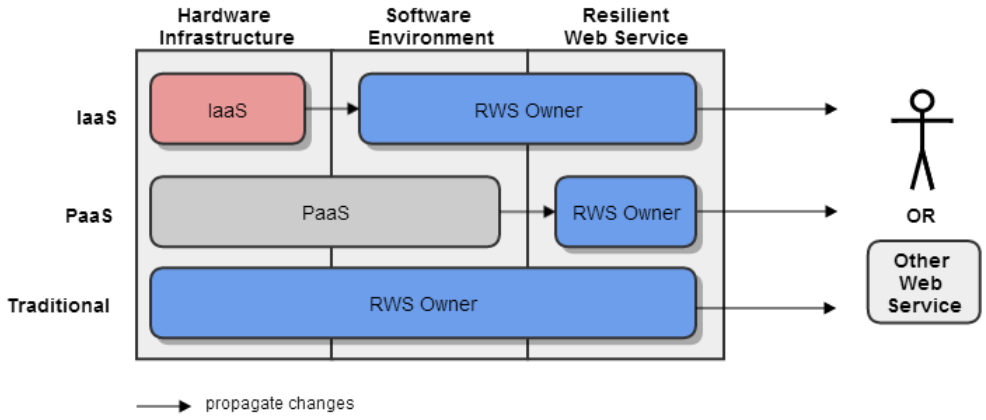

Figure 4.7: Communication paths between resilient parties

Since in a pull strategy the consumer is responsible for fetching changes, an urgent notification can be delayed. That can harm the correct Web Service execution even before the provider gets notified about it and is able to react on it. Therefore, it is recommended to use a push strategy for propagating changes in a resilient environment. However, the implementation of a push mechanism involves usually a higher effort than a pull approach. One simple solution for a push style implementation approach can be realized by *RSS*. By utilizing the *channel* concept of RSS, a separate channel can be established for each communication path.

Depending on the communication path, the content of the feed varies. Think about the following scenario: The service makes outsources the service platform to a PaaS. The PaaS provider is forced to update Java to the new release version. Subsequent to the update process, the PaaS provider has to push a notification according to POLICY 11 to a certain RSS channel notifying the PictuerService owner. The service owner himself now has to detect which of the operations are affected by the updates. In case of the *retrieveAlbum()* operation behaving differently than expected (e.g. the sequence of the images in the collection has changed), the owner has to notify its clients.

Discussion:
Since RSS is a push style notification approach, an update notification can be automatically fetched and processed. According to the presented scenario above, tools like *rsstail*[11] are able to listen for updates on a certain RSS channel. We can utilize such a tool to trigger an execution of the demo operations automatically. In case of a change in the execution behaviour is detected, a further step will be to automatically create a report summarizing which operation remains unchanged and which behaves differently. That report can be automatically forwarded to the service owner and its consumers without any timing delay. Subsequently to this instant notification, both the service owner and the affected consumers can elaborate their further necessary actions.

[11] `http://www.vanheusden.com/rsstail`

```
<rss version="2.0">
   <channel>
    <title>PaaS updates related to the PictureService</title>
    <link>http://paas.com/services/pictureservice/rss</link>

    <item>
      <title>Java version 1.7</title>
      <description>Updates Java from version 1.6 to 1.7</description>
      <link>http://resilient.com/PictureService/rss/5</link>
      <author>PaaS Administrator</author>
      <guid>5</guid>
      <pubDate>2014-04-12T23:20:50.52Z</pubDate>
    </item>

    <item>
      ...
    </item>

  </channel>
</rss>
```

Figure 4.8: RSS feed channel hosted by the PaaS provider.

Figure 4.8 presents a RSS feed channel between the PaaS provider and the service owner. For each update, a new *item* gets added. It includes a *link* element holding an URL linking to the PREMIS description of the change.

POLICY XII Update notifications have to be pushed via some channel (recommended is RSS) to subscripted consumers. Each resilient party has to establish a separate RSS channel for each resilient service.

4.8 Remote Web Service dependencies

Additional to the already mentioned aspects, there is the aspect of remote Web Service dependencies as presented in figure 4.3. In case of a RWS (service A) depends on another Web Service (service B), it is mandatory that service B must also be resilient. Furthermore, the guaranteed validity period of service A (=its lifetime) must not exceed service B's validity period. Otherwise, service A can not guarantee resilience. To mitigate the risk of a remote software component to become unavailable, which can have severe impact on the Web Service execution and—in the wider sense—to process execution, a *Software Escrow* system can be used. By introducing a legal third party, the *Escrow Agent*, in-between the consumer and the developer of a software component, it establishes a trustworthy way to deposit software. The escrow agent itself is responsible for depositing the software and realizing by a contracted trigger event. In addition to a standard usage and maintenance license between a consumer and a developer, both parties have

to sign an escrow contract between them and the escrow agent. Such a contract considers on the one hand technical issues and also legal issues. Leading technical issues are: completeness and quality of the deposited material. What are the necessary source code files, libraries , datasets or compile instructions. On the legal side, important issues are the usage rights for deposited third party libraries or to specify the event that triggers the release of the deposited material. Weigl *et al.* [44] introduce a software solution for supporting the escrow agent by handling the mentioned technical issues. As a subpart of that tool, they are able to detect remote calls by analyzing the source code of a software artifact.

POLICY XIII In case of a RWS depends on another Web Service, that remote service must also be resilient.

POLICY XIV The maximal lifetime of the Web Service must not exceed the maximum lifetime of any of the remote services it is depending on.

4.9 Summary of the Resilient Web Service Framework

In that chapter we present requirements necessary to address RWS challenges. Various concepts (e.g. demo operations, view path capturing, consumer push notification) are introduced to guide Web Service providers through the development and maintenance process of a RWS. A major issues addressed in this chapter are the challenges according to a proper Web Service dependency management. Maybe the most critical challenge is a complete identification of all Web Service dependencies. Therefore, we present the view path concept, which can be used to describe a Web Service dependency collection. It encompasses software and hardware artifacts of a Web Service's execution environment. We propose the Web Service's View Path identification by executing the required demo operations provided by the Web Service owner. Once the identification has been successfully, a proper dependency modification tracking strategy has to be applied. If a modification is detected, is has to be validated if the Web Service remains unchanged in its function and behaviour. Therefore, current Web Service's execution results are compared to previous captured results.

In the following we present a compact overview of all resilience policies introduced in this chapter.

POLICY 1 The RWS provider has to follow the resilient versioning policy to ensure a resilient Web Service evolution.

POLICY 2 To enable provenance tracking of a service response, the RWS provider has to attach provenance attributes, specifically the *Web Service release version*, *VCS revision number* and the *Timestamp of last update* to each SOAP response header.

POLICY 3 A RWS provider has to attach its contact information—compliant to the UDDI contact data structure—to the public WSDL interface of its published services.

POLICY 4 The RWS provider has to enhance an operation description, by providing information about the parameter exchange format, supported validity period and the classification of each offered operation.

POLICY 5 The RWS provider has to maintain a Web Service operation for at least the guaranteed *validity period*.

POLICY 6 A Resilient Web Service's source code must be tested till it exhibits a code coverage rate in between 70% - 80% or higher. The higher the rate, the lower is the possibility for any bugs or other uncertainties. A stable, faultless implementation will prevent the Web Service provider from ongoing update procedures.

POLICY 7 For each test methods, the RWS provider has to implement a rollback strategy to avoid the persisting of any state changes or data modifications.

POLICY 8 For each stateful Web Service operation, the Web Service provider must offer a demo operation, which performs the original functionality, without persisting any state changes. By convention, the signature of the demo operation must be equal to the original operation including a *Demo* suffix for the name. Those additional demo operations must be provided by the RWS interface. The service owner has to provide sample request for each demo operation. Those requests are used in a later point in time create the Web Service View Path (see section 4.7).

POLICY 9 The service hosting platform provider is responsible for capturing the Web Service's View Path.

POLICY 10 The service hosting platform has to provide a push notification mechanism to inform those events to Web Service users as well as providing an interface to allow them to pull the respective informations.

POLICY 11 The service hosting platform has to propagate occurred changes in the following format. For software updates the *SoftwareReplacement* event has to be used. In case of a hardware update the *HardwareReplacement* event has to be used.

POLICY 12 Update notifications have to be pushed via some channel (recommended is RSS) to subscripted consumers. Each resilient party has to establish a separate RSS channel for each resilient service.

POLICY 13 In case of a RWS depends on another Web Service, that remote service must also be resilient.

POLICY 14 The maximal lifetime of the Web Service must not exceed the maximum lifetime of any of the remote services it is depending on.

Nevertheless, we want to conclude that chapter by some critical remarks.

Demo operations are hard to implement Providing a RWS, it requires to provide demo operations which do not alter the Web Service's state or do not leave any traces in its environment. Due to the complex Web Service interaction scenarios including other artifacts (e.g. another Web Service, database, external device) it becomes very challenging and time consuming to provide intelligent demo operations. Therefore, a framework supporting an intelligent rollback strategy is required.

Validation of Web Service responses are hard to automate To validate an unchanged functionality of a Web Service operation, one way is to compare the current response to previous responses. However, that can become very challenging because of non-deterministic operations and complex return types (e.g. images, pdf reports). It is hardly feasible to computational automate that validation process for complex deterministic results. Also its reliability would only be verifiable to a certain extend. For a proper response validation—either for backward-compatibility changes or in any other scenario—human judgment is indispensable.

Complete and correct identification of the Web Service's view path The theoretically concept of the view path is easily applicable for a Web Service execution environment. The practical implementation of that concept is much more complex. Firstly, automatic dependency identification is only applicable to some extend. Only those dependencies, which are accessed by the execution of the demo operations can be identified. That means that the quality of the provided Web Service demo operations is a crucial factor. Secondly, it is hardly feasible to verify the correctness of the identified view path. Therefore, a critical review of the automatically identified View Path by experts is unavoidable.

CHAPTER 5

Resilient Web Service Framework Tools

That chapter presents the practical part of this thesis. More precisely, the RWS framework consists of two parts. Firstly, the resilience annotations which have to be used by the Web Service provider to add the required semantics to the Web Service source code file. Secondly, the framework encompasses a tool box including:

- A tool for the verification of the provided annotations
- A tool for the identification of the Web Service's View Path
- A tool to transform the semantic enhanced source code file into a semantic enhanced service interface

In the upcoming sections, every particular contribution is presented in more detail.

5.1 Resilience annotations

The resilience annotations supports three types of custom Java annotations[1] to enhance the Web Service's definition and its provided operations. Firstly, Java class level annotations to identify and describe that class as a RWS. Secondly, a set of method annotations used to describe the operations more precisely. Thirdly, a parameter annotation which has to be used to define the message exchange formats. Table 5.1 lists and describes the developed custom annotations which are available at development time.

Applying those annotations on our non-resilient *PictureService* version is depicted in listing 5.1.

[1] `https://jcp.org/en/jsr/detail?id=250`

Table 5.1: Summary of the resilience annotations offered by the Resilient Web Service Framework

Annotation	Description
Class level	
`@Resilient(version)`	Declare a class as a RWS and identify its current release version.
Method level	
`@Stateful`	Annotate a stateful Web Service method.
`@Stateless`	Annotate a stateless Web Service method.
`@Validity(endDate)`	Used to set the validity by which the Web Service Provider guarantees an unchanged execution behaviour.
`@Return`	Declare the return file format. It can be used in case the return type exceeds a simple data type.
`@Demo`	Declare an operation as a demo operation.
Parameter level	
`@FileFormat`	Used to identify the file formats, encoding types of method parameters.

Listing 5.1: Java source code file with the additional resilience annotations

```
1  import com.resilient.annotations.*;
2  ...
3
4  @WebService
5  @Resilient(version = "2.4.0")
6  public class ResilientPictureService implements IPictureService {
7
8          @WebMethod
9          @Stateless
10         @Deterministic
11         @Validity(endDate = "2015-01-01T23:59:59.00Z")
12         @Return(format="fmt/13")
13         public Image convertJpeg2Png(
14                                 @FileFormat(format="fmt/41") Image image)
15                                 throws PictureServiceException {
16                 ...
17         }
18         ...
19         @WebMethod
20         @Demo
21         @Stateless
22         @NonDeterministic
23         @Validity(endDate = "2015-01-01T23:59:59.00Z")
24         public List<URL> searchPicturesForLocations(String... places) throws
               PictureServiceException {
25                 // for demonstration purpose
26         }
27 }
```

The none-bold annotations like @WebService (#4) and @WebMethod (#8,19) are the standard Java annotations used to declare a class as a Web Service and a method as an Web Service operation. The bold annotations represent the resilience annotations according to table 5.1. To declare that Web Service as a RWS, we add the **@Resilient** (#5) annotation on class level. The **@Resilient** annotation also includes the current version identifier provided by the Web Service owner. Since the *convertJpeg2Png* operation has an image data type as parameter and also an image return type, the operation has to declare both, the **@FileFormat** (#14) annotation for the parameter and the **@Return** (#12) annotion for the return type. Next to the *convertJpeg2Png* we present the declaration of the demo version of the *searchPicturesForLocation* operation. Except of the **@Demo** annotion (#20), the declaration of the real *searchPicturesForLocation* will be equal.

5.2 Functional compatibility verification tool

We have developed a Java based tool for verifying both, the correct application of the resilience annotations and the functional compatibility against the previous version. The comparison is done on source code level. In a first step, the correctness of the applied annotations is verified. In more detail, the tool checks the following:

- In case an operation parameter type is a non-trivial data type (e.g. base64Binary) the *@FileFormat* annotation must be present.
- For each stateful operation there must exist a demo operation according to the naming convention.
- The timestamp of the validity annotation posses a date in the future.

In a second step the current annotation values are compared to its previous version values. In case of the current annotations of an operation differs from its previous ones, the commit gets also aborted. (e.g. @Return(format="fmt/13") $\nrightarrow$ @Return(format="fmt/154")). A sample outcome of the tool is presented in figure 5.1. The tool reveals in total two violations. Due to its violation report, the service owner can easily fix the issues.

Subsequently, the tool fetches the last committed version of the service from the VCS. In case of the applied changes in the service source code are incorrectly reflected by the version (see section-4.3) the commit gets aborted. (e.g. an operation has been deleted, but the new version identifier only reflects a minor change v1.3$\rightarrow$1.4 instead of $\rightarrow$2.0)

Usage The purpose of the verification tool is to deny commits to the VCS which violate the resilience annotation policy. Therefore, we have developed a pre-commit-hook which executes the verification tool. In case the tool reports an error, the commit gets aborted.

```
trying to fetch file ResilientPictureService
start verification for file ResilientPictureService.java

(0) Class level annotations => valid

(1) Method level annotations =>
  @Validity annotation is missing for method 'retrieveAlbumWithComments'.
  @FileFormat annotation is missing for param 'image' of method '
      convertJpeg2Png'.

_____________________________________________________________

In total 2 validation violation(s) found!
```

Figure 5.1: Sample result of the annotations verification tool

5.3 Java2RWSDL converter

As described in the requirements, it is essential to expose the annotations also in the WSDL file. Therefore, we make use of the Apache Axis2 framework. We adjust the standard Java2WSDL component by adding attributes which represent the resilience annotations and their values. The tool adds the *behaviour={stateful, stateless};{deterministic, non-deterministic}* and *validity=timestamp* attribute to the existing *operation* element. In case of the operation is a demo operation, an additional *demo="true"* attribute is added. The *@FileFormat* annotation becomes part of the input and output message type. Listing 5.2 presents an excerpt of the resulting *resilient WSDL (RWSDL)*. The bold elements are the additional elements which reflects the resilience annotations.

Listing 5.2: Excerpt of the resilient WSDL including the values of the resilience annotations (marked in bold)

```
<xs:complexType name="convertJpeg2Png">
<xs:sequence>
<xs:element name="arg0" type="xs:base64Binary" minOccurs="0" fileformat="fmt
    /41" />
</xs:sequence>
</xs:complexType>
<xs:complexType name="convertJpeg2PngResponse">
<xs:sequence>
<xs:element name="return" type="xs:base64Binary" minOccurs="0" fileformat="
    fmt/13"/>
</xs:sequence>
</xs:complexType>
...
<operation name="convertJpeg2Png" behaviour="stateless;deterministic"
    validityDate="2015-01-01T23:59:59.00Z">
<input wsam:Action="http://resilient.com/ResilientPictureService/
    convertJpeg2PngRequest" message="tns:convertJpeg2Png"/>
<output wsam:Action="http://resilient.com/ResilientPictureService/
    convertJpeg2PngResponse" message="tns:convertJpeg2PngResponse"/>
```

```
<fault message="tns:PictureServiceException" name="PictureServiceException"
    wsam:Action="http://resilient.com/ResilientPictureService/convertJpeg2Png
    /Fault/PictureServiceException"/>
</operation>
...
<operation name="convertJpeg2PngDemo" behaviour="stateless;none-deterministic
    " validityDate="2015-01-01T23:59:59.00Z">
<input wsam:Action="http://resilient.com/ResilientPictureService/
    convertJpeg2PngDemoRequest" message="tns:convertJpeg2PngDemo"/>
<output wsam:Action="http://resilient.com/ResilientPictureService/
    convertJpeg2PngDemoResponse" message="tns:convertJpeg2PngDemoResponse"/>
<fault message="tns:PictureServiceException" name="PictureServiceException"
    wsam:Action="http://resilient.com/ResilientPictureService/
    convertJpeg2PngDemo/Fault/PictureServiceException"/>
</operation>
...
<service name="PictureService" version="2.3.1">...</service>
```

5.4 Provenance enriched SOAP header

For a Web Service consumer it is important to know with which service version it is interacting. To that purpose, our framework automatically adds provenance information to each Web Service response. After a request has been processed, the framework attaches the attributes `releaseVersion`, `sourceCodeRevision`, `lastUpdate` as header information to the response. Because the header information is meta data, the response payload does not get altered. Therefore, the framework makes us of the interceptor concept[2] of the Java Enterprise Edition (EE) specification. Interceptors are used to intercept a current process at a specific point in its life-cycle (e.g. interceptor is triggered before Web Service operation is invoked). The custom provenance interceptor of the framework gets triggered after the Web Service operation has been executed and adds the provenance information to the response header. Figure 5.2 presents a provenance enriched response generated by framework. The attached provenance elements are visualized in bold.

5.5 Capture Web Service's View Path

We have developed a script based tool, which can be used to trigger Web Service operations by utilizing *cUrl*. *cUrl* is a command line tool for sending and receiving files using URL syntax. It supports among others the following Internet protocols: HTTP(S), FTP(S), SCP, IMAP, SMTP. The tool requires two input paramters. First parameter is the service address and the second parameter is the directory of the provided demo requests. The request file has to have the extension *request*. For each provided request, the tool triggers the appropriate operation and persist the response attached with the current timestamp. Listing 5.3 depicts the script.

[2] `https://jcp.org/en/jsr/detail?id=318`

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Header>
      <ns2:provenanceInformation>
         <releaseVersion>2.1</releaseVersion>
         <sourceCodeRevision>23</sourceCodeRevision>
         <lastUpdate>2014-05-08T08:32:40.02Z</lastUpdate>
      </ns2:provenanceInformation>
   </S:Header>
   <S:Body>
      <ns2:convertTIFF2JPEGResponse xmlns:ns2="http://example/">
         <return>iVBORw0KGgoAAAANSUhEUgAAAPoAA...
         </ns2:convertTIFF2JPEGResponse>
    </S:Body>
</S:Envelope>
```

Figure 5.2: Provenance enhanced Web Service response

Listing 5.3: Call WebService test methods with cUrl.

```
#!/bin/sh

# use curl to call web service test methods
# INPUT: remote service address
# INPUT: directory including the provided requests
# OUTPUT: each request gets fired and a 'response' file including the current
    timestamp                is produced.

serviceURL="$1"
dir="$2"
timestamp=$(date +%Y-%m-%dT%H:%M:%S)
error_msg="the program 'curl' is not available under /usr/bin/curl! You can
   use 'sudo apt-get install curl' to install curl on your system."

if [ ! -d "$dir_out" ]; then
  mkdir "$dir_out"
fi

[ -f /usr/bin/curl ] &&
        echo "start requesting Web Service ("$serviceURL")." ||
                { echo "$error_msg" ; exit 1;}

for input in $(find "$dir_in" -name "*.request")
do
   echo "input file is "$input
   curl --header "Content-Type: text/xml;charset=UTF-8"  --data @"$input" "
       $serviceURL" > "$input"."$timestamp".response
   echo "output file is " "$dir_out"."$timestamp".response
   mv "$input"."$timestamp".response "$dir_out"/`basename "$input"`.response
done
```

Usage One approach to collect the service's dependencies is to execution the demo operations. As part of the framework, we have developed a script based tool for collecting Web Service dependencies. More precisely, we utilize the *Process Migration Framework* provided by [8]. As aforementioned, it is used to create a context model for a input process. By calling the *demo operation execution* tool within the PMF framework the service's context model can be received.

5.6 Resilient Web Service Framework at a glance

The practical part of the Resilient Web Service Framework consists of following components:

- **Resilience Annotations** (section 5.1) - Provides a set of resilience annotations which have to be applied to the service source code in order to improve the semantics of the service interface.
- **Verification Tool** (section 5.2) - In a first step the tool verifies that the annotations are applied correctly. In a second step it verifies that the attached version identification does not violate the resilient versioning strategy.
- **Java2RWSDL** (section 5.3) - Transform Java Web Service source code file into a resilient WSDL. It processes the applied resilience annotations and attaches them to the service's interface.
- **Provenance Enriched Header** (section 5.4) - Attaches the provenance information to a service response header by intercepting the request.
- **Pre-commit-Hook** - is responsible for triggering the *verification tool* on a commit (section 5.2).
- **Dependency Collection Tool** (section-5.5) - Collects the Web Service's dependencies by requesting the demo operations with the provided input requests. The service's view path is presented as a context model instance.

Table 5.2 depicts which component supports which policy. We have to note, that two policies (P13,P14) are not covered within this framework.

Table 5.2: Which tool supports which policy

	P1	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**	**P9**	**P10**	**P11**	**P12**
Resilience Annotations	x			x	x	x						
Verification Tool	x		x	x	x	x	x					
Java2RWSDL				x		x						
Provenance Response		x										
Capturing Tool									x		x	
RSS Notifications										x	x	x

Derived from the implemented feature set, we have explored them to two different levels. At a first level we distinguish either the feature is relevant for the service provider or the service consumer. At the second level we distinguish if a feature is applied during the development process (Deploy Time) or at Runtime. Figure 5.3 depicts the outcome of that assignment. Each feature gets classified as a supporting tool either for provider or consumer and its application phase is provided by the figure. Additional the figure sets the features in relation to the Web Service life cycle phases present in section 2.4.

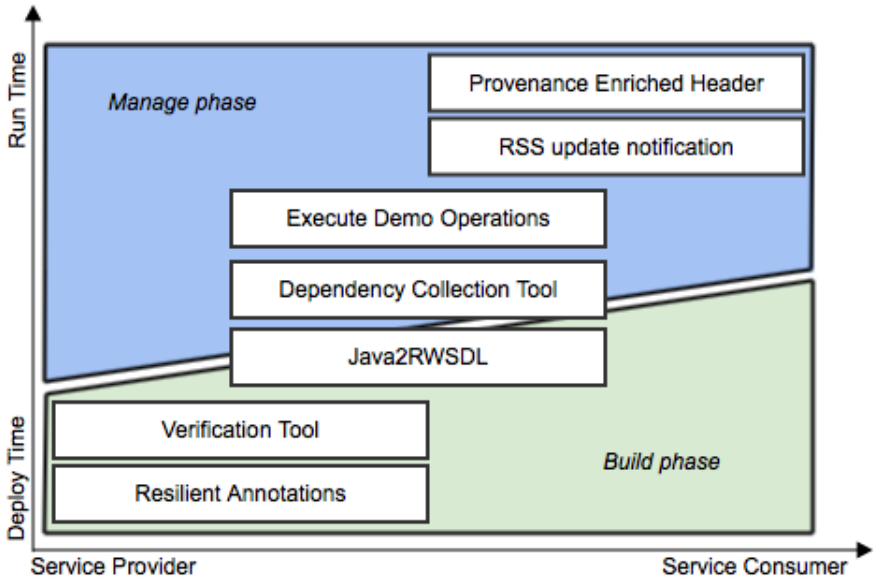

Figure 5.3: Feature appliance assigned to involving party and Web Service life cycle phases

The framework is available for download at `http://www.ifs.tuwien.ac.at/dp/process/projects/rws.html`.

CHAPTER 6

Proof of concept & Demonstration

To demonstrate the capabilities of the Resilient Web Service Framework we will go trough various scenarios addressing different aspects of applying it. Starting point is the *PictureService* introduced in section 4.1. It is a Java Web Service using Apache Maven as the dependency management and build tool.

6.1 Transforming an existing Web Service into a RWS

As a first step the service provider has to setup the Resilient Web Service Framework.

Setup of the Resilient Web Service Framework

Setting up the framework on a Linux environment includes the following steps:

- Create a writeable directory named `resilient` in the root folder.
- Assuming the service is named *PictureService*, the newly created directory has to contain the following files:
  - *PictureService.svn* - file storing the SVN URL and the credentials.
  - *PictureService.contact* - file storing the contact information in a UDDI compliant format.
  - *PictureService.server* - file storing the location and credentials of the application server.
  - *PictureService.rss* - RSS file storing all RSS feeds for that certain service channel.
  - Verification tool - Gets executed during the pre-commit-hook.
  - Java2RWSDL tool - Generates a (R)WSDL for a Java Web Service source code file.
  - Dependency Collection Tool - Used to collect the Web Service's dependencies.

- Copy the `pre-commit-hook` file provided by the framework in the *hooks* subdirectory of your SVN repository. Make sure that the hook is executable.
- In case the service provider makes use of a resilient PaaS provider, he has to subscribe its service to the related PaaS RSS channel. That step is necessary to get notifications on dependency updates.

To transform an existing Web Service into a RWS, the Web Service provider has to apply the following steps:

- The basic service source code has to be extended with the provided resilience annotations.
- For each service operation, a demo operation has to be provided (see POLICY 8).
- The service provider has to apply source code tests to fullfil the required code coverage (see POLICY 6,7).
- In addition to the resilience annotations, the service provider has to attach the following annotation `@Interceptors(ProvenanceInterceptor.class)` on class level of the service. That ensures that the provenance information is attached to the SOAP response header.

Applying all the aforementioned instructions will result in the following benefits:

- The resilient versioning policy is forced by the SVN automatically.
- Java2RWSDL generates a resilient WSDL out of the source code (an example already has been presented in section-5.3).
- Each SOAP response is provenance enriched.
- In case of the service URL is http://my.server.com/pictureService-v1/PictureService, the contact information of the provider can be found `http://my.server.com/pictureService-v1/PictureService/contact` and the RSS feed is reachable at `http://my.server.com/pictureService-v1/PictureService/updates`

6.2 Adding a new resilient operation results in new a minor release

Scenario description In this scenario we want to demonstrate the framework capabilities according to a source code update introduced by the service owner. The source code update is about adding a new operation to the service. Since adding a new operation does not break the compatibility, it is considered a minor update.

1. The service owner implements a new resilient operation the *retrieveAlbumByTags*.

```
$ svn commit -m "add retrieveAlbumByTags operation to PictureService"
Sending        src/main/java/com/resilient/service/pictureService/
    ResilientPictureService.java
Transmitting file data .svn: E165001: Commit failed (details follow):
svn: E165001: Commit blocked by pre-commit hook (exit code 2) with output:
Resilient Pre-Commit-Hook gets executed
try to fetch file /tmp/5-22/trunk/src/main/java/com/resilient/service/
    pictureService/ResilientPictureService.java
start annotation validation for service [ResilientPictureService.java]
  Class level annotations are valid.

------------------------------------------------------------------------
@Return annotation is missing for method 'retrieveAlbumByTags'.
@Validity annotation is invalid for method 'retrieveAlbumWithComments'. Not a
     proper RFC3339 timestamp.
------------------------------------------------------------------------
In total 2 validation violation(s) found!
Commit aborted!
```

Figure 6.1: Pre commit hook detects two violations and aborts the commit

```
$ svn commit -m "add missing annotations"
Sending        src/main/java/com/resilient/service/pictureService/
    ResilientPictureService.java
Transmitting file data .
Committed revision 6.
```

Figure 6.2: Source code annotations are valid. The commit action was successfully

2. The service owner commits the changes to the repository. The pre-commit-hook gets triggered and starts with the annotation verification. The verification tool reports two violations and aborts the commit (Figure 6.1).

3. The service owner adds the missing annotation and corrects the invalid timestamp. Subsequent he tries to commit again (Figure 6.2). This time the commit is successfully.

4. The service owner deploys the service to the application server according to the version policy. Since it is only a minor update and the compatibility does not break, the service can be redeployed at the original location.

5. After the successfully deployment of the service, the application server triggers the capturing of the service's View Path. For this, the PaaS provider executes the *Dependency Collection Tool* provided by the Resilient Framework. It is under the responsibility of the PaaS provider to manage the View Path(s) of its hosted resilient service(s). Since the Web Service View Path contains nearly 600 individuals, we can not visualize them in a figure properly. Therefore, we only present an excerpt of the View Path. Figure 6.3 presents a subset of the Web Service's View Path as the context model representa-

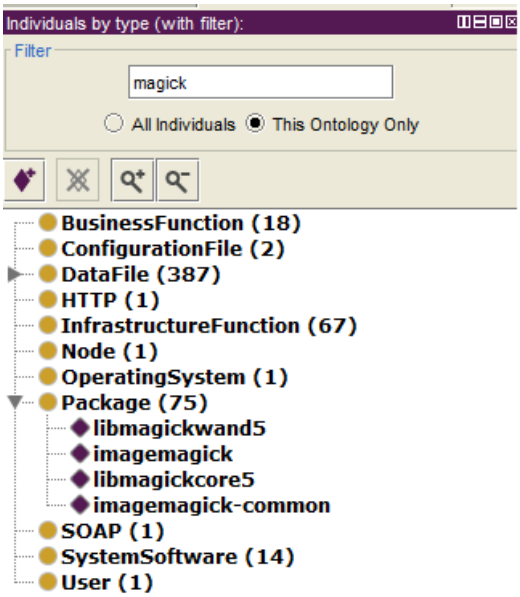

Figure 6.3: Excerpt of the Web Service's viewed by Protege

tion viewed with Protege 4.3.0. By applying for e.g. *magick* via the string filter, the four depending packages `libmagickwand5`,`imagemagick`,`imagemagickcore5` and `imagemagick-common` are presented.

6.3 PaaS updates ImageMagick via the package manager

Scenario description In that scenario we demonstrate the framework capabilities according to a dependency update applied by the PaaS provider. Since the *Collect Dependency Tool* is based on the PMF framework, only package based operating systems are applicable. Therefore, we assume the PaaS provider uses a Linux operating system.

- Due to the release of a new version of imageMagick the PaaS provider wants to update the current installed version (`ImageMagick 6.7.7-10 2012-11-06 Q16`).
- The PaaS provider first runs `apt-get -s install imagemagick` to detect which installed packages are affected by that update. Figure 6.4 depicts the result of the command. It shows that in total 7 packages will be upgraded and 11 packages will additionally installed. None of the current installed packages have to be removed.
- According to that information the PaaS provider can reason over the View Path to detect if the PictureService will be affected by the update. That can be done by executing a simple SPARQL query reasoning for the items listed at *The following packages will be upgraded* in figure 6.4. The result of the SPARQL query is presented in figure 6.5.

```
unma ~ # apt-get -s install imagemagick
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following package was automatically installed and is no longer required:
  authbind
Use 'apt-get autoremove' to remove it.
The following extra packages will be installed:
  dpkg gcc-4.9-base imagemagick-6.q16 imagemagick-common libfftw3-3 libfftw3-
      double3 libfftw3-long3 libfftw3-single3 libgomp1 libmagickcore-6.q16-2
  libmagickcore-6.q16-2-extra libmagickwand-6.q16-2 libnetpbm10 libselinux1
      libselinux1:i386 libtiff5 netpbm
Suggested packages:
  imagemagick-doc autotrace cups-bsd lpr lprng enscript ffmpeg gnuplot grads
      graphviz hp2xx html2ps libwmf-bin povray radiance texlive-base-bin
      transfig libfftw3-bin
  libfftw3-dev inkscape
The following NEW packages will be installed:
  gcc-4.9-base imagemagick-6.q16 libfftw3-double3 libfftw3-long3 libfftw3-
      single3 libmagickcore-6.q16-2 libmagickcore-6.q16-2-extra libmagickwand
      -6.q16-2 libnetpbm10
  libtiff5 netpbm
The following packages will be upgraded:
  dpkg imagemagick imagemagick-common libfftw3-3 libgomp1 libselinux1
      libselinux1:i386
7 upgraded, 11 newly installed, 0 to remove and 1436 not upgraded.
```

Figure 6.4: Execute update apt-get install imagemagick to updates its version.

Following packages of the PictureService View Path are affected by that certain update: *imagemagick,libgomp1,libselinux1,imagemagick-common*

- Subsequently the provider executes the update. That changes the installed ImageMagick version to `ImageMagick 6.8.9-6 Q16 2014-09-06`.
- After the installation of the new version of ImageMagick, the service provider re-executes the *Dependency Collection Tool* and gets the updated View Path. Figure 6.6 shows the same View Path excerpt as it is presented in figure 6.3, but now the View Path contains the updated imageMagick dependencies.
- At this point the PaaS provider adds a new feed concerning the update to the Picture-Service RSS channel. According to the POLICY 11 the feed has to provide a *Software Replacement* event for all affected packages. Figure 6.7 shows the RSS notification—pushed by the PaaS provider—with a standard RSS feed reader (e.g. Mozilla Firefox Browser). By clicking on the link in the feed, the detailed PREMIS update informations are provided.
- Receiving a notification forces the RWS provider to verify the service behaviour. If the service behaviour has changed, the service provider has to push a notification—forward

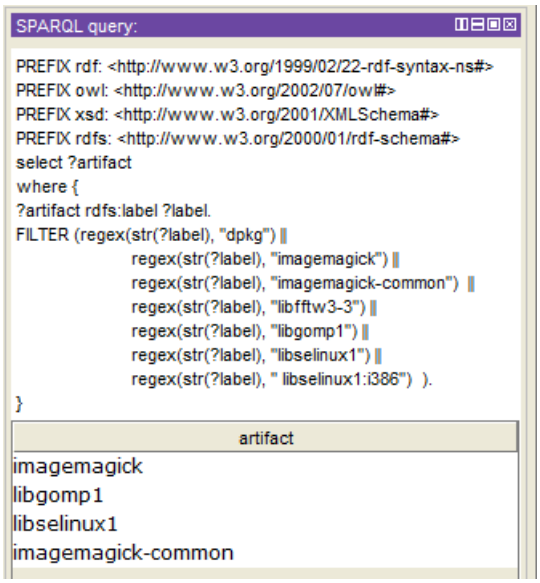

Figure 6.5: Result of the SPARQL query to detect if and which elements are affected.

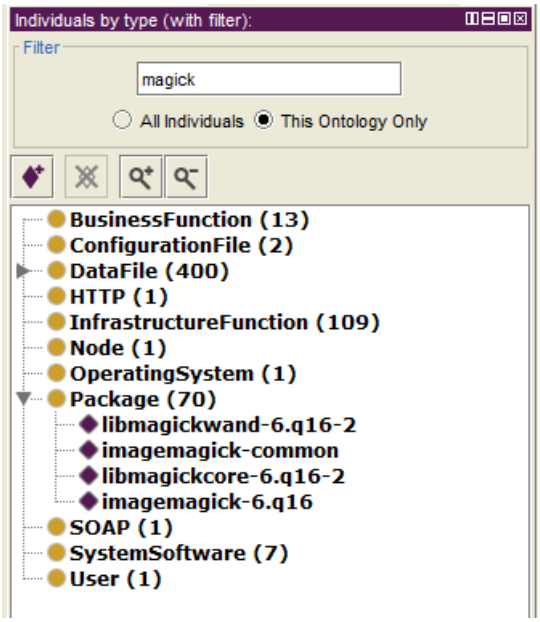

Figure 6.6: Excerpt of the updated Web Service's viewed by Protege

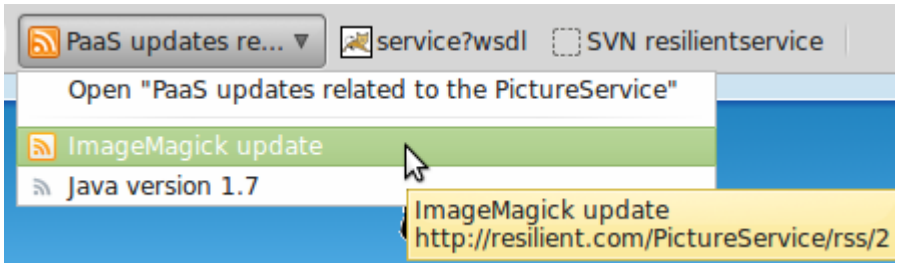

Figure 6.7: A new push notification from the PaaS provider is available

this event—to his consumers. If the PaaS update affects the service view path, but has no impact on the service behaviour, the service provider does not has to inform its consumers.

6.4 Limitations of the Resilient Web Service Framework

Although, the framework covers a lot of different aspects, we also want to present its limitations.

- **Behavioural Backward Compatibility** - At the beginning of this thesis, addressing *behavioural backward compatibility* was one of the cornerstones. But the investigations reveal that this is only doable to a certain extent. For stateless deterministic operations, a response comparison solves that issue. For stateful determinstic it is a bit more complicated, but also feasible. By tracing and re-executing the state changes a response verification should be possible. But for non-deterministic operations a behavioural backward compatibility verification is—by definition—impossible. Hence, the stress of this thesis had shifted and we focused on other important resilience aspects.

- **WSDL driven approach** - RESTful Web Services, as a lightweight alternative to SOAP services, have become popular in recent years. Most of the developed policies and also the tools are applicable also on RESTful services (except the Java2RWSDL, which would not make sense because RESTful services do not have a WSDL interface). In case of missing the Java2RWSDL tool-support, an alternative has to be found to attach the resilience annotations to the RESTful interface. The XML based *Web Application Description Language* (WADL) is used to model the resources provided by a HTTP-based web application in a machine readable manner. Since a RESTful service is a typical implementation of a HTTP web application, the Resilient Web Service Framework can be extended to also support a resilient RESTful to a resilient WADL transformation.

- **Comments on the capturing of the View Path** - Since the process of capturing Web Service dependencies is using `strace` to trace system calls a lot of data is generated. The subsequent processing to gather the context model is very time intensive. For the *PictureService* it takes nearly twenty minutes on a moderate computer. Another important aspects is that the completeness of the dependency collection can not be verified. On the one hand the quality of the View Path always depends on the quality of the provided demo

operations by the service provider. On the other hand, expert knowledge is required for a critical review of the service's view path.

- **Hardware dependency gathering** - The current version of the PMF tool does not extract hardware dependencies. Utilizing the Linux Hardware Extractor[1] developed by the TIMBUS project, the PMF tool can be extended to also extract hardware related informations. Since Web Services are often deployed in virtual environment (like in a PaaS scenario) hardware changes will not affect the service directly.

[1] `https://opensourceprojects.eu/p/timbus/context-population/extractors/linux-hw/`

CHAPTER 7

Summary & Outlook

In this thesis we contributed the concept of Resilient Web Services (RWS) aiming to ensure process continuity. Firstly, we motivated this thesis by focusing on the causes, which force the problems according to process continuity. Among others, volatile external third party artifacts are the main reason for process decay [5]. Web Services are a common way to realize such remote artifacts. However, Web Services are subject to constant functional and behavioural changes. Such changes can be triggered either by arising business needs and regulations or modifications in the Web Service dependency stack. To satisfy those needs, Web Service providers are constantly forced to release updates of an existing service.

As a first step we investigated the reasons for the dynamic nature of Web Services. We presented the main challenges leading to outdated processes through the volatility of Web Services. We investigated the various kind of changes according to Web Service evolution. In the related work we presented contributions addressing the changes of the Web Service interface by a proper versioning strategy. Since a proper version strategy does not guarantee resilience, we investigated further challenges related to RWS. Such challenges are e.g. a resilient dependency management, a semantic enhanced Web Service interface or a notification support for consumers in case of an update event has happened. According to those challenges we introduced a framework including a set of requirements to support RWS. Subsequently we derived a policy catalog out of the requirements. The framework encompasses policies in the following areas:

- Enforcing a resilient versionig strategy
- Adding semantics to the Web Service interface
- Providing provenance enriched Web Service responses
- Testing of the Web Service behaviour
- Identifying and monitoring of Web Service dependencies
- Notifying Web Service consumers on updates

To prove the applicability of the framework, we developed the following prototype tools:

- Verification tool - Used to verify the interface backward compatibility based on the previous source code version of the Web Service.
- Java2RWSDL - Used to transform a Java class to a semantic enhanced version of WSDL.
- Dependency Collection Tool - Used to identify the Web Service View Path.

To demonstrate the capabilities of the Resilient Web Service Framework we presented scenarios addressing different aspects of service evolution. The first scenario dealed with a typical Web Service source code update introduced by the service provider. The second scenario demonstrated the benefits of the RWS framework according to an update of Web Service dependency stack. In the last section, we discussed the limitations of the framework.

7.1 Future work

Although we tried to address the most important aspects according to RWS, certain not less important aspects are missing. Some of them are already mentioned in the chapters above, but for a more comprehensive view we will present them in this section again. Next to the known limitations (see section 6.4) of the current version of the framework, a future contribution has to address the following missing issues to improve the RWS framework:

- *Better integration of expert knowledge according to the dependency capturing process.* Since the Web Service dependency identification process utilizes the provided testing methods, the completeness of the Web Service view path can not be guaranteed. Therefore, additional expert knowledge would be necessary to verify and adapt the captured view path. Since the Web Service's view path is presented as a process context model, tools like Protege supports the visualization of the dependencies. However, the users of the framework have to install an additional application.
- *Artifacts substitutes.* At the moment the framework functionality is lacking of an alternative recommendation system for outdated dependencies. In case of the Web Service depends on an outdated dependency, the accountable resilient provider has to replace that dependency by a suitable substitute. To find such a substitute, a proper recommendation system is crucial to ensure Web Service resilience.
- *Remote Web Service dependencies.* The framework currently does not support the identification of Web Service's remote dependencies. We do not inspect the source to detect such external dependencies. Currently the developed policies only cover the aspect of the operation lifetime guaranteed by the remote service. One approach to support that issues, can be e.g. a tool polling the lifetime of the remote service and triggering a notification in case of an unexpected unavailability of that dependency. A violation of the guaranteed lifetime can be covered by a SLA.

APPENDIX A

Appendix A

A.1 WSDL 2.0 sample

Listing A.1 presents the WSDL 2.0 interface of the *The GreatH Web Service*. It is the description of a hypothetical hotel reservation service used as a scenario in the official WSDL2.0 specification to promote the features of WSDL2.0.

Listing A.1: WSDL sample

```
<?xml version="1.0" encoding="utf-8" ?>
<description xmlns="http://www.w3.org/ns/wsdl"
targetNamespace="http://greath.example.com/2004/wsdl/resSvc"
xmlns:tns= "http://greath.example.com/2004/wsdl/resSvc"
xmlns:ghns="http://greath.example.com/2004/schemas/resSvc"
xmlns:wsoap="http://www.w3.org/ns/wsdl/soap"
xmlns:soap="http://www.w3.org/2003/05/soap-envelope" xmlns:wsdlx=
"http://www.w3.org /ns/wsdl-extensions">
<documentation>The GreatH Web service. </documentation>
<types>
<xs:schema xmlns:xs="http://www.w3.org/2001/ XMLSchema"
targetNamespace = "http://greath.example.com/2004/ schemas/resSvc"
xmlns="http://greath.example.com/2004 /schemas/resSvc">
<xs:element name="checkAvailability" type= "tCheckAvailability"/>
<xs:complexType name="tCheckAvailability">
<xs:sequence><xs:element name="checkInDate" type="xs:date"/>
<xs:element name="checkOutDate" type="xs:date"/>
<xs:element name="roomType" type="xs:string"/>
</xs:sequence> </xs:complexType>
<xs:element name="checkAvailabilityResponse" type="xs:double"/>
<xs:element name="invalidDataError" type="xs:string"/>
</xs:schema></types>
<interface name = "reservationInterface" >
<fault name = "invalidDataFault" element = "ghns:invalidDataError"/>
<operation name = "opCheckAvailability"
pattern="http://www.w3.org/ns/wsdl/in-out"
```

```
style="http://www.w3.org/ns/wsdl/style/iri" wsdlx:safe = "true">
<input messageLabel="In" element="ghns:checkAvailability" />
<output messageLabel="Out" element= "ghns:checkAvailabilityResponse"
/>
<outfault ref="tns:invalidDataFault" messageLabel="Out"/>
</operation></interface>
<binding name="reservationSOAPBinding" interface =
"tns:reservationInterface" type="http://www.w3.org/ns/wsdl/soap"
wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/">
<fault ref="tns:invalidDataFault" wsoap:code="soap:Sender"/>
<operation ref="tns:opCheckAvailability" wsoap:mep
="http://www.w3.org/2003/05/soap/mep/soap response"/>
</binding>
<service name="reservationService" interface =
"tns:reservationInterface">
<endpoint name="reservationEndpoint" binding =
"tns:reservationSOAPBinding" address =
"http://greath.example.com/2004/reservation"/>
</service></description>
```

A.2 WSDL-Temporal sample

Listing A.2 presents a possible WSDL-T interface of the *The GreatH Web Service*. It is important to notice, that each WSDL element includes the two additional elements *validity* and *timestamp*.

Listing A.2: WSDL-T sample

```
<?xml version="1.0" encoding="utf-8" ?>
<description xmlns="http://www.w3.org/ns/wsdl"
targetNamespace="http://greath.example.com/2004/wsdl/resSvc"
xmlns:tns= "http://greath.example.com/2004/wsdl/resSvc"
xmlns:ghns="http://greath.example.com/2004/schemas/resSvc"
xmlns:wsoap="http://www.w3.org/ns/wsdl/soap"
xmlns:soap="http://www.w3.org/2003/05/soap-envelope" xmlns:wsdlx=
"http://www.w3.org /ns/wsdl-extensions" validity="latest" timestamp
    ="07/11/2011 16:53:34" >
<documentation> The GreatH WSDL-Temporal Web service.
</documentation><types>
<xs:schema xmlns:xs="..." targetNamespace="..." xmlns="...">
<xs:element name="checkAvailability#1.0.0" type = "
tCheckAvailability#1.0.0" validity="latest" timestamp="07/11/2011
16:53:34" />
<xs:complexType name="tCheckAvailability#1.0.0" validity="latest"
timestamp="07/11/2011 16:53:34" >
<xs:sequence><xs:element name="checkInDate#1.0.0" type="xs:date"
validity="latest" timestamp="07/11/2011 16:53:34" />
validity="latest" timestamp="07/11/2011 16:53:34" />
<xs:element name="roomType#1.0.0" type="xs:string"
validity="latest" timestamp="07/11/2011 16:53:34" />
<xs:element name="numberOfRooms#1.1.0" type="xs:int"
validity="latest" timestamp="17/11/2011 11:23:54" />
<xs:element name="branchHotelName#1.1.0" type="xs:string"
```

```
validity="latest" timestamp=="17/11/201111:23:54"/>
</xs:sequence> </xs:complexType>
<xs:element name="checkAvailabilityResponse#1.1.0"
type="xs:tCheckAvailabilityResponse" validity="latest" timestamp =
"17/11/2011 11:23:54" />
<xs:complexType name="tCheckAvailabilityResponse#1.1.0"
validity="latest" timestamp="07/11/2011 16 :53:34" >
<xs:sequence><xs:element name="numberOfRooms#1.1.0"
type="xs:int" validity="latest" timestamp="17/11/2011 11:23:54" />
<xs:element name="totalFare#1.1.0" type="xs:string" validity="latest"
timestamp="17/11/2011 11:23:54" />
</xs:sequence> </xs:complexType>
<xs:element name="checkAvailabilityResponse#1.0.0" type= "xs:double"
validity="past" timestamp="07/11/2011 16:53:34" />
<xs:element name="invalidDataError#1.0.0" type="xs:string"
validity="latest" timestamp="07/11/2011 16:53:34" />
</xs:schema> </types>
<interface name="reservationInterface#1.0.0" validity="latest"
timestamp="07/11/2011 16:53:34" >
<fault name = "invalidDataFault#1.0.0" element = "ghns:
invalidDataError#1.0.0" validity="latest" timestamp="07/11/2011
16:53:34" />
<operation name="opCheckAvailability#1.0.0" pattern= "http://www.w3.
org/ns/wsdl/in-out" style= "http://www.w3.org/ns/wsdl/style/iri" wsdlx:safe
= "true" validity="latest" timestamp="07/11/2011 16:53:34" >
<input messageLabel="In" element="ghns:checkAvailability#1.0.0" />
<output messageLabel="Out" element= "ghns:
checkAvailabilityResponse#1.0.0" />
<outfault ref="tns: invalidDataFault#1.0.0" messageLabel="Out"/>
</operation></interface>
<binding name="reservationSOAPBinding#1.0.0" interface = "tns:
reservationInterface#1.0.0" type="http://www.w3.org/ns/wsdl/soap"
wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/"
validity="latest" timestamp="07/11/2011 16:53:34" >
<fault ref="tns:invalidDataFault#1.0.0" wsoap:code="soap:Sender"/>
<operation ref="tns:opCheckAvailability#1.0.0" soap:mep =
"http://www.w3.org/2003/05/soap/mep/soap-response"/>
</binding>
<service name="reservationService#1.0.0" interface = "tns:
reservationInterface#1.0.0" validity="latest" timestamp="07/11/2011
16:53:34" >
<endpoint name="reservationEndpoint#1.0.0" binding="tns:
name="reservationSOAPBinding#1.0.0" address =
"http://greath.example.com/2004/reservation" validity="latest"
timestamp="07/11/2011 16:53:34" />
</service></description>
```

A.3 WSDL of PictureService

Listing A.3 presents the standard WSDL of the PictureService. It gets automatically generated from the Java code by deploying the service to an JavaEE application server (e.g. JBoss).

Listing A.3: WSDL of the PictureService

```
<?xml version='1.0' encoding='UTF-8'?><!-- Published by JAX-WS RI at http://
    jax-ws.dev.java.net. RI's version is Metro/2.3 (tags/2.3-7528; 2013-04-29
    T19:34:10+0000) JAXWS-RI/2.2.8 JAXWS/2.2 svn-revision#unknown. --><!--
    Generated by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is
    Metro/2.3 (tags/2.3-7528; 2013-04-29T19:34:10+0000) JAXWS-RI/2.2.8 JAXWS
    /2.2 svn-revision#unknown. --><definitions xmlns:wsu="http://docs.oasis-
    open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" xmlns:
    wsp="http://www.w3.org/ns/ws-policy" xmlns:wsp1_2="http://schemas.xmlsoap
    .org/ws/2004/09/policy" xmlns:wsam="http://www.w3.org/2007/05/addressing/
    metadata" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="
    http://pictureService.service.resilient.com/" xmlns:xsd="http://www.w3.
    org/2001/XMLSchema" xmlns="http://schemas.xmlsoap.org/wsdl/"
    targetNamespace="http://pictureService.service.resilient.com/" name="
    PictureServiceService">
<types>
<xsd:schema>
<xsd:import namespace="http://pictureService.service.resilient.com/"
    schemaLocation="http://wsatvie006:8080/pictureService-1.0-SNAPSHOT/
    PictureServiceService?xsd=1"/>
</xsd:schema>
</types>
<message name="convertJpeg2Png">
<part name="parameters" element="tns:convertJpeg2Png"/>
</message>
<message name="convertJpeg2PngResponse">
<part name="parameters" element="tns:convertJpeg2PngResponse"/>
</message>
<message name="PictureServiceException">
<part name="fault" element="tns:PictureServiceException"/>
</message>
<message name="searchImages">
<part name="parameters" element="tns:searchImages"/>
</message>
<message name="searchImagesResponse">
<part name="parameters" element="tns:searchImagesResponse"/>
</message>
<message name="retrieveAlbum">
<part name="parameters" element="tns:retrieveAlbum"/>
</message>
<message name="retrieveAlbumResponse">
<part name="parameters" element="tns:retrieveAlbumResponse"/>
</message>
<message name="retrieveAlbumWithComments">
<part name="parameters" element="tns:retrieveAlbumWithComments"/>
</message>
<message name="retrieveAlbumWithCommentsResponse">
<part name="parameters" element="tns:retrieveAlbumWithCommentsResponse"/>
</message>
<portType name="PictureService">
<operation name="convertJpeg2Png">
<input wsam:Action="http://pictureService.service.resilient.com/
    PictureService/convertJpeg2PngRequest" message="tns:convertJpeg2Png"/>
```

```
<output wsam:Action="http://pictureService.service.resilient.com/
    PictureService/convertJpeg2PngResponse" message="tns:
    convertJpeg2PngResponse"/>
<fault message="tns:PictureServiceException" name="PictureServiceException"
    wsam:Action="http://pictureService.service.resilient.com/PictureService/
    convertJpeg2Png/Fault/PictureServiceException"/>
</operation>
<operation name="searchImages">
<input wsam:Action="http://pictureService.service.resilient.com/
    PictureService/searchImagesRequest" message="tns:searchImages"/>
<output wsam:Action="http://pictureService.service.resilient.com/
    PictureService/searchImagesResponse" message="tns:searchImagesResponse"/>
<fault message="tns:PictureServiceException" name="PictureServiceException"
    wsam:Action="http://pictureService.service.resilient.com/PictureService/
    searchImages/Fault/PictureServiceException"/>
</operation>
<operation name="retrieveAlbum">
<input wsam:Action="http://pictureService.service.resilient.com/
    PictureService/retrieveAlbumRequest" message="tns:retrieveAlbum"/>
<output wsam:Action="http://pictureService.service.resilient.com/
    PictureService/retrieveAlbumResponse" message="tns:retrieveAlbumResponse
    "/>
<fault message="tns:PictureServiceException" name="PictureServiceException"
    wsam:Action="http://pictureService.service.resilient.com/PictureService/
    retrieveAlbum/Fault/PictureServiceException"/>
</operation>
<operation name="retrieveAlbumWithComments">
<input wsam:Action="http://pictureService.service.resilient.com/
    PictureService/retrieveAlbumWithCommentsRequest" message="tns:
    retrieveAlbumWithComments"/>
<output wsam:Action="http://pictureService.service.resilient.com/
    PictureService/retrieveAlbumWithCommentsResponse" message="tns:
    retrieveAlbumWithCommentsResponse"/>
<fault message="tns:PictureServiceException" name="PictureServiceException"
    wsam:Action="http://pictureService.service.resilient.com/PictureService/
    retrieveAlbumWithComments/Fault/PictureServiceException"/>
</operation>
</portType>
<binding name="PictureServicePortBinding" type="tns:PictureService">
<soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="
    document"/>
<operation name="convertJpeg2Png">
<soap:operation soapAction=""/>
<input>
<soap:body use="literal"/>
</input>
<output>
<soap:body use="literal"/>
</output>
<fault name="PictureServiceException">
<soap:fault name="PictureServiceException" use="literal"/>
</fault>
</operation>
```

```
<operation name="searchImages">
<soap:operation soapAction=""/>
<input>
<soap:body use="literal"/>
</input>
<output>
<soap:body use="literal"/>
</output>
<fault name="PictureServiceException">
<soap:fault name="PictureServiceException" use="literal"/>
</fault>
</operation>
<operation name="retrieveAlbum">
<soap:operation soapAction=""/>
<input>
<soap:body use="literal"/>
</input>
<output>
<soap:body use="literal"/>
</output>
<fault name="PictureServiceException">
<soap:fault name="PictureServiceException" use="literal"/>
</fault>
</operation>
<operation name="retrieveAlbumWithComments">
<soap:operation soapAction=""/>
<input>
<soap:body use="literal"/>
</input>
<output>
<soap:body use="literal"/>
</output>
<fault name="PictureServiceException">
<soap:fault name="PictureServiceException" use="literal"/>
</fault>
</operation>
</binding>
<service name="PictureServiceService">
<port name="PictureServicePort" binding="tns:PictureServicePortBinding">
<soap:address location="http://wsatvie006:8080/pictureService-1.0-SNAPSHOT/
    PictureServiceService"/>
</port>
</service>
</definitions>
```

Listing A.3 presents the XSD Schema for the in- and output types of the PictureService. It gets automatically generated from the Java code by deploying the service to the Glassfish applications server.

Listing A.4: XSD Schema for the in- and output types of the PictureService

```
<?xml version='1.0' encoding='UTF-8'?><!-- Published by JAX-WS RI at http://
    jax-ws.dev.java.net. RI's version is Metro/2.3 (tags/2.3-7528; 2013-04-29
    T19:34:10+0000) JAXWS-RI/2.2.8 JAXWS/2.2 svn-revision#unknown. --><xs:
```

```
    schema xmlns:tns="http://pictureService.service.resilient.com/" xmlns:xs
    ="http://www.w3.org/2001/XMLSchema" version="1.0" targetNamespace="http
    ://pictureService.service.resilient.com/">

<xs:element name="PictureServiceException" type="tns:PictureServiceException
    "/>

<xs:element name="convertJpeg2Png" type="tns:convertJpeg2Png"/>

<xs:element name="convertJpeg2PngResponse" type="tns:convertJpeg2PngResponse
    "/>

<xs:element name="retrieveAlbum" type="tns:retrieveAlbum"/>

<xs:element name="retrieveAlbumResponse" type="tns:retrieveAlbumResponse"/>

<xs:element name="retrieveAlbumWithComments" type="tns:
    retrieveAlbumWithComments"/>

<xs:element name="retrieveAlbumWithCommentsResponse" type="tns:
    retrieveAlbumWithCommentsResponse"/>

<xs:element name="searchImages" type="tns:searchImages"/>

<xs:element name="searchImagesResponse" type="tns:searchImagesResponse"/>

<xs:complexType name="retrieveAlbumWithComments">
<xs:sequence>
<xs:element name="arg0" type="xs:string" minOccurs="0"/>
</xs:sequence>
</xs:complexType>

<xs:complexType name="retrieveAlbumWithCommentsResponse">
<xs:sequence>
<xs:element name="return" type="xs:string" minOccurs="0" maxOccurs="unbounded
    "/>
</xs:sequence>
</xs:complexType>

<xs:complexType name="PictureServiceException">
<xs:sequence>
<xs:element name="message" type="xs:string" minOccurs="0"/>
</xs:sequence>
</xs:complexType>

<xs:complexType name="retrieveAlbum">
<xs:sequence>
<xs:element name="arg0" type="xs:string" minOccurs="0"/>
</xs:sequence>
</xs:complexType>

<xs:complexType name="retrieveAlbumResponse">
<xs:sequence>
```

```
<xs:element name="return" type="xs:string" minOccurs="0" maxOccurs="unbounded
    "/>
</xs:sequence>
</xs:complexType>

<xs:complexType name="convertJpeg2Png">
<xs:sequence>
<xs:element name="arg0" type="xs:base64Binary" minOccurs="0"/>
</xs:sequence>
</xs:complexType>

<xs:complexType name="convertJpeg2PngResponse">
<xs:sequence>
<xs:element name="return" type="xs:base64Binary" minOccurs="0"/>
</xs:sequence>
</xs:complexType>

<xs:complexType name="searchImages">
<xs:sequence>
<xs:element name="arg0" type="xs:string" nillable="true" minOccurs="0"
    maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>

<xs:complexType name="searchImagesResponse">
<xs:sequence>
<xs:element name="return" type="xs:base64Binary" minOccurs="0" maxOccurs="
    unbounded"/>
</xs:sequence>
</xs:complexType>
</xs:schema>
```

A.4 Recommendations for Arifact Versioning in SOA

Table A.4 presents a list of recommendations for artifact version in SOA as presented by Novakouski *et al.* [36]. Each recommendation is assigned to a certain topic. This contribution is mainly referring to the topics of *Service Interface Design*, *Life-Cycle Policy* and *Tool Strategies*.

Number	Topic	Recommendation
1	Key Artifacts	Place all WSDL documents under version control.
2	Key Artifacts	Define data types used in service interfaces in separate XML schemas, and place them under version control.
3	Key Artifacts	If the development effort for service-oriented systems includes composite services, consider the documents that control the composition as key artifacts and place them under version control.

4	Key Artifacts	Ensure that the versioning policy contains guidance for how to handle changes in SOA infrastructure components.
5	Key Artifacts	Identify all metadata that is relevant to service consumers, decide how to document it, and place the resulting artifacts under version control.
6	Service Interface Design	If the service provider will support multiple interfaces for a single service, include policies about how long to support each exposed interface. In addition, use a naming convention that indicates that these are all variants of the same interface.
7	Service Interface Design	Develop policies for how long to support multiple versions of the same interface that account for the faster change rates of unique interfaces. Even though over- loaded interfaces have lower change rates, place both the interface itself and the schemas that represent operation input types under version control.
8	Service Interface Design	Consistently with standard software development, version all key artifacts for internal use. From the perspective of the external service consumer, version either the entire service or the individual operation interfaces, depending on the needs of the potential consumers. Avoid exposing version information at a level lower than operation, as that is likely to confuse service consumers.
9	Service Interface Design	If a system provides services at different QoS levels, each with a different service interface, place all exposed interfaces under version control. In addition, use a naming convention that indicates that these are all variants of the same interface.
10	Policy Elements	Construct a comprehensive naming scheme, including version creation thresh- olds, change types, compatibility rules, and a scheme for determining the identi- fication name or number of a new artifact.
11	Policy Elements	Describe backward- and forward-compatibility requirements and goals in the versioning policy, as appropriate to the context. Actively seek backward com- patibility in service-oriented systems development, but realize that forward compatibility is much more difficult to ensure and may not be feasible.

12	Policy Elements	Use major and minor version classification to communicate compatibility issues to service consumers. However, do not make it a critical part of a versioning policy in SOA environments because it is not appropriate for all contexts.
13	Policy Elements	Use basic numeric naming schemes, including major/minor designations, for web services. Consider more complex schemes as the consumer base or capabilities increase in size.
14	Technology Strategies	Use WSDL documents as the backbone of any service-versioning strategy, and use the namespace field to differentiate services and interfaces. In more com- plex environments, extend or annotate the WSDL format to manage extra in- formation.
15	Technology Strategies	Make service design decisions, particularly regarding the use of namespaces, before defining XML-schema versioning policies.
16	Technology Strategies	For composite services, version the composition-control documents and make them version aware. With BPEL, use the extensions described by Juric and colleagues. Otherwise, consult recommended practices for the selected business-process-engine technology to enable version awareness.
17	Key Artifacts/ Technology Strategies	Place SLA documents under version control. Either use an ESB infrastructure to provide a standard way of managing SLA concerns, or investigate custom solutions for SLA management.
18	Technology Strategies	Plan the service infrastructure well in advance to avoid significant infrastructure change. If possible, use common open-technology standards to minimize the potential impact of infrastructure change. Architect service-oriented systems in a way that allows the infrastructure to evolve with minimal disruption to services and consumers.
19	Technology Strategies	In all but the most basic service-oriented systems (i.e., those with a small number of services that have well-known consumers), use some form of broker or router to simplify the interface exposed to consumers and enable greater control by the provider.

20	Tool Strategies	Select a VCS that is sufficiently robust to accommodate all the needs of the software development project. To improve productivity, also select an IDE that integrates well with the chosen VCS.
21	Tool Strategies	In all but the most basic service-oriented systems (i.e., those with a small num- ber of services that have well-known consumers), use registries in conjunction with service repositories to store additional service metadata and related artifacts. For larger implementations, use advanced service registry features to inform service consumers of changes and deal with multiple service versions.
22	Technology Strategies	If possible, use open web-service standards not only to support versioning if necessary but also to ensure compatibility with other systems.
23	Technology Strategies	If standard versioning approaches appear to be insufficient for a given service- oriented system, consult current research for ideas about new standards and methods for extending existing standards.
24	Life-Cycle Policy	Align the version-control policy with the organization testing strategy.
25	Life-Cycle Policy	Use compatibility testing for new versions of a service for both backward and forward compatibility to ensure proper support for consumers.
26	Life-Cycle Policy	Explicitly determine how many versions of a service to support and for how long.
27	Life-Cycle Policy	Release early versions of a service to support testing by service consumers, but manage and name them consistently to differentiate clearly between test versions and production versions of a service.
28	Life-Cycle Policy	When participating in the construction of a multi-organizational service-oriented system, write a codified communication policy about service changes to ensure the smooth evolution of the system.
29	Life-Cycle Policy	Actively provide notification of changes for transparent service interfaces; use passive policies for opaque service interfaces.
30	Life-Cycle Policy	Select update rates based on customer needs and SLAs, but ensure that the procedure can accommodate on-demand changes in critical situations.
31	Life-Cycle Policy	Follow a predictable update and deprecation schedule to make change coordination significantly easier.

32	Life-Cycle Policy	At the end of a service life cycle, manage its retirement process and eliminate all references to the service to prevent rogue services.

Table A.1: Recommended Practices for Artifact Versioning in Service-Oriented Systems (from [36])

Bibliography

[1] IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, pages 1–84, Dec 1990.

[2] Hema Banati, Punam Bedi, and Preeti Marwaha. Extending bpel for wsdl-temporal based web services. In *Proceedings of the 12th International Conference on Hybrid Intelligent Systems (HIS 2012)*, pages 484—489, Pune, India, December 2012.

[3] Hema Banati, Punam Bedi, and Preeti Marwaha. WSDL-Temporal: An approach for change management in Web Services. In *Proceedings of the 2nd International Conference on Uncertainty Reasoning and Knowledge Engineering (URKE 2012)*, pages 44–49, Jakarta, Indonesia, August 2012.

[4] Gabriel Bechara. http://www.oracle.com/technetwork/articles/web-services-versioning-094384.html.

[5] Khalid Belhajjame, Marco Roos, Esteban Garcia-Cuesta, Graham Klyne, Jun Zhao, David De Roure, Carole Goble, Jose Manuel Gomez-Perez, Kristina Hettne, and Aleix Garrido. Why workflows break - understanding and combating decay in taverna workflows. In *Proceedings of the 8th IEEE International Conference on E-Science (E-SCIENCE 2012)*, pages 1–9, 2012.

[6] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, 2001.

[7] N. Bieberstein, R. Laird, K. Jones, and T. Mitra. *Executing SOA: A Practical Guide for the Service-Oriented Architect*. Pearson Education, 2008.

[8] Johannes Binder, Stephan Strodl, and Andreas Rauber. Process migration framework – virtualising and documenting business processes. In *Workshop Proceedings of the 18th IEEE International EDOC Conference (EDOC'14)*, pages 95–103, Ulm, Germany, September 2014.

[9] David Booth, Hugo Haas, and Francis McCabe. Web service architecture. Technical report, http://www.w3.org/TR/ws-arch/#whatis, 2004.

[10] Vadym Borovskiy, Juergen Müller, Matthieu-Patrick Schapranow, and Alexander Zeier. Ensuring service backwards compatibility with generic web services. In *Proceedings of the ICSE Workshop on Principles of Engineering Service Oriented Systems (PESOS 2009)*, pages 95–98, Vancouver, Canada, May 2009.

[11] Kyle Brown and Michael Ellis. Best practices for web services versioning. http://www.ibm.com/developerworks/webservices/library/ws-version/, Jan 2004.

[12] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, and W.E. Wong. Code coverage of adaptive random testing. *IEEE Transactions on Reliability*, 62(1):226–237, March 2013.

[13] Tsong Yueh Chen, T. H. Tse, and Y. T. Yu. Proportional Sampling Strategy: A Compendium and Some Insights. *Journal of Systems and Software*, 58(1):65–81, August 2001.

[14] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web services description language (wsdl) version 2.0 part 1: Core language. Technical report, http://www.w3.org/TR/wsdl20/, June 2007.

[15] Steve Cornett. http://www.bullseye.com/minimum.html, 2006.

[16] Dimitris Dranidis, Ervin Ramollari, and Dimitrios Kourtesis. Run-time Verification of Behavioural Conformance for Conversational Web Services. In *Proceedings of the 7th IEEE European Conference on Web Services (ECOWS 2009)*, pages 139–147, Eindhoven, The Netherlands, November 2009.

[17] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[18] Thomas Erl, Stephen G. Bennett, Benjamin Carlyle, Clive Gee, Robert Laird, Anne Thomas Manes, Robert Moores, Robert Schneider, Leo Shuster, Andre Tost, Chris Venable, and Filippos Santas. *SOA Governance: Governing Shared Services On-Premise and in the Cloud (The Prentice Hall Service Technology Series from Thomas Erl)*. Prentice Hall, 2011.

[19] Thomas Erl, Anish Karmarkar, Priscilla Walmsley, Hugo Haas, L. Umit Yalcinalp, Kevin Liu, David Orchard, Andre Tost, and James Pasley. *Web Service Contract Design and Versioning for SOA*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2009.

[20] John Evdemon. Principles of Service Design: Service Versioning. http://msdn.microsoft.com/en-us/library/ms954726.aspx, August 2005.

[21] Marios Fokaefs, Rimon Mikhaiel, Nikolaos Tsantalis, Eleni Stroulia, and Alex Lau. An empirical study on web service evolution. In *18th IEEE International Conference on Web Services (ICWS 2011)*, pages 49–56, Washington, DC, USA, July 2011.

[22] David Frank, Linh Lam, Liana Fong, Ru Fang, and Manoj Khangaonkar. Using an Interface Proxy to Host Versioned Web Services. In *Proceedings of the IEEE International Conference on Services Computing (SCC 2008)*, pages 325—332, Honolulu, Hawaii, USA, July 2008.

[23] Tom Gruber. *Encyclopedia of Database Systems*. Springer US, 2009.

[24] Mark Guttenbrunner and Andreas Rauber. A Measurement Framework for Evaluating Emulators for Digital Preservation. *ACM Transactions on Information Systems (TOIS 2012)*, 30(2), March 2012.

[25] Kristina M. Hettne, Katherine Wolstencroft, Khalid Belhajjame, Carole A. Goble, Eleni Mina, Harish Dharuri, David De Roure, Lourdes Verdes-Montenegro, Julian Garrido, and Marco Roos. Best practices for workflow design: How to prevent workflow decay. In *Proceedings of Semantic Web Applications and Tools for Live Sciences (SWAT4LS 2012)*, Paris, France, November 2012.

[26] Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F. Patel-Schneider, and Sebastian Rudolph, editors. *OWL 2 Web Ontology Language: Primer*. W3C Recommendation, 11 December 2012. Available at `http://www.w3.org/TR/owl2-primer/`.

[27] Piotr Kaminski, Hausi Müller, and Marin Litoiu. A design for adaptive web service evolution. In *International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2006)*, pages 86–92, Shanghai, China, May 2006.

[28] Alexander Keller and Heiko Ludwig. The WSLA framework: Specifying and Monitoring Service Level Agreements for Web Services. *Journal of Network and Systems Management*, 11(1):57–81, March 2003.

[29] Heather Kreger. Web services conceptual architecture (wsca 1.0). Technical report, May 2001.

[30] Philipp Leitner, Anton Michlmayr, Florian Rosenberg, and Schahram Dustdar. End-to-End Versioning Support for Web Services. In *Proceedings of the IEEE International Conference on Services Computing (SCC 2008)*, pages 59–66, Honolulu, Hawaii, USA, July 2008.

[31] Jun Li, Yingfei Xiong, Xuanzhe Liu, and Lu Zhang. How does web service api evolution affect clients? In *Proceedings of the 20th IEEE International Conference on Web Services (ICWS 2013)*, pages 300–307, Santa Clara Marriott, CA, USA, June 2013.

[32] Anbazhagan Mani and Arun Nagarajan. Understanding quality of service for web services. Technical report, http://www.ibm.com/developerworks/webservices/library/ws-quality, 2002.

[33] Rudolf Mayer, Johannes Binder Stephan Strodl, and Andreas Rauber. Automatic discovery of preservation alternatives supported by community maintained knowledge bases. In *Proceedings of the 11th International Conference on Digital Preservation (iPres 2014)*, Melbourne, Australia, October 6–10 2014.

[34] Tomasz Miksa, Rudolf Mayer, and Andreas Rauber. Ensuring sustainability of web services dependent processes. *International Journal of Computational Science and Engineering (IJCSE)*. Accepted for publication.

[35] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[36] Marc Novakouski, Grace Lewis, William Anderson, and Jeff Davenport. Best practices for artifact versioning in service-oriented systems. Technical report, January 2012.

[37] Mike P. Papazoglou. The challenges of service evolution. In *Advanced Information Systems Engineering*, volume 5074 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin Heidelberg, 2008.

[38] B.P. Rimal, Eunmi Choi, and I Lumb. A taxonomy and survey of cloud computing systems. In *INC, IMS and IDC, 2009. NCM '09. Fifth International Joint Conference on*, pages 44–51, Aug 2009.

[39] Michael Rosen, Boris Lublinsky, Kevin T. Smith, and Marc J. Balcer. *Applied SOA service-oriented architecture and design strategies*. Wiley Publishing, Inc., 2008.

[40] Yogesh Simmhan, Beth Plale, and Dennis Gannon. A Survey of Data Provenance Techniques. Technical Report IUB-CS-TR618. Technical report.

[41] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. A survey of data provenance in e-science. *SIGMOD RECORD*, 34(3):31–36, September 2005.

[42] Stephan Strodl, Rudolf Mayer, Gonçalo Antunes, Daniel Draws, and Andreas Rauber. Digital preservation of a process and its application to e-science experiments. In *Proceedings of the 10th International Conference on Preservation of Digital Objects (IPRES 2013)*, Lisbon, Portugal, September 2013.

[43] Raymond J. van Diessen. Preservation requirements in a deposit system. Technical report, BM/KB Long-Term Preservation Study Report Series Number 3 Chapter 3, 2002. http://www.kb.nl/sites/default/files/docs/2-authenticity.pdf.

[44] Elisabeth Weigl, Johannes Binder, Stephan Strodl, Barbara Kolany, Daniel Draws, and Andreas Rauber. A framework for automated verification in software escrow. In *Proceedings of the 10th International Conference on Preservation of Digital Objects (IPRES 2013)*, Lisbon, Portugal, September 2013.