Analyzing the Evolution of WSDL Interfaces using Metrics

Master's Thesis

Maria Kalouda

Analyzing the Evolution of WSDL Interfaces using Metrics

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Maria Kalouda
born in Athens, Greece

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Analyzing the Evolution of WSDL Interfaces using Metrics

Author: Maria Kalouda
Student id: 4181409
Email: m.kalouda@student.tudelft.nl

Abstract

Recent studies have investigated the use of source code metrics to predict the change- and defect-proneness of source code. While the indicative power of these metrics was validated for several systems, it has not been tested on Service-Oriented Architectures (SOA). In particular, the SOA paradigm prescribes the development of systems through the composition of services, i.e., network-accessible components. In one implementation of SOA which is very popular in industry, services are specified using WSDL interface descriptions. Thus, service consumers are highly affected by the changes performed on an evolving WSDL interface. This fact reveals the importance of assessing the change-proneness of interfaces in SOA.

This work aims at investigating the correlation between several cohesion and data type complexity metrics and the change-proneness of a WSDL interface. We empirically investigate the correlation between the number of fine-grained interface changes and complexity and cohesion metrics including a newly defined data type cohesion (*DTC*) metric. Furthermore, we perform a manual analysis of the interfaces to gain better insight to our conclusions. We performed these measurements on multiple versions of ten widely used, open-source WSDL interfaces.

Our results show that data type complexity expressed in number of nodes is an appropriate metric to represent data type complexity but not sufficient to predict the change-proneness of an interface. In addition, we investigate three other cohesion metrics: *LCOS*, *SFCI* and *SIDC* presented in the literature and the newly designed *DTC* metric. Our empirical study shows that among the tested metrics it is the *DTC* cohesion metric that exhibits the strongest correlation with the number of fine-grained changes performed in subsequent versions of WSDLs. Finally, based on the *DTC* metric results about the cohesion in data types, we manually analyzed the examined WSDLs and we conclude that highly referenced data types are less change-prone.

Thesis Committee:

Chair:	Prof. Dr. Geert-Jan Houben, Faculty EEMCS, TU Delft
University supervisor:	Dr. Andy Zaidman, Faculty EEMCS, TU Delft
Committee Member:	Dr. Alberto Bacchelli, Faculty EEMCS, TU Delft
Committee Member:	Daniele Romano, Faculty EEMCS, TU Delft

Preface

This thesis document is the research result of my Master's Thesis Project. This work is based on tools being developed by the Software Engineering Research Group (*SERG*) at the Delft University of Technology. As a software developer I am very glad to have the opportunity to study the software improvement field. This work gave me good insight in the state-of-art literature in software engineering and the opportunity to work as a researcher. For this opportunity I would like to thank several people.

Firstly, I would like to thank Dr. Martin Pinzger for initiating me into the world of software improvement and providing me with the opportunity to carry-out this project. A special thank you goes to Dr. Andy Zaidman for the project supervision and his valuable ideas that always pushed me a step further. I am particularly grateful to Daniele Romano for the many, long, horizon-extending discussions we had and his significant contributions to the project. Also I would like to thank Prof. Dr. Geert-Jan Houben and Dr. Alberto Bacchelli for their work as committee members. Finally, I would like to thank my husband Christos Papameletis for his patience and valuable comments while I was working on this report. Last but not least, I would like to say a warm 'thank you' to my family and friends for their support.

Maria Kalouda
Delft, the Netherlands
September 10, 2013

Contents

List of Figures

Chapter 1

Introduction

This chapter presents the context of this study. It also presents the definition of the tackled problem accompanied by the motivation of this work. Next we present a detailed analysis of the research questions investigated in this project and an overview of the findings. Finally, we provide information about the structure of this thesis.

1.1 Problem Definition

Evolution describes the process of gradual development of an entity from a simpler to a more complex form over a period of time. The evolution term is widespread in several domains. Species, cities, ideas, societies evolve over time in order to form more complex structures. The evolution term carries a connotation of gradual improvement in order to adjust to new environments. Different environmental factors affect the process of evolution by making it more or less aggressive in terms of changes in time.

The interpretation of the term evolution in the context of software technology can be approached by two views [2]. In the verbal view the software evolution is studied as an activity that needs to be executed, controlled, managed and improved. This view focuses on how a software system evolves. However, the nounal approach considers the evolution of a software system as a phenomenon seeking its causes, measuring the impact and identifying the characteristics of the evolved system. It is necessary to support both views which are correct and complementary since both views try to achieve a better understanding of the how and why aspects of software evolution [3]. In this study we affiliate the second view of software evolution that helps the industry to address its concerns about efficient software maintenance.

In general, the process of software maintenance and evolution involves high costs and high implementation times for many newly developed systems [4]. Among these systems, Service Oriented Architecture (*SOA*) is one of the most popular architectures for designing web systems because it provides unique features like reuse of common logic with different clients existing in different web environments (desktop, mobile and web applications). Service oriented systems rely on open standards that are inter-operable across different computing platforms [5].

The process of SOA system evolution and maintenance is quite complex and challenging for two reasons [6]. First, the distributed nature of services, the parts of which can reside in different servers and organizations makes the maintenance activities more complex. Second, the design of service-oriented systems involves the challenge of designing high quality stable interfaces. In addition, services evolve in order to address new requirements and fix faults. But the service providers do not know the service subscribers a priori in order to assess the impact of a change. Any changes performed on the service implementations and not affecting the service interfaces are completely transparent to its subscribers. On the other hand, the interfaces can be perceived as a contract between service providers and service subscribers. If the interface introduces a breaking change in one revision, every subscriber that uses this interface has to adapt his system.

The goal of this work is to investigate the relationship of data type complexity and cohesion with the change-proneness of service interfaces. Interfaces with high number of changes in future revisions are considered more change-prone. Our instinctive expectations are that increased complexity and less cohesion leads to more change-prone interfaces. We aim at studying the relationship between external interface characteristics (*i.e.* cohesion and data type complexity) and the interface stability.

1.2 Research Context

DeMarco claims that "you cannot control what you cannot measure" [7]. Thus, metrics are a very important medium to control system quality. Software metrics can be perceived as the actual quantitative means to get feedback about the quality of a software system during the implementation, maintenance or even design phase. There is extensive literature about several software metrics which are used to evaluate the system characteristics that are related to the evolution of a software system [8],[9]. Specific metrics for measuring the software quality of *SOA* are slowly starting to appear in literature studies [10],[11],[12],[13].

Note that there are two types of web services: the web services which are based on the HTTP and the REST (Representational State Transfer) protocol, and the SOAP services. This study focuses on SOAP services which are based on Simple Object Access Protocol which is a protocol specification for exchanging structured information. In SOA the service provider creates a web service and describes its interface using Web Services Definition Language (WSDL). Then the provider makes the WSDL of the service available to the service registry (UDDI). The service consumers can select the best service that satisfies their requirements selecting from those available in the UDDI.

For that reason the service consumers need to be able to choose the most stable services and predict the cost and effort associated with adding such a dependency into their systems. As described in the previous section, the goal of this work is to help the developers evaluate the change-proneness of a WSDL interface. To be more specific, we examine the correlation of several data type complexity and cohesion metrics with the number of changes performed on a WSDL throughout its revisions. The data type complexity and the service cohesion are the most important measurable external characteristics of the service interfaces because they do not require implementation details in order to be measured. If we have an indication

that there is correlation between any of these characteristics and the change-proneness of a WSDL, developers can use this conclusion to decide about the change-proneness of a WSDL interface.

In [14] the authors present three complexity metrics based on the tree-representation of data types: the tree depth (*Branching Factor*), the *Number of Nodes* and the *Tree Height*, but they do not provide any evaluation of the applicability of these metrics. In the literature there are also several cohesion metrics: the Service Interface Data Cohesion (*SIDC*) [11], the Lack of Cohesion in Service (*LCOS*) [10] and the Service Functional Cohesion Index (*SFCI*) [10] that express the service cohesion.

Previous work by Romano and Pinzger [15] provides a tool called *WSDLDiff* to extract fine-grained changes from subsequent versions of a web service interface defined in WSDL. We use this work to extract the changes from the WSDLs and correlate them with the values of the metrics.

1.3 Research Questions

The main question addressed in this work is:

"How can we distinguish the more change-prone web service interfaces?"

The answer to this research question are valuable to several actors involved in SOA architectures: software developers, managers and researchers. Our findings may interest service developers and assist them in designing interfaces that are less change-prone but also managers as service producers want to measure the quality of their services. In addition, the service consumers, managers and developers, are very interested in selecting the most stable web-services for their system. This is a broad research question that can be discussed further:

- **RQ1**: *Which complexity metrics can be used to identify the change-prone WSDL interfaces?*
  This research question aims to test the data type complexity metrics defined in [14] with an empirical study and assess their relationship with the data type change-proneness. Also we discuss the usefulness of these metrics to extract conclusions about the change-proneness of WSDL interfaces.

- **RQ2**: *Which cohesion metrics can be used to identify the change-prone WSDL interfaces?*
  The investigation of this research question aims to test cohesion metrics found in the literature ([11], [10]) and evaluate their effectiveness. In the context of this research question we also defined a cohesion metric named Data Type Cohesion (*DTC*) that is based on data type cohesion and not on message cohesion like in the other examined metrics. Another goal that is derived from the investigation of this research question, is to analyze the change proneness of data types after characterizing them as highly or rarely referenced.

The research questions were investigated through an empirical study with analyzing ten publicly available WSDL interfaces. We measured various metrics and we also extracted the number of fine-grained changes using the *WSDLDiff* tool [15]. Then several hypotheses were formulated and statistical tests were performed to validate them.

1.4 Contributions

Through this work we contribute the following:

- We found that there is correlation between the complexity metrics defined in [14] and the change-proneness of data types. The *Number of Nodes* metric exhibits the highest correlation with the number of changes compared to the other complexity metrics. But the data type complexity is necessary in most systems, a fact that reduces the importance of these metrics.
- We proposed a new service cohesion metric called *DTC* that exhibits substantial correlation with the number of fine-grained changes performed on a WSDL.
- Our results show that two cohesion metrics *LCOS* and *SFCI* defined in [10] are not suitable metrics for the examined systems because of the developers' tendency to define separate messages and data types for each service operation.
- Also we evaluate the *SIDC* defined in [11] with an empirical study. This metric does not produce significant results for the examined data set.
- In addition, we examined the change-proneness of the data types in conjunction with the number of times a data type is referenced. We concluded that the highly referenced data types are less change-prone in future revisions.
- We further examined the highly referenced data types and we noticed that they are used by the developers as "building blocks" to compose more complex and therefore less reused data types.

In our data set we also observed that:

- developers are trying to maintain backwards compatibility in their interfaces by declaring new elements as optional.
- in many systems there is at least one revision that involves major maintenance activity.

1.5 Structure

This report is split into seven chapters. The first Chapter starts with the introduction, which describes the problem, the context of this work, the research questions, the goal, the contributions of this work and the structure of this thesis. Chapter 2 provides background knowledge in the research field related to this study: software evolution, SOA and metrics.

Chapter 3 describes prior work conducted in SOA metrics and change-proneness analysis. Chapter 4 describes the research framework, the metrics used in this work and our research plan to examine the posed research questions. Chapter 5 discusses the project selection, the tested hypotheses, the method followed to test them and the results of our empirical study. Chapter 6 presents the answers to the research questions, the implications of the results and potential threats to validity for this study. Conclusions and future work are discussed in the final chapter.

Chapter 2

Background Knowledge

In this chapter background information about the core notions of our study are provided. Section 2.1 discusses the evolution of software systems. Section 2.2 provides background knowledge about Service Oriented Architectures and their unique characteristics. Finally Section 2.3 provides some information about the role of metrics in software development.

2.1 Software Evolution and Change-proneness

Software evolution refers to the dynamic behavior of the systems during maintenance phase. Software evolution is becoming highly important as systems become longer lived [16]. However, anticipating change is not an easy task because there are many reasons why a system evolve. For example a system may need to change because of changes in the underlying problem or in the background environment.

Lehman in 1980 firstly introduced a programs taxonomy in his effort to explain why programs vary in their evolution properties. According to this classification three types of programs are introduced: S, E and P type systems [17]. The well-specified systems belong to the S-type category because their acceptance is based on the conformance with the requirements. The problem is completely defined, and there are specific solutions known beforehand. As Figure 2.1 shows, the problem solved by an S-system is related to the real world which is always subject to change. Nevertheless, if the real world changes, the result is a new problem that needs to be specified from scratch. On the other hand, the E-type systems are those that involve high dependency and interaction with the real world. Since the real world is dynamic these systems are exposed to evolutionary pressures which creates the need to continually adapt them in order to remain consistent with their environment. Figure 2.1 shows the changeability of an E-system and its dependence on its real world context. Since the problem addressed in a E-type system can not be specified completely, the system should be evaluated by its results under actual operating conditions. The third category P (for problem) includes systems that cannot be fully specified and their design is an iterative process that involves compromises from the stakeholders in order to reach a practical solution. However, P category is not so wide-spread and other studies claim that the P-type systems fulfill either the S or E-type categories criteria [18].

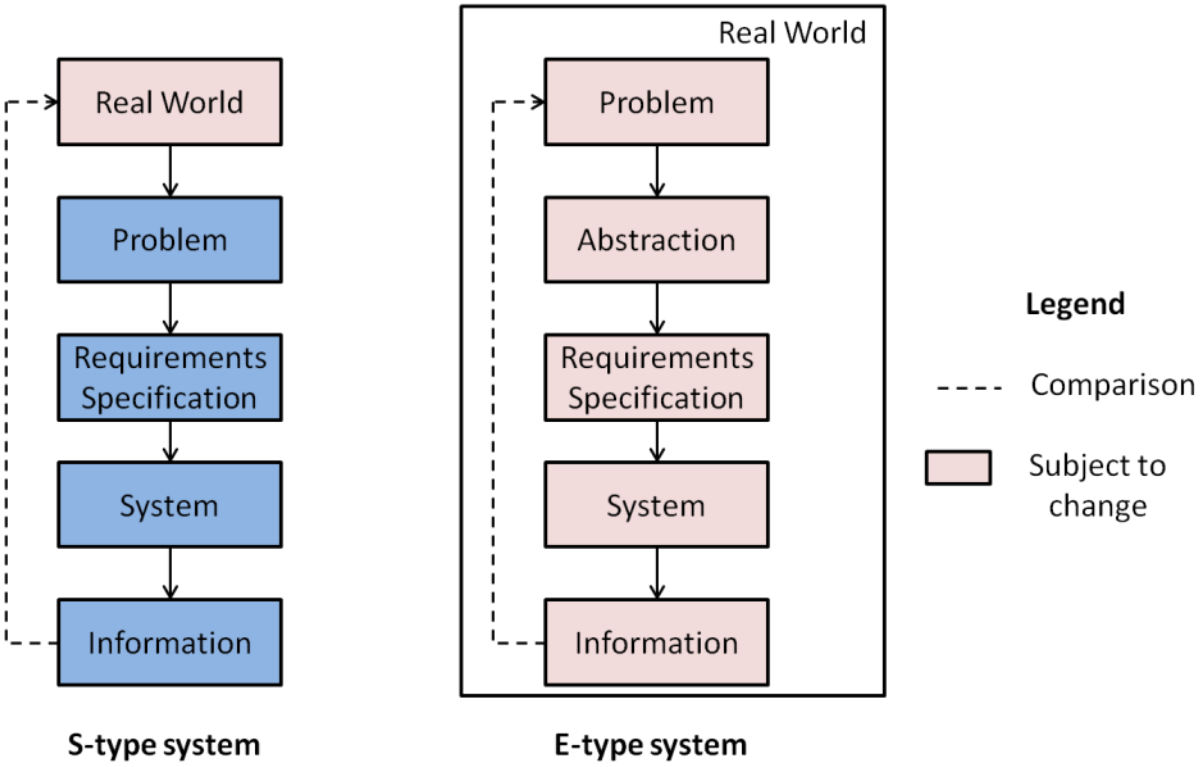

Figure 2.1: S and E type systems [1].

Multiple studies over the 1970s and 1980s based on direct observation and measurement of the evolutionary behavior of a variety of E-type systems, proposed eight laws listed in Table 2.1 [17], [19], [20]. These laws have been empirically successively evaluated during the FEAST 1/ and 2/ studies [21], [22] which were undertaken to further explore the evolution phenomenon. However, it is important to stress that the use of newer programming paradigms like object oriented, component and service oriented paradigms as well as newer process approaches (e.g., open source, agile and extreme programming) introduce new situations and new factors affecting the evolution of the systems. Nevertheless, these laws reflect the fact that as the systems grow in size and complexity, it becomes more difficult to add new functionalities unless countermeasures are taken to re-organize the overall design, this is generally accepted and adopted not only by the science community but also from industry.

The Continuing Change law is very important because it verifies the continuous evolution of E-type systems. There are different types of changes performed in systems. Thus, corrective maintenance is the process that involves changes related to error correction, moreover, adaptive maintenance is the process which is triggered by a hardware or software upgrade and the end product should be updated in order to preserve functionality. Preventive maintenance refers to changes that occur for preventing reasons removing faults before becoming failures, finally, perfective maintenance refers to perfecting changes like comments

No	Brief Name	Law
1 (1974)	Continuing Change	E-type systems must be continually adapted else they become progressively less satisfactory in use.
2 (1974)	Increasing Complexity	As an E-type system is evolved its complexity increases unless work is done to maintain or reduce it.
3 (1974)	Self Regulation Global	E-type system evolution processes are self-regulating.
4 (1978)	Conservation of Organizational Stability	Unless feedback mechanisms are appropriately adjusted, average effective global activity rate in an evolving E-type system tends to remain constant over product lifetime.
5 (1978)	Conservation of Familiarity	In general, the incremental growth and long term growth rate of E-type systems tend to decline.
6 (1991)	Continuing Growth	The functional capability of E-type systems must be continually increased to maintain user satisfaction over the system lifetime.
7 (1996)	Declining Quality	Unless rigorously adapted to take into account changes in the operational environment, the quality of E-type systems will appear to be declining.
8 (1996)	Feedback System (Recognized 1971,formulated 1996)	E-type evolution processes are multi-level, multi-loop, multi-agent feedback systems.

Table 2.1: The laws of Software Evolution.

and documentation update or clarification changes [23].

In this work we measure the change-proneness of systems as the number of fine-grained changes between two revisions. These changes can have corrective, adaptive, preventive or perfective role. In order to study the evolution behavior of an E-type system the nature of changes should be analyzed. We focus on system characteristics like cohesion, complexity, coupling and response time that can be perceived as triggers for future changes.

2.2 Service Oriented Architectures

There are different ways and architectures to create software systems. At the beginning of the new century, a new model has emerged called Service Oriented Computing (SOC) according to which services are the fundamental elements to develop new solutions on the web. Recently, this model has become extremely popular.

The term “service” has been presented in industry for a long time and it has been used in many different ways. According to Papazoglou [24] the services are computational elements that support distributed applications. SOC is a technology that deals with services and addresses the need of loosely-coupled, protocol-independent and self-describing elements. Another unique characteristic of services is that they are independent of the state or the context of other services in their environment. Services can perform any function from a simple request to complicate business process and allow organizations to expose their in-

terfaces to Internet or intra-net using XML based open standards. The basic advantage of SOC is that it enables software to run in distributed and heterogeneous computer platforms and can be implemented using different programming languages. Furthermore, due to characteristics of simplicity and interoperability, the web services also provide the opportunity to be orchestrated in order to create larger and more complex business processes.

SOC relies on Service Oriented Architecture (SOA) to build a service model consisting of interacting services. Many definitions for SOA are available in literature, some claim that it is a technical approach that provides the tools to integrate multi-discipline platforms and promotes the reusability in business tasks [25]. Some others give a broader aspect like in the OASIS reference model [26] which claims that SOA is a design style that "organizes and utilizes distributed capabilities that may be under the control of different ownership domains". In general the idea of SOA departs from object oriented programming which suggests that data and its business logic should be bound together [27]. In this work we perceive SOA systems as a subset of E-type of systems that follow the laws of software evolution defined in Table 2.1.

SOA is an architectural paradigm and therefore many implementations of it are possible. In this work we examine the SOA architectures which are implemented using web services. According to *W3C*[1]:
"A web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other web-related standards."

A web service is implemented by a service provider according to the requirements of service consumers. It consists of a contract, one or more interfaces and an implementation. The web service provider is using the Web Services Description Language (WSDL[2]) that describes the provided service and the invocation methods and then the service is registered in a public service registry using the Universal Description Discovery and Integration (UDDI). Service clients can discover the desired services in the registry and obtain a URL to a WSDL. Finally the clients can implement applications that invoke the selected web services according to the defined WSDLs, using an XML-based object access protocol called Simple Object Access Protocol (SOAP). The call can be performed in either asynchronous messaging or using remote procedure call (RPC) event driven mode. The general web services model is illustrated in Figure 2.2.

According to WSDL, a service is defined as a collection of network ports in a WSDL document. A port is defined by associating a network address with a reusable binding and a collection of ports define a service. Thus, a WSDL document uses the following elements in order to describe services:

- Types: the container of data type definitions which are using some type system like XSD.
- Message: a description of the data being communicated.

[1] http://www.w3.org/TR/2004/NOTE-ws-arch-20040211
[2] http://www.w3.org/TR/wsdl

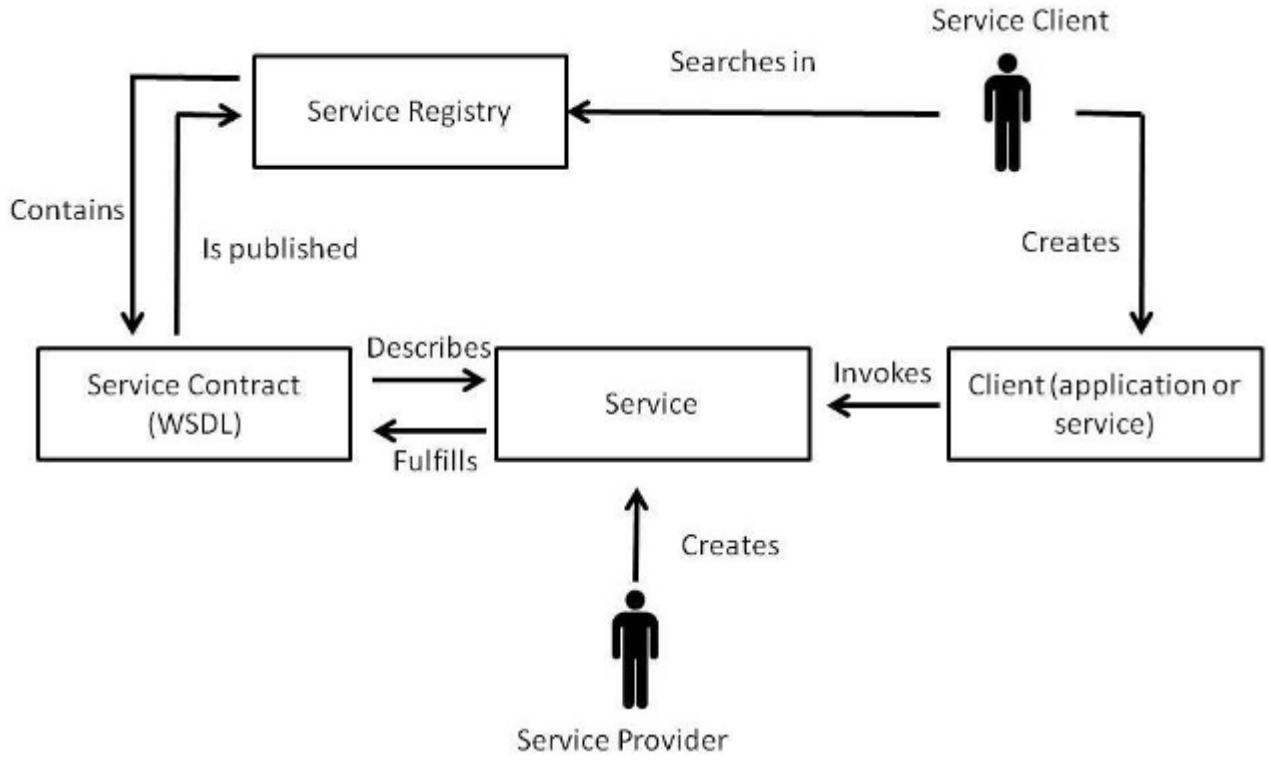

Figure 2.2: The web services model.

- Operation: a description of the actions provided by this service.
- Port Type: a set of operations supported by an endpoint.
- Binding: specification of the protocol and data format for a particular port type.
- Port: a single endpoint defined as a combination of a binding and a network address.
- Service: a collection of related endpoints.

It is important to note here, that WSDL does not introduce a new type definition language but supports the XML Schemas specification (XSD[3]) as a rich type system for describing messages.

Analyzing and dealing with evolution of web services in SOA is a challenging task due to the distributed nature of services [28]. According to the SOA model the service provider may not know a priori the service clients and for that reason the WSDL interfaces are considered as contracts between the two parts. For this reason there is a major requirement for interfaces stability [29]. In addition, the web services are constantly evolving in order to address new requirements, improve performance or fix errors [15].

[3] `http://www.w3.org/XML/Schema`

SOA focuses on exposing business-relevant functionality to end-user applications or other services and tries to achieve an increased level of decoupling of interfaces and implementation. A crucial factor in the development of services is the quest for the right degree of abstraction. There are several principles that help to achieve the goals of SOA [29], [30], [31], [32]:

- *Reusability*: The ultimate goal of services is their reusability by multiple clients in order to decrease the cost and improve flexibility.
- *Loose coupling*: Services should have low number of inter-dependences.
- *Service cohesion*: Developers must thrive to create services which serve one specific business goal with elements which are highly connected. The services with high functional cohesion are low coupled and highly reusable.
- *Composability*: Several services can be orchestrated to implement more complex business goals.
- *Abstraction*: The services hide their internal logic and communicate with their environment using their interface.
- *Autonomy*: Services have the complete control of the business logic they implement.
- *Discoverability*: Services are designed in a descriptive way so that they can be discovered and assessed by service consumers.
- *Statelessness*: Services are designed in order not to store activity specific information.

These principles are mostly prescriptive and there has been little work to be quantitatively measured in practice. In this work we try to assess the stability of service interfaces by examining the cohesion and data type complexity of service interfaces. These characteristics are measured using WSDL interface specific metrics.

Papazoglou [28] classifies the nature of changes performed in services based on the effects they cause:

- *Shallow changes*: the effects are restricted to a single service and the clients of that service.
- *Deep changes*: require that a business process that contains multiple services, be redefined completely.

In this work, we aim at correlating the shallow changes performed single web service interfaces (WSDLs) with specific interface characteristics like data type complexity and service cohesion.

2.3 Metrics

Quantitative measurements are essential to software development as to all sciences. Software metrics have an important role in the state-of-the-practice in software engineering [33]. A software metric can be defined as a measure of some property of a system or its specifications [34]. Metrics have been adopted by several companies in order to better understand, control and assess software products and processes. Software metrics are used to obtain objective reproducible measurements that can be useful for quality assurance, performance, debugging, management, predicting defective code and estimating costs. There are different types of metrics used in industry like product metrics, process metrics, and project metrics. In this work we focus on metrics related to software product quality.

It is important to firstly identify what to measure on a system and secondly to take the correct measurement and then collect the data and extract knowledge. Measurement technology is a separate discipline in software technology [35]. Every measurement procedure results in a measurement scale for the property that is being measured. The different scales can be grouped to several categories: nominal scales are employing semantic expressions to identify objects, typological scales are used in order to categorize already identified objects, ordinal scale measurement involves the value assessment already measured entities, interval scale is used for perceiving increments, ratio scale permits ratio calculation with rational zero reference point and finally the absolute scale is the most unambiguous scale which is unique like counting lines of code.

In the software technology field, there are software engineering processes that play the role of software measurement instruments. There is a variety of tools mostly automated to track data about the software quality like: observation, estimation, review, test, tracking and monitoring, audit, data collection tools, survey forms, assessment questionnaires, experiments, logs *etc*. It is important to note here that classically every measurement may introduce noise which distorts the actual information. This noise can be created from measurement errors, low accuracy or precision.

Moving from measurements to metrics is like moving from observation to understanding. Metrics should be designed in such a way that they reveal a chosen characteristic in a meaningful manner. There is a further classification of the software metrics to internal and external aligned with internal and external quality as suggested in ISO 9124 standard [36]. External metrics are computed by measuring the quality of the system in production phase; internal metrics on the contrary are measured during development phase as a comparison of the required functionality and implemented at a certain time.

Software metrics can be further classified according to the type of their measurement so in that way there is direct, indirect/derived and prediction measurement. Direct measurement always involves an absolute number providing no further knowledge like the number of lines of code. From the other side, indirect/derived measurement is usually a ratio or another number that is derived from an arithmetic operation on one or more metrics like defect density which is the number of defects divided by the total size of software. Finally there is another category that involves prediction for the future like effort prediction in agile methodologies. Another interesting point is that metrics should be designed uniquely for every system in order to be applied directly to the system environment and the needs of

users.

2.4 Summary

This chapter presented some general knowledge for our study. According to Lehman software systems can be classified into S- and E- type based on their evolution activity. The S-type systems are well specified with known solutions at the development time. E-type systems involve high dependency and interaction with the real world. Also, Lehman formulated the law of continuous evolution for the E-type of systems. Systems developed using SOA can be classified as E-type systems because they usually provide solutions for real world problems and they involve high interaction between several actors. For every actor involved in SOA, it is important to investigate the stability of SOA interfaces. Service providers want better quality interfaces and subsequently less maintenance effort. On the other hand, service consumers desire stable interfaces. Software metrics is an important medium in the process of investigating the change-proneness of software. There are different types of metrics depending on the type of measurement and the development phase in which can be computed. In the next chapter we present related with our study work in SOA metrics and system evolution.

Chapter 3

Related Work

This chapter summarizes related work on SOA metrics and the system's change-proneness. It presents a number of tools and approaches about metrics specialized for SOA and evolution analysis in software systems. Each of the studies presented in this section is accompanied by a short description of the approach and the validation technique. Section 3.1 presents work on SOA metrics. Section 3.2 discusses work on change-proneness analysis in general and for SOA.

3.1 SOA Metrics

There is a variety of metrics in literature some of which apply in all types of systems like Source Lines Of Code (SLOC) metric and some others which are specialized for specific types of systems like SOA. The design of SOA is guided by a set of principles that help in achieving the goals of SOA. These principles have been well-documented in the literature [29], [30], [31], [32] and include notions of cohesion, coupling, reusability, composability, granularity, statelessness, autonomy, abstraction and so on. In the below sections metrics for service complexity, coupling, cohesion, granularity and reusability of SOA will be presented.

3.1.1 Service Complexity

Measuring the complexity of software is an important branch in software technology field. Based on complexity metrics developers can assess the developed modules and predict the maintenance effort. There are some traditional complexity metrics such as Lines Of Code (*LOC*), external coupling, and decision points such as McCabe's Cyclomatic Complexity [37]. All of these metrics require to know the system implementation something impossible in SOA from service consumer's perspective.

Thi *et al* [14] proposed a set of metrics to evaluate the complexity of WSDLs. These metrics are based on the complexity involved in data types. They represent the data types as trees and they measure tree characteristics like tree height, branching factor and number of nodes. In this work there is only theoretical validation of the metrics with no empirical study in real life systems. In terms of practical perspective, they conducted a case study

and the results obtained from their metrics are compared to the results of other proposed metrics.

3.1.2 Service Cohesion

In all types of systems, cohesion measures the degree to which the elements of the system belong together [38]. This definition is quite generic and can be applied to different types and levels of encapsulation like module, class, component or service with the necessary adaptations. High cohesion provides designs which can be better tested, have better stability and eventually are better maintainable [36]. Three metrics measuring this attribute are discussed in this section: Service Interface Data Cohesion (*SIDC*), Lack of Cohesion of Service Operations (*LCOS*) and Service Functional Cohesion Index (*SFCI*).

Service Interface Data Cohesion (SIDC)

Perepletchikov *et al.* [11] analyzed the impact of service cohesion in the analyzability attribute of service oriented systems. In this work the cohesion notion, as known from procedural and object oriented paradigms, is extended to encapsulate the characteristics of SOA. He proposed eight semantic categories of service cohesion:

- Coincidental: a service contains operations which do not have any semantically meaningful relationships.
- Logical: the service operations provide common functionality such as, for example, data update or retrieval.
- Temporal: the service operations provide common functionality performed within a predefined time period.
- Communicational: the service operations share the same data abstractions.
- External: all the service operations are used by external service consumers.
- Implementation: the operations of a service are implemented using the same implementation elements.
- Sequential: the operations of a service are sequentially connected via input and output values.
- Conceptual: the operations of a service serve a common business functionality.

A metric is developed to capture the different cohesion types named Total Interface Cohesion of a Service (*TICS*) and it is a combination of three other metrics: Service Interface Data Cohesion, Service Interface Usage Cohesion and Service Interface Implementation Cohesion defined in the same work. This metric presents an overall service cohesion based on how related are the operations exposed in a service interface. The evaluation of the theory is done in a small scale controlled experiment with user participants who are trying to complete several tasks related with the analyzability of the system. A weak point is the

small scale of this experiment, the subjectivity involved from the participants whose knowledge depth is not defined as well as the tasks construction for which there is no in-depth explanation on how appropriate are.

Lack of Cohesion of Service Operations (LCOS)

Two different variants of the Lack of cohesion in methods *LCOM* metric [34] are defined by Sindhgatta *et al.* [10]: *LCOS1* according to which pairs of operations that do not contain similar messages are non-cohesive and *LCOS2* that if an operation uses all the messages then it is cohesive. The *LCOS1* has low discrimination power because of it is not normalized and tends to classify most of the services highly cohesive. On the other hand, *LCOS2* is increasing sharply with the increase in the number of operations and therefore most services are not cohesive because increasing the number of operations it becomes impossible a single operation to use all messages.

Service Functional Cohesion Index (SFCI)

The shortcomings of the *LCOS* metric drove Sindhgatta *et al.* to create the *SFCI* metric. This metric is a value between 0 (non-cohesive) and 1 (perfectly cohesive) and indicates a highly cohesive service if all the operations use one common message. The idea behind this is that cohesive services should operate based on a small set of messages relevant to the service. In order to compute this metric authors proposed to extract utility messages (operations inputs/outputs and their data types) from the code to avoid further noise in the metric calculation.

3.1.3 Service Coupling

The second characteristic of SOA is the service coupling which measures how strong the dependencies are and the associations between services and their messages. The service coupling is investigated under the SOA context by Perepletchikov *et al.* [12]. In this study the coupling concept as known from OOP, is redefined in order to be applied to SOA. The authors define nine coupling metrics related to the implementation elements of a service assuming different weights for relationships between elements. The element level metrics are designed in such way to cover the three different types of coupling connections between services:

- *Intra-service*: dependencies coming from elements belong to the same service.
- *Indirect extra-service*: dependencies between elements of a service and an interface of another external service.
- *Direct extra-service*: dependencies coming from implementation elements of two different services.

Finally an aggregation of these metrics is used to define coupling at the service level.

The authors in a later work [39] conducted a controlled experiment to assess the theoretically validated metrics' impact to maintainability. The weakness of this work is that the artifacts used by metrics are analysis of classes, sequences and collaboration diagrams which are not always available.

On the other hand, Sindgatta *et al.* [10] defined except from coupling coming from the direct dependence of a service on another service, the indirect reference on its messages. Accordingly, they propose the Service Message Coupling Index metric in order to quantify the message coupling. In this theory, messages are coupled when operation input messages are processed and then are returned as outputs according to the interface as well as additional messages needed in other service operations invocations.

3.1.4 Service Reusability

The success of SOA architecture is highly dependent on the reusability of the services which should be designed to enable its usage by multiple consumers. Service composition is highly related with reusability and it is a process of composing a system using several services in order to provide specific functionalities. Sindhgatta *et al.* [10] proposed the Service Reuse Index (*SRI*) metric for measuring the number of consumers of a service and the Operation Reuse Index (*ORI*) for measuring the consumers of a specific operation. Finally, the Service Composability Index (*SCOMP*) metric is defined based on the number of compositions to which a service participates distinguishing between predecessor and successor services of the given service.

Choi *et al.* [13] also described metrics to evaluate both atomic and composite services reusability. Firstly, they define reusability as the degree that a service can be reused without much effort and then they identify five main attributes related to services reusability:

- *Business commonality*: it measures the degree that a functionality (or non-functionality) of a service is commonly used by the consumer.
- *Modularity*: it measures the extent that a service provides functionality without relying on other services.
- *Adaptability*: it evaluates the capability of the service to adapt to different consumers.
- *Standard Conformance*: it describes the degree that a service confronts with widely accepted industry standards.
- *Discoverability*: it assesses how easily the service can be found by the service consumers matching their requirements.

For each of these attributes a metric is designed to measure their existence and the resulting formulas has been combined to a unified reusability quality model. However, the discussed metrics require more information than what is provided in the service designs and can be computed only if you know implementation details.

3.1.5 Service Granularity

Another characteristic of SOA is the service granularity which refers to the amount of functionalities encapsulated in a service. Service granularity is closely related to reuse and composability and in that sense a coarse grained service should ideally provide some distinct functions with large number of consumers. As it is described in [29] there are two kinds of granularity: capability and data granularity. Capability granularity is based on the service functionality and data granularity refers to data used to provide this functionality.

Two metrics are defined by Sindhgatta *et al.* [10] following this classification: the Service Capability Granularity (*SCG*) and Service Data Granularity (*SDG*). *SCG* is described based on the number of operations of the service and *SDG* is calculated based on the number of messages exchanged by these processes. For these metrics higher values mean coarser granularity. In the same work the authors are reasoning that small fine-grained operations will be the result of high granularity. This forces the service consumer in order to execute a complicated business process to call multiple operations which is also increasing the number of exchanged messages. For this reason the Process Service Granularity, Process Operation Granularity and Depth of Process Decomposition metrics are defined for every business process.

Xiao [40] presents a granularity metric which is based on the mutual information content of service operations and their usage occurrences. This metric groups operations that are used together into a single service. In order to show the applicability of this metric, they perform a small scale case study with no empirical evaluation.

3.2 Change-Proneness Analysis

Over the last decades, researchers developed many tools to analyze the changes performed in software systems in their attempt to investigate the causes of system's evolution [41], [42], [43]. Xing *et al.* [42] present a tool called *UMLDiff* to detect structural changes among UML5 design diagrams for Object Oriented Systems. This algorithm produces as output a tree of structural changes that reports the differences between the two designs in terms of additions, removals, edits, moves and renamings performed in classes, packages, fields methods and dependencies. Tsantalis *et al.* [43] present a tool called *WebDiff* which is a web-based and generic differencing service, designed to support the comparison of various types of software artifacts (*e.g.* source code, uml diagrams in several versions).

Fluri *et al.* [44] proposed a tree differencing algorithm for fine-grained source code change extraction. The input of this algorithm is abstract syntax trees which are compared. The result is the number of changes and a minimum edit script that can transform one tree to the other. This tool uses the bigram string similarity to match the several source code statements like method invocations and the subtree similarity of Chawathe *et al.* [45] to match source code structures like loops and if statements. Romano *et al.* [41] empirically investigate the correlation between several metrics and the number of fine-grained source code changes in Java interfaces. The number of changes is extracted from the various source code revisions of each examined file using the *ChangeDistiller* provided by Fluri *et al.*

As it is mentioned in the previous chapters, in SOA systems understanding and coping with changes is even more important because of the nature of the services [28]. In Chapter 2, services are classified as E-type of systems because they are very dependent on their surrounding environment that involves high communication between actors. On the other hand, service clients desire stable interfaces in order to avoid extra maintenance effort.

Fokaefs *et al.* [6] analyzed the evolution of web services using a tree-alignment algorithm called *VTracker*. This algorithm is based on the Zhang-Shashas tree-edit distance [46] algorithm, which calculates the minimum edit distance between two trees taking into account different costs for every edit operation (insert, delete, change). In an earlier study [47], this algorithm had also been applied in web-service specifications. *VTracker* represents WSDLs with an intermediate XML representation to reduce the verbosity of the WSDL language. To be more specific, this algorithm replaces the data type references with the data types themselves. The output of their analysis is the percentage of added, changed and removed elements between subsequent versions of WSDL interfaces. They also performed an empirical study in open source WSDL interfaces to show that *VTracker* is a helpful tool in evolution analysis of WSDLs.

Romano *et al.* [15] proposed the WSDL*Diff* tool to extract fine-grained changes from subsequent revisions of WSDLs. They use the Eclipse Modeling Framework (*EMF*) to compute changes between WSDL models. The main differences with the Fokaefs' work are that they take into account the syntax of WSDL and XSD and they do not replace references with the actual types avoiding multiple times detection of the same changes. In addition, they report the types of elements affected by the changes (*e.g.* operation, message, type) and the type of change in greater detail (*e.g.* element move, attribute value update, type addition).

Wang *et al.* [48] analyze the evolution of dependencies among services using an impact analysis model. In this work, they construct the intra-service relation matrix for each service and the extra-service matrix each pair of services. In that way they calculate the impact effect caused by a change in a specific service. The defined matrices support service changes like additions, deletions and modifications. In addition, Aversano *et al.* [49] present an approach, based on Formal Concept Analysis, to understand how the relationships between two sets of services evolve. They used the concept lattice notion to classify the services based on their relationships. As the service evolves, the service relationships are also changing and thus position of the service in the lattice changes. In that way, changes in the service and its relationships are identified.

There are several approaches to classify the changes performed in service interfaces. Feng *et al.* [50] and Treiber *et al.* [51] propose to classify the changes performed in web services based on their impact to the different stakeholders (*e.g.* developers, providers, brokers, end users, service integrators) who interact in services ecosystems.

3.3 Summary

In this chapter we presented prior work performed in metrics specialized for SOA and change-proneness analysis. SOA metrics can be classified based on the principle they serve.

Thus, we present metrics to measure the complexity of a web service interface, the cohesion of its elements, the coupling between services, the reusability and granularity of a service. Most of the presented metrics lack of a proper empirical evaluation in real life systems. The metrics for service complexity and cohesion will be discussed in further detail in the next chapter since they are empirically evaluated in our study.

In the literature, there is a variety of tools to analyze the changes performed in several revisions of software components. The WSDL*Diff* tool will be discussed further in the next chapter because it is a part of our system to extract fine-grained changes from subsequent revisions of WSDLs.

Chapter 4

Research Framework

The goal of this work is to investigate the relationship between several software metrics and the change-proneness of WSDL interfaces. This chapter describes and explains the steps of our research approach. Section 4.1 provides a schematic overview of our research framework. Then, every step is described in detail: Section 4.2 refers to the tested metrics, Section 4.3 gives details about the process of changes extraction and Section 4.4 describes the tools used to perform the correlation analysis.

4.1 Overview

Figure 4.1 shows an overview of the approach used in studying the change-proneness of service interfaces. Our plan consists of three major steps. The inputs for our study are subsequent versions of WSDLs originating from the same web services. The first step is to measure the data type complexity and cohesion metrics described in Section 4.2 in each WSDL revision. Next, we measure the number of fine-grained changes between subsequent revisions of WSDLs. Then, we use the WSDL*Diff* tool in order to analyze the changes between two revisions as it is described in Section 4.3. Furthermore, we refine information coming from these two steps according to the type of the tested hypotheses and then we combine the results into one unified comma separated file (csv). This file contains information that differs depending on the examined hypotheses. It has variables for each revision like several metrics values, the number of changes in each revision with the next one or the number of all future changes performed in a WSDL. This csv file is used as input for the correlation analysis. The correlation analysis is performed in order to test the correlation between variables according to several hypotheses described in the next chapter. The correlation tests reports the estimate of the correlation (*rho*) and the significance of the test (*p-value*). From these results we can conclude about the acceptance of the examined hypotheses.

The first two steps are implemented as plugins of the Eclipse development platform using the Eclipse Modeling Framework[1] (*EMF*). *EMF* is a modeling framework that helps developers to build tools and other applications using a structured data model and code

[1] `http://www.eclipse.org/modeling/emf/`

generation facilities. This framework provides tools to produce a set of Java classes from a model specification as well as a set of adapter classes which enable model viewing and editing. Following the *EMF* the developers describe the models using some Eclipse meta-models called Ecore. The WSDL*Diff* tool which is used to extract the fine-grained changes, is implemented as Eclipse plugin based on *EMF* and the metrics computation is added as extension to this plugin.

The correlation analysis is performed using the R software environment [52]. R is a high-level language and R sudio is an environment for data analysis and graphics. The R language is very popular among statisticians and data miners for developing statistical software and data analysis. It supports a wide variety of statistical and graphical techniques like linear and nonlinear modeling, classical statistical tests, classifications and clustering. In this work we used the Spearman's rank correlation test for statistical dependence implemented natively in R environment.

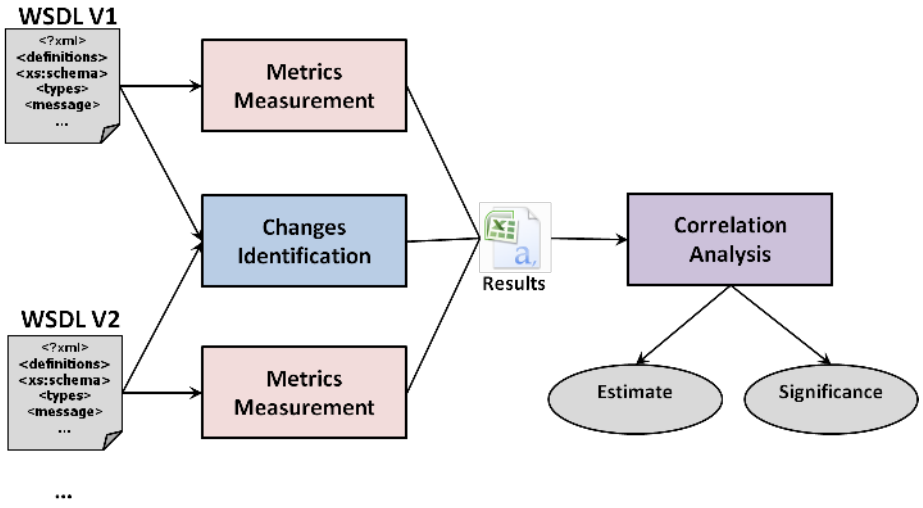

Figure 4.1: Overview of our research plan about the change-proneness of web service interfaces.

4.2 Metrics Computation

The computation of the metrics consists of a three step process. First, several versions of WSDLs coming from selected projects are imported to eclipse platform. The metrics are computed in every revision of a WSDL. Thus, every WSDL is parsed using the *org.eclipse.wst.wsdl* and *org.eclipse.xsd frameworks*. The output of this stage is an *EMF* model that represents the original WSDL file. In the next stage, we extract information from the *EMF* model about the data types, messages and the operations defined in a service interface. We compute two types of metrics described in the below sections: data type complexity and service cohesion metrics.

4.2.1 Data Type Complexity Metrics Computation

As mentioned in Chapter 2 the WSDL file provides information about data types used in a web service. These data types can be XSD types or complex types. For each type element we build a tree structure that represents the data type. Every tree node contains information about the name and type of the node. If a child node represents an XSD type, then it is a leaf node. On the contrary, if a child node represents a complex type then it is a root node of the sub-tree of that type. This process is performed recursively for all types defined in the WSDL. In Listing 4.1 below you can see some data types declared in FedEx ShipService (they have been truncated for demonstration purposes) and Figure 4.2 shows the corresponding data type tree. In case of recursion, *ie* a complex type A has a reference to another complex type B which has a reference to type A, we stop the recursion and we do not count the extra nodes.

Listing 4.1: Example of types definition in FedEx ShipService

```
<xs:complexType name="ProcessShipmentReply">
    <xs:sequence>
        <xs:element name="HighestSeverity" type="ns:NotificationSeverityType"/>
        <xs:element name="Notifications" type="ns:Notification" maxOccurs="
            unbounded"/>
        <xs:element name="TransactionDetail" type="ns:TransactionDetail" minOccurs
            ="0"/>
        <xs:element name="Version" type="ns:VersionId"/>
    </xs:sequence>
</xs:complexType>
<xs:simpleType name="NotificationSeverityType">
    <xs:restriction base="xs:string">
        <xs:enumeration value="ERROR"/>
        <xs:enumeration value="FAILURE"/>
        <xs:enumeration value="NOTE"/>
        <xs:enumeration value="SUCCESS"/>
        <xs:enumeration value="WARNING"/>
    </xs:restriction>
</xs:simpleType>
<xs:complexType name="Notification">
    <xs:sequence>
        <xs:element name="Severity" type="ns:NotificationSeverityType"/>
        <xs:element name="Source" type="xs:string"/>
        <xs:element name="Code" type="xs:string" minOccurs="0"/>
        <xs:element name="Message" type="xs:string" minOccurs="0"/>
        <xs:element name="LocalizedMessage" type="xs:string" minOccurs="0"/>
        <xs:element name="MessageParameters" type="ns:NotificationParameter"
            minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="NotificationParameter">
    <xs:sequence>
        <xs:element name="Id" type="xs:string" minOccurs="0"/>
```

```
                <xs:element name="Value" type="xs:string" minOccurs="0"/>
        </xs:sequence>
</xs:complexType>
<xs:complexType name="TransactionDetail">
        <xs:sequence>
                <xs:element name="CustomerTransactionId" type="xs:string" minOccurs="0"/
                    >
                <xs:element name="Localization" type="ns:Localization" minOccurs="0"/>
        </xs:sequence>
</xs:complexType>
<xs:complexType name="Localization">
        <xs:sequence>
                <xs:element name="LanguageCode" type="xs:string"/>
                <xs:element name="LocaleCode" type="xs:string" minOccurs="0"/>
        </xs:sequence>
</xs:complexType>
<xs:complexType name="VersionId">
        <xs:sequence>
                <xs:element name="ServiceId" type="xs:string" minOccurs="1" fixed="ship"/
                    >
                <xs:element name="Major" type="xs:int" fixed="5" minOccurs="1"/>
                <xs:element name="Intermediate" type="xs:int" fixed="0" minOccurs="1"/>
                <xs:element name="Minor" type="xs:int" fixed="0" minOccurs="1"/>
        </xs:sequence>
</xs:complexType>
```

Thi *et al* [14] propose a suite of metrics which assesses the complexity of a web service using its WSDL interface. The most important metric defined in this work is the Complexity Based Types (CBT). It is based on the complexity of the data types and has the following formula:

$$CBT = \frac{\sum_{i=1}^{n} c_i}{n} \tag{4.1}$$

where c_i denotes the complexity of the data type i. The complexity can be computed using three different ways: the maximum branching factor of the data type tree, the tree height and the number of nodes. In the example presented in Figure 4.2 the maximum branching factor is 6, the tree height is 4 and the number of nodes is 23. *CBT* metric captures the complexity of a service according to the data types. If the value of this metric is small then the level of complexity of the types defined in the service is low. Thus, the service usability increases.

4.2.2 Service Cohesion Metrics Computation

SOA has several characteristics like service cohesion, coupling and reusability, which can be measured and may be related to the service change-proneness. In this work we focus on service cohesion as it can be measured in interfaces without knowing the underlining service implementation.

Sindhgatta *et al* inspired by Chidamber and Kemerer metrics [34] propose the Lack of Cohesion of Service Operations (*LCOS*) metric which is defined as following: for a service

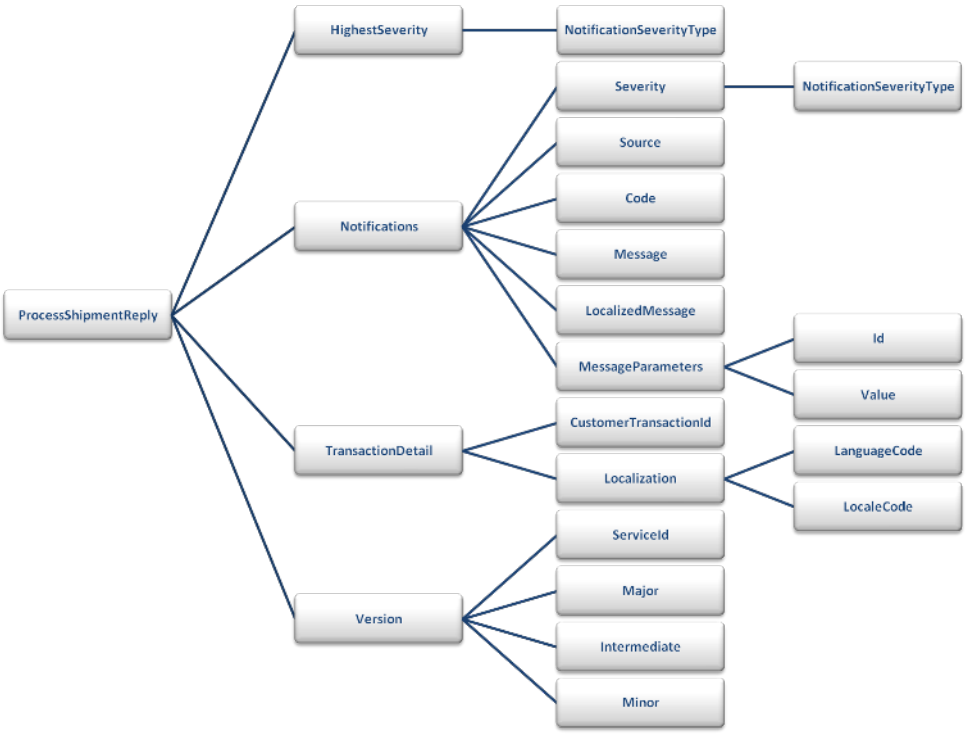

Figure 4.2: Data type tree extracted from the WSDL definitions.

s with a set of operations $O(s)$, let $M(s)$ be the set of messages and the corresponding data types,

$$LCOS(s) = \begin{cases} \dfrac{\left(\dfrac{1}{|M(s)|} \cdot \sum_{m \in M(s)} \mu(m)\right) - |O(s)|}{1 - |O(s)|} & ,|O(s)| > 1 \\ 0 & ,|O(s)| \leq 1 \end{cases} \tag{4.2}$$

It is obvious that $LCOS \in [0,1]$. If every operation uses all the messages then $LCOS = 0$. If each operation uses a distinct message, then the numerator becomes equal to $1 - |O(s)|$ and so $LCOS = 1$.

In practice, the discriminating power of $LCOS$ is low, and most services tend to be classified as low cohesive. For this reason, the authors propose the Service Functional Cohesion Index ($SFCI$) metric. This metric defines the functional cohesion of the operations based on the message that is most widely used across all the operations. It is defined as follows:

$$SFCI(s) = \begin{cases} \dfrac{\max\limits_{m \in M(s)} \mu(m)}{|O(s)|} & ,|O(s)| > 0 \\ 0 & ,|O(s)| = 0 \end{cases} \tag{4.3}$$

where $\mu(m)$ denotes the number of operations using a message m. The value of *SFCI* metric is always between zero that represents a non-cohesive service and one which means a highly cohesive service.

Perepletchikov *et al* [11] proposed eight semantic categories of service cohesion as presented in the previous chapter. The communicational cohesion can directly be quantified from WSDL interfaces. Perepletchikov *et al* propose the Service Interface Data Cohesion (*SIDC*) which quantifies the communicational cohesion and reflects the coincidental and conceptual cohesion.

SIDC is defined as follows:

$$SIDC(s) = \frac{(Common(Param(O(s))) + Common(Return(O(s))))}{Total(O(s)) * 2} \tag{4.4}$$

where $O(s)$ denotes set that contains the operations of a service and $(Common(Param(O(s)))$ is a function that computes the number of service operation pairs that have at least one input parameter in common. Also, $(Common(Return(O(s)))$ is a function that computes the number of service operation pairs that have the same return type and $Total(O(s))$ is a function that returns the number of all possible combinations.

We designed a new metric called Data Type Cohesion (*DTC*) to measure the cohesion of a service taking into account how cohesive the data types are. We define three levels of cohesion in a WSDL file: message, operation and service cohesion. The cohesion between two messages represented by data types m and n, is calculated as follows:

$$C_{dt}(m,n) = \frac{common(elements_m, elements_n)}{common(elements_m, elements_n) + different(elements_m, elements_n)} \tag{4.5}$$

where $common(elements_m, elements_n)$ is a function that returns the number of common nodes in the elements of the two message trees. The nodes are compared based on the element name and element type. The $different(elements_m, elements_n)$ is a function that returns the number of different nodes. The cohesion between two operations o and w is calculated as follows:

$$C_O(o,w) = \frac{\sum_{m,n \in MP(s)} C_{dt}(m,n)}{MP(s)} \tag{4.6}$$

where $MP(s)$ denotes the set of all possible pairs of data types of the message parts including parameters and return types. Finally the service cohesion is defined as:

$$DTC(s) = \frac{\sum_{o,w \in OP(s)} C_O(o,w)}{OP(s)} \tag{4.7}$$

where $OP(s)$ denotes the set of all possible pairs of operations of the service s.

DTC value is bound in $[0,1]$ with 0 means complete lack of cohesion and 1 that the interface is fully cohesive. The interface is fully cohesive when all operations use the same data types-the intuition here is that a cohesive service typically operates on a small set of key objects relevant to that service, so these objects should appear in most of its operations. A service is not cohesive if every operation uses a different data type which has no common

elements with the others. We define that *DTC* is 0 when the WSDL contain less than two operations. According to our view ,the source of change-proneness comes from the data types and this metric captures the cohesion of the data types by investigating how they share elements with each other.

4.3 Changes Extraction

For the changes extraction we used the WSDL*Diff* tool proposed by Romano and Pinzger [15]. To be more specific, the WSDL*Diff* tool is used to extract fine-grained changes from subsequent versions of a web service interface defined in WSDL. It is based on Eclipse Modeling Framework which provides an *EMF* Compare plug-in for model comparison and an Ecore meta-model for WSDL interfaces. Thus, the *EMF* Compare is used to compare instances of WSDL models representing WSDL interfaces.

Figure 4.3 shows the process of extracting fine-grained changes between two versions of WSDLs as it is implemented in WSDL*Diff*. It contains four stages:

- Stage A: The two versions of a WSDL interface are parsed using the org.eclipse.wst.wsdl and org.eclipse.xsd APIs. The output are two *EMF* models corresponding to the initial interfaces.
- Stage B: In this stage the *EMF* models are transformed to the corresponding XSD to improve the accuracy of the results.
- Stage C: In the third stage, the Matching Engine of the *EMF* compare framework is used to identify the matching nodes of the two models.
- Stage D: The Match Model produced in Stage C is fed into the Differencing Engine provided by *EMF* compare in order to identify the differences among the two WSDL models. The result of this step is a tree of structural differences in the nodes of the two models. The differences are reported as additions, removals, moves and modifications.

In this work we use the changes reported by WSDL*Diff* in the leaf nodes in order not to have duplicate entries. The metrics computation process has been built as an extension of the WSDL*Diff* tool.

4.4 Correlation analysis

Dependence in the statistics context refers to any kind of statistical relationship between two random variables. In addition, correlation refers to any class of statistical relationships that involves dependence. Correlation analysis is a powerful means for identifying dependence between two phenomenally random data sets. It is broadly used in several domains like biology, medicine and empirical software.

There are several correlation coefficients to measure the degree of correlation. The commonest of these is the Pearson correlation coefficient, which is applicable only to find

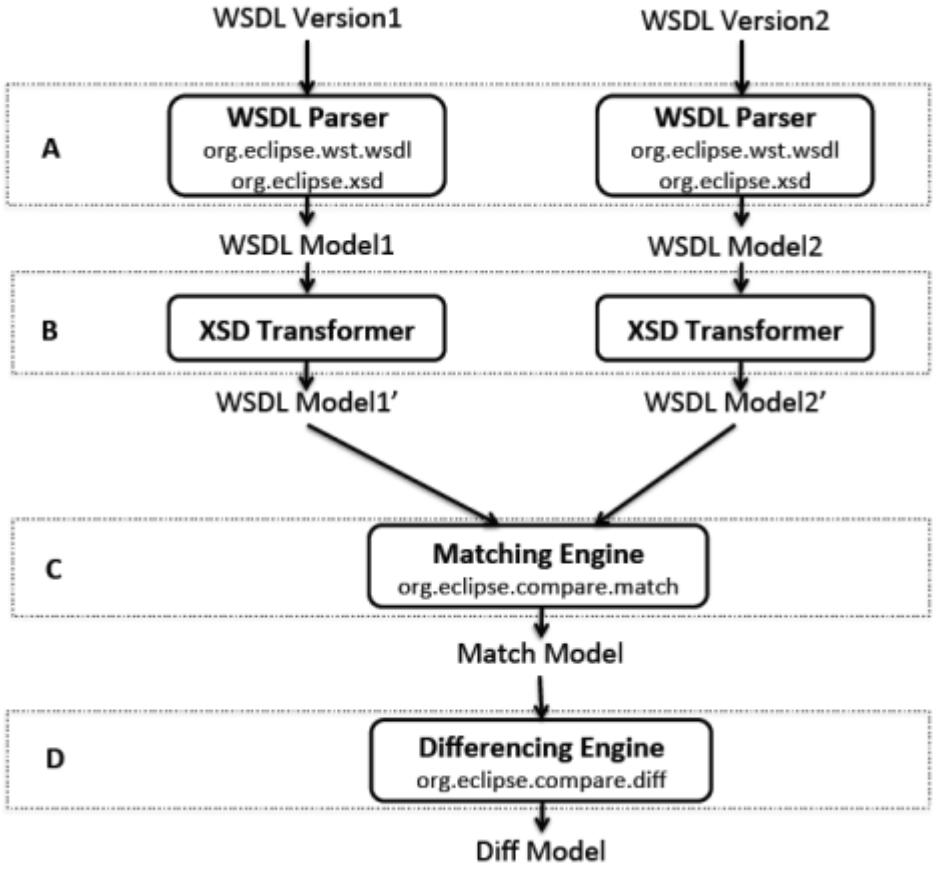

Figure 4.3: The process implemented in WSDL*Diff* to extract fine-grained changes between two versions of a WSDL interface.

a linear relationship between two variables. There are some other coefficients which have been designed to be more sensitive to nonlinear relationships like Spearman's rank correlation coefficient.

Spearman's rank correlation coefficient or Spearman's *rho* is named after Charles Spearman and it is often denoted with Greek letter ρ (rho). It is a non-parametric measure of statistical dependence and it assesses how well a relationship can be described using a monotonic function. Figure 4.4 shows examples of monotonic and non-monotonic relationships. In order to describe a relationship as monotonic it should have one of the following: (a) as the value of one variable increases, so does the value of the other or (b) as the value of one variable increases, the other variable value decreases. The Spearman's rho is a value in $[-1, 1]$ depending on how perfect the monotone function is. It is appropriate for both continuous and discrete variables.

The test except Spearman's *rho* it also reports the statical significance of the results. The statistical significance denoted as *p-value* is related with the confidence the researchers should have on their findings. The *p-value* as defined by Ronald Fisher is the probability

that the observed data would occur by chance in a given single "null hypothesis". In our case the null hypothesis is that there is no relationship between two measured phenomena.

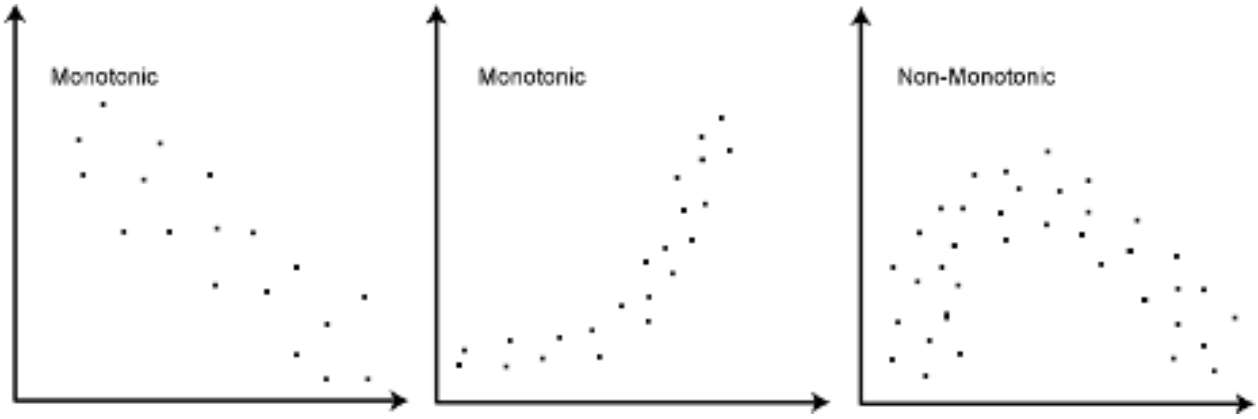

Figure 4.4: Examples of monotonic and non-monotonic relationships.

In this work we use the Spearmans rank correlation. This type of correlation was chosen because in contrast to Pearson correlation, it does not require any assumptions about the data distribution, variances and relationship types [53].

The level of test significance *(p-value)* that was used in this study is the *p-value* to be less than 0.01. If the *p-value* of a correlation test is larger than 0.01, we assume that there is no significance in the test and we ignore its results.

The Spearman's rank correlation coefficient (*rho*) indicates the kind of the correlation between the two examined variables . Tests with *rho-values* which are greater than 0.3 or less than -0.3, are considered to have some correlation using the thresholds defined by Hopkins [54]. Table 4.1 presents the interpretation of *rho-values* for our study.

Positive Correlation Value	**Negative Correlation Value**	**Type**
0	0	No correlation
< 0.3	> -0.3	Weak correlation
$[0.3, 0.5)$	$(-0.5, -0.3]$	Substantial correlation
≥ 0.5	≤ -0.5	Strong correlation
1	-1	Perfect correlation

Table 4.1: Interpretation of the *rho-values* for the Spearman tests.

4.5 Summary

In this chapter we presented our research framework to study the change-proneness of WSDL interfaces. It contains three major steps: metrics computation, changes extraction in subsequent revisions and correlation analysis. We represent the data types defined in a WSDL as a tree and we test three data type complexity metrics: number of nodes, tree

height and maximum branching factor. In addition we compute four cohesion metrics, three of which can be found in the literature (*LCOS*, *SFCI* [10] and *SIDC* [11]) and one that we designed to capture the cohesion inside the data types called *DTC*. In the next step we use the WSDL*Diff* tool to compute the number of fine-grained changes between subsequent WSDL revisions. The metrics and changes computation are implemented as an Eclipse plugin using the EMF framework. The metrics values and the changes info are combined and exported in a csv file which next is imported to R studio for the correlation analysis. Then we perform correlation analysis using Spearman's correlation coefficient in order to test several hypotheses described in the next chapter.

Chapter 5

Analysis and Results

In this chapter we present the correlation analysis performed in order to answer the following research questions defined in Chapter 1:

RQ1: Which complexity metrics can be used to identify the change-prone WSDL interfaces?

RQ2: Which cohesion metrics can be used to identify the change-prone WSDL interfaces?

Section 5.1 introduces the data set of our empirical study. Section 5.2 addresses *RQ1* presenting the hypotheses, the method and the results obtained by testing the various complexity metrics. Similarly, Section 5.3 presents the results for *RQ2*. Section 5.4 contains some additional tests we performed in order to further investigate the data type cohesion. Finally, in Section 5.5 we motivate our results by manually inspecting the changes performed on the examined WSDL interfaces.

5.1 Data Set

The context of our research consists of multiple revisions of ten publicly available WSDLs. Five WSDLs are provided by *Amazon*[1], four by *FedEx*[2] and one by *eBay*[3]. A general description of the core-activities provided by these services is given below:

- *Amazon Elastic Compute Cloud (Amazon EC2)*: is a web service that provides scalable computer power in the Amazon Cloud. It allows its users to obtain, configure and control several computing resources. In this study 22 versions are analyzed.

- *Amazon Flexible Payments Service (AmazonFPS)*: is a web service that allows businesses to charge customers under Amazon's Payments Platform. The customers can

[1] `http://aws.amazon.com/`
[2] `http://www.fedex.com/us/developer/`
[3] `https://go.developer.ebay.com/`

use the same login credentials, shipping address and payment information they already have as Amazon's clients. It supports selling goods, services, raising donations and executing recurring payments. In this study 3 different versions are analyzed.

- *Amazon Simple Queue Service (AmazonQueueService)*: is a scalable queue service provided by Amazon. It can be used to transmit any volume of data at any level of throughput and pay on demand. This study includes 4 versions.

- *Amazon Product Advertizing Service (AWSECommerceService)*: is a service that provides programmatic access to Amazon product selection and discovery in order to allow other developers to advertise Amazon products to monetize their websites. In this study 5 versions of this service are analyzed.

- *Amazon Mechanical Turk (AWSMechanicalTurkRequester)*: is a service that gives access to scalable, on-demand workforce. The clients can get results faster by having multiple Workers complete Human Intelligence Tasks (HITs) in parallel. In this study 6 versions of this service are analyzed.

- *eBay*: provides several operations to enterprise businesses in order to gain access to the global marketplace. It gives the option to its clients to create eBay businesses in order to sell their items. It also offers order management and access to read-only data such as searching for items, popular products and profiles. 5 out of the 8 versions of the eBay API are analyzed.

- *FedEx Package Movement Information Service (PackageMovement)*: is a service to check shipping service availability, validate postal codes and service commitments. 4 out of the 5 versions are analyzed in this study.

- *FedEx Rate Service (RateService)*: is a service that can be used by FedEx's clients to request pre-ship rating information and to determine estimated billing quotes. It can also be used for multiple-package shipments. 11 out of the 13 versions are analyzed in this study.

- *FedEx Ship Service (ShipService)*: can be used to create, validate and submit several shipment requests to FedEx. Clients can also delete and cancel shipment and returns requests. In this study 8 out of the 12 versions are analyzed.

- *FedEx Track Service (TrackService)*: is a service used to obtain timely and accurate tracking information of the shipments, request proof of delivery and manage client delivery notifications. In this study 5 out of the 6 versions are analyzed.

These services are chosen because they are widely used, they evolve continuously and different versions are publicly available for our analysis. In addition, some of them are previously used in other empirical studies appearing in the literature ([6], [15]). Details about the characteristics of the WSDLs in terms of number of operations, number of message parts and number of complex types as they are measured in each revision are provided in Tables 5.1 and 5.2. The data in these tables show that the web services evolve differently.

WSDL	Version	Complex Types	Operations	Parts
Amazon EC2	1	60	14	28
Amazon EC2	2	75	17	34
Amazon EC2	3	81	19	38
Amazon EC2	4	81	19	38
Amazon EC2	5	87	20	40
Amazon EC2	6	85	20	40
Amazon EC2	7	111	26	52
Amazon EC2	8	137	34	68
Amazon EC2	9	151	37	74
Amazon EC2	10	157	38	76
Amazon EC2	11	171	41	82
Amazon EC2	12	179	43	86
Amazon EC2	13	259	65	130
Amazon EC2	14	272	68	136
Amazon EC2	15	296	74	148
Amazon EC2	16	326	81	162
Amazon EC2	17	350	87	174
Amazon EC2	18	366	91	182
Amazon EC2	19	390	95	190
Amazon EC2	20	464	118	236
Amazon EC2	21	465	118	236
Amazon EC2	22	467	118	236
AmazonFPS	1	19	29	58
AmazonFPS	2	15	25	50
AmazonFPS	3	18	27	54
AmazonQueueService	1	26	8	16
AmazonQueueService	2	32	11	22
AmazonQueueService	3	51	15	30
AmazonQueueService	4	51	15	30
AWSECommerceService	1	35	23	46
AWSECommerceService	2	35	23	46
AWSECommerceService	3	35	23	46
AWSECommerceService	4	35	23	46
AWSECommerceService	5	35	23	46
AWSMechanicalTurkRequester	1	86	40	80
AWSMechanicalTurkRequester	2	87	40	80
AWSMechanicalTurkRequester	3	89	41	82
AWSMechanicalTurkRequester	4	85	39	78
AWSMechanicalTurkRequester	5	94	41	82
AWSMechanicalTurkRequester	6	102	44	88

Table 5.1: The number of operations, message parts and complex types reported in each revision of the first five examined web services.

WSDL	Version	Complex Types	Operations	Parts
eBay	9	897	156	313
eBay	11	897	156	313
eBay	13	897	156	313
eBay	15	899	156	313
eBay	17	902	156	313
PackageMovement	2	15	2	4
PackageMovement	3	15	2	4
PackageMovement	4	15	2	4
PackageMovement	5	15	2	4
RateService	1	43	1	2
RateService	2	47	1	2
RateService	3	53	2	4
RateService	4	68	1	2
RateService	5	69	1	2
RateService	6	96	1	2
RateService	7	109	1	2
RateService	8	121	1	2
RateService	9	123	1	2
RateService	10	125	1	2
RateService	13	140	1	2
ShipService	2	74	1	2
ShipService	5	109	9	16
ShipService	6	109	9	16
ShipService	7	119	7	12
ShipService	8	131	7	12
ShipService	9	144	7	12
ShipService	10	146	7	12
ShipService	12	166	7	12
TrackService	2	29	3	6
TrackService	3	29	3	6
TrackService	4	31	4	8
TrackService	5	33	4	8
TrackService	6	33	4	8

Table 5.2: The number of operations, message parts and complex types reported in each revision of the last five examined web services.

For example, the number of operations declared in the *Amazon EC2* service is always increasing except for 4 pairs of versions (between 3 and 4, 5 and 6, 20 and 21,22). In contrast, the *AWSECommerceService* or *PackageMovement* services remain stable and do not have any additions/removals in operations or messages. These two services have changes that include only simple element additions/removals in existing data types and attribute edits. On the contrary, the *ShipService* follows a different evolution pattern. The number of operations increases in version 5 and then is reduced again in version 7. We can notice that the number of complex types is increasing in the vast majority of the revisions of the services. One possible explanation for this can be the addition of new functionality in the services. Finally, as it will be shown in the next sections, one important observation is that in most of the cases the number of message parts is equal to the number of operations doubled. This indicates that there is one request and one response message part for every operation.

5.2 Data Type Complexity

5.2.1 Hypotheses

The first goal of our research (*RQ1*) is to investigate the relationship between the data type complexity and the change-proneness of a WSDL interface. As mentioned in Chapter 4, in order to measure the data type complexity three metrics based on the tree representation of a data type were used: the maximum *Branching Factor*, the *Tree Height* and the *Number of Nodes*. The correlation between the values of these metrics and the number of changes a WSDL interface undergoes, is analyzed in this study. In order to answer *RQ1* and examine the correlation of the complexity metrics with the number of changes we test the following hypotheses:

H1: The number of changes performed on a data type is correlated with the *Branching Factor* metric.

H2: The number of changes performed on a data type is correlated with the *Tree Height* metric.

H3: The number of changes performed on a data type is correlated with the *Number of Nodes* metric.

5.2.2 Method

In order to verify hypotheses *H1*, *H2* and *H3* we analyzed the correlation between each of the aforementioned complexity metrics and the number of future changes performed on the complex data type. To be more specific, assuming that there are k $(v_1, v_2, ..., v_k)$ versions of a WSDL, the number of future changes of a complex data type is defined as the sum of the changes performed on the data type between the revision it first appears in (i) and all

subsequent revisions:

$$totalChanges(dt) = \sum_{j=i}^{k} changes(dt_j, dt_{j+1}) \tag{5.1}$$

where we denote the data type in version j with dt_j. The number of changes is the dependent variable of the test. The independent variable is the value of each complexity metric as it is calculated for each data type in the first revision it appears in the WSDL.

5.2.3 Results

Table 5.3 shows the correlation results (*rho-values*) for each WSDL in the examined data set. In this test, we correlate the values of the metrics as measured for each complex type in the first revision it appears with all the future changes performed on the same data type. Table 5.3 shows that the *Number of Nodes* metric has the strongest correlation because there are six WSDLs with significant results which also have strong correlation. This implies that the bigger the data type tree, the more change-prone it is in future revisions.

WSDL	Branching Factor	Tree Height	Number of Nodes
AWSECommerceService	**0.677062**	**0.7966628**	**0.7680973**
AWSMechanicalTurkRequester	*N/S*	*N/S*	*N/S*
eBay	0.4726281	0.44496	0.473432
Amazon EC2	0.34724	0.3418404	0.4093327
PackageMovement	**0.7428794**	*N/S*	**0.771911**
RateService	**0.5682955**	**0.5971906**	**0.6181661**
ShipService	0.4855903	0.4747396	**0.521194**
AmazonQueueService	-0.3704376	*N/S*	-0.4143885
TrackService	**0.5586796**	**0.5835859**	**0.6161851**
AmazonFPS	**0.6136275**	**0.7207667**	**0.7552271**
Strong Correlation	5/9	4/7	6/9

Table 5.3: Spearman rank correlation (*rho-value*) between each complexity metric and the total number of future changes. The non significant results (p-*value* ≤ 0.01) are marked with *N/S* and the bold font indicates the results with strong correlation ($rho \geq 0.5$).

There is only one result reported in the *AmazonQueueService* WSDL that conflicts with this conclusion. In this case the bigger the tree, the less change prone it is. This is caused by the developers' tendency to introduce new functionality adding new operations and messages with new root types instead of editing or changing the already existing ones.

The examined complexity metrics exhibit high correlation with the change-proneness in the *AWSECommerceService* and *PackageMovement* WSDLs. As shown in Tables 5.1 and 5.2, both services have relatively stable WSDLs mainly with additions to existing data types and attribute values changes (complexity metrics values are not affected by XML attribute changes). Also the *AmazonFPS* WSDL exhibits high correlation results. We observed that

in the *AmazonFPS* WSDL the number of operations and data types are always increasing between subsequent revisions.

The *Branching Factor* and *Tree Height* metrics are less correlated with the number of changes compared to *Number of Nodes*. *Tree Height* has the lowest correlation with 4 results having strong correlation out of 7 significant tests. *Branching Factor* metric has slightly lower correlation than *Number of Nodes* with 5 strong correlated results out of 9 significant tests.

Based on these observations we conclude that hypotheses *H1* and *H3* are valid since both the *Branching Factor* and the *Number of Nodes* metrics exhibit at least substantial correlation ($rho \geq 0.3$) in the majority of the WSDLs. On the other hand, we can partially accept *H2* because *Tree Height* shows substantial correlation for all significant results.

The examined complexity metrics successfully capture the complexity and size of a data type. However, the results verify something very obvious: the bigger and more complex the data type, the more change-prone it is. Nonetheless, our world includes great complexity and the data types are small-scale representations of real-world concepts. So, most systems require complex data types [55]. For this reason, we subsequently try to use cohesion metrics to analyze the relations between the various data types as a potential cause of change-proneness.

5.3 Service Cohesion

5.3.1 Hypotheses

In general, the software developers aim for systems with high cohesion. The second research question (*RQ2*) aims at examining the relationship between the cohesion of the examined WSDL interfaces and their change-proneness. To answer *RQ2*, we selected the cohesion metrics mentioned in Chapter 4: lack of service cohesion (*LCOS*), service functional cohesion index (*SFCI*), service interface data cohesion (*SIDC*) and our newly defined data type cohesion metric (*DTC*). The correlation between the values of these metrics for each WSDL and its change-proneness is analyzed in this section. In order to answer *RQ2* and examine the correlation of the examined cohesion metrics with the change-proneness of a WSDL we evaluate the following hypotheses:

H1: The *LCOS*, *SFCI* and *SIDC* metrics are correlated with the number of changes performed on WSDLs.

H2: The *DTC* metric is correlated with the number of changes performed on WSDLs.

H3: As the cohesion of a WSDL increases, it becomes less change-prone.

5.3.2 Method

In order to verify the cohesion hypotheses, statistical tests for each cohesion metric are performed. Assuming that there are k versions $(v_1, v_2, ..., v_k)$ of a WSDL, we measure each

cohesion metric in every WSDL revision i and then we compute the total number of changes performed on the WSDL in all future revisions:

$$totalChanges(s_i) = \sum_{j=i}^{k} changes(s_j, s_{j+1}) \tag{5.2}$$

where the service in version i is denoted as s_i. The results are then appended in one file and the number of changes is normalized with the number of the revisions over which they are computed $(k-i)$.

We tried several other normalization factors like the number of lines of code and the number of elements. However, we decided to use the number of revisions because the changes are summarized over all future revisions. The number of changes is the dependent variable of the tests while the independent variable is the cohesion metric value measured in each revision of every WSDL.

5.3.3 Results

The significance (*p-value*) and correlation (*rho*) results of the cohesion tests are illustrated in Table 5.4. This table indicates that the *LCOS* and the newly defined *DTC* metrics have significant results ($p\text{-value} \leq 0.01$) with substantial correlation ($rho \geq 0.3$). On the other hand, *SFCI* and *SIDC* do not to provide any significant results ($p\text{-value} > 0.01$) on the examined data set.

The *LCOS* metric expresses the lack of cohesion and the results show a inverse correlation. This means that greater service cohesion leads to greater number of changes in subsequent WSDL revisions. On the contrary, the *DTC* metric which describes the existence of service cohesion has also inverse correlation with the number of changes indicating that a more cohesive service is less change-prone in future revisions.

Metric Name	p-value	rho
CohesionLCOS	**0.005840496**	**-0.373733**
CohesionSFCI	0.3271726	0.1372233
CohesionSIDC	0.04541098	-0.2760514
CohesionDTC	**0.0077991**	**-0.3616314**

Table 5.4: *p-value* and *rho* reported for each cohesion metric from the correlation tests performed to test the cohesion hypotheses (significant results with substantial correlation are marked with bold font).

Nevertheless, we noticed that the *LCOS* metric has value equal to one in the vast majority of the examined WSDLs. This is happening due to the formula used to compute this

metric:

$$LCOS(s) = \begin{cases} \dfrac{\left(\dfrac{1}{|M(s)|} \cdot \sum_{m \in M(s)} \mu(m)\right) - |O(s)|}{1 - |O(s)|} & , |O(s)| > 1 \\ 0 & , |O(s)| \leq 1 \end{cases} \tag{5.3}$$

where $|M(s)|$ represents the total number of message parts and their constituent data types in the service s and $|O(s)|$ is the total number of operations. In addition, the number of operations using a message m is defined as $\mu(m)$ where $m \in M(s)$. An operation uses a message when its data type is referenced by a message part belonging to that operation. This detail in connection with the fact that in the vast majority of the WSDLs every operation defines new data types for its input and output messages (message part types are not re-used) makes the numerator of the Equation 5.3 equal to the denominator which is $1 - |O(s)|$. This implies that the value of the *LCOS* metric in the majority of the revisions of all examined WSDLs is equal to one.

For the same reasons, the *SFCI* metric ends up being dependent only on the number of operations. As we observe for the *LCOS* metric, this happens because of the formula of the *SFCI* metric:

$$SFCI(s) = \begin{cases} \dfrac{\max\limits_{m \in M(s)} \mu(m)}{|O(s)|} & , |O(s)| > 0 \\ 0 & , |O(s)| = 0 \end{cases} \tag{5.4}$$

This conclusion is verified performing a correlation test between the *SFCI* metric and the number of operations. The results show a *p-value* equal to $2.847248e-43$ and *rho* equal to -0.9899302 which are results with high significance expressing very high correlation.

The main reason why both *LCOS* and *SFCI* do not show substantial correlation is that they are based only on the service messages and not on the rest WSDL. Changes may be performed in several places in a WSDL file and large amount of the elements are describing the data types which are ignored by these metrics. Based on the previous analysis we reject *H1* and we conclude that *H2* is valid. Likewise, we accept the *H3* hypothesis. We reach this conclusion based on the results of the *DTC* metric which represents the WSDL cohesion better and shows that the increasing cohesion leads to less-change prone WSDL interfaces.

The *DTC* metric formula encloses interesting information about how the data types are referenced by other data types defined in the WSDLs. For that reason and in order to examine further the *H3* hypothesis, the correlation between the data type usage and the future changes will be investigated in the next section.

5.4 Data Type Reference

5.4.1 Hypotheses

The *DTC* cohesion metric results revealed a third goal for our research: to further examine the relationship between the way the complex data types are referenced by other data types,

and their change-proneness. To be more specific, we want to test whether the frequency that a complex type is referenced in a WSDL is correlated with the number of changes performed on this data type. In order to explore this relationship, the following hypothesis is formulated:

H1: The complex data types that are more frequently referenced by the other types in the same WSDL are less change-prone.

5.4.2 Method

In order to examine the relationship between the frequency the data types are referenced and their change-proneness, we perform one correlation test for each WSDL of the examined data set. The independent variable of these correlation tests is the total number of references existing in each WSDL revision for each complex type. The dependent variable is defined as the total number of future changes performed in a complex data type in all future revisions of a WSDL. To be more specific, we compute the number of changes performed on a data type in every revision i as the total number of changes performed over all subsequent revisions as long as the number of references remains unchanged:

$$totalChanges(dt_i) = \sum_{j=i}^{s} changes(dt_j, dt_{j+1}) \tag{5.5}$$

where we denote the data type in version i as dt_i and the version in which the number of references changes (increased or decreased) is denoted as s.

5.4.3 Results

The results of the correlation tests performed to evaluate *H1* are presented in Table 5.5. We observe that four WSDLs have significant results ($p\text{-}value < 0.01$), three of which have strong correlation values ($rho \geq 0.5$) (*AWSMechanicalTurkRequester*, *PackageMovement* and *TrackService*) and one of which substantial correlation ($rho \geq 0.3$)(*RateService*).On the other hand, the other web services do not produce any significant results. This is most probably due to the lack of sufficient number of revisions for specific WSDLs and the low number of changes in existing complex data types.

We observe that all the significant results exhibit an inverse correlation. Thus, we can conclude that the more frequently a data type is referenced by other types, the less change-prone it is in future revisions. This is an important finding that will be discussed in the next section. Based on the above analysis we accept hypothesis *H1*.

The next step is to extend our research with manual analysis of the several WSDLs in order to reveal the developer's intentions behind specific changes performed on various complex data types.

WSDL	p-value	rho
AWSECommerceService	0.08928	0.2915316
AWSMechanicalTurkRequester	**2.534e-08**	**-0.5027539**
eBay	0.63842	-0.01568748
Amazon EC2	0.2485	-0.04875922
PackageMovement	**0.005086**	**-0.5123475**
RateService	**1.552e-10**	**-0.4187084**
ShipService	0.0004375	0.1930675
AmazonQueueService	0.3013	-0.1301984
TrackService	**2.403e-05**	**-0.5594746**
AmazonFPS	0.6123	-0.1042486

Table 5.5: *p-value* and *rho* results for each WSDL as reported from the data type reference correlation tests (significant results with substantial correlation are marked with bold font).

5.5 Manual Analysis

To further investigate the relationship between the rarely referenced data types and their change-proneness, we manually analyzed the changes performed in the examined WSDLs. With this analysis we aim at illustrating the types of changes performed in the more change-prone data types in order to motivate the previous correlation results.

For every WSDL in our data set, we analyze the types of changes performed on the rarely and highly referenced data types. We assume that the rarely referenced data types are those referenced only once in a WSDL and the highly referenced data types are those types which have more than one reference. Table 5.6 shows the number of changes performed on the highly and rarely referenced data types defined in the WSDLs. We select the *AWSMechanicalTurkRequester*, the *PackageMovement*, the *RateService* and the *TrackService* WSLDs because they report significant results in the previous data type reference correlation tests presented in Section 5.4. The changes are categorized into seven types according to [15]: element additions, element deletions, minOccurs update, maxOccurs update, fixed value update, reference update and enumeration update.

Table 5.6 shows clearly that the total number of changes performed on the highly referenced data types is significantly smaller than that performed in the rarely referenced types. Based on this table we conclude that the type of change that is performed most frequently on the highly referenced data types is the element addition or deletion (*TrackService*: 5 out of 17 changes, *RateService*: 404 out of 1090 changes, *PackageMovement*: 3 out of 6 changes and *MechanicalTurkRequester*: 61 out of 80 changes). The same conclusion holds for the rarely referenced data types. At this point we must note that there are many highly referenced data types that do not have any change in any revision. This observation means that developers tend not to change the highly referenced data types because that would cause a number of chain changes (ripple effect) in many other elements in the WSDLs.

In order to discover why the rarely referenced data types are more change-prone, we manually analyzed some data types from various WSDLs in our data set. The analyzed

WSDL	Type	Element Additions	Element Deletions	Min Occurs Update	Max Occurs Update	Fixed Value Update	Reference Update	Enumeration Update
Track Service	*HR*	1	4	0	0	4	0	8
	RR	37	30	5	2	28	11	35
Rate Service	*HR*	297	107	129	28	12	103	426
	RR	861	375	223	55	26	386	903
Package Movement	*HR*	0	3	0	0	3	0	0
	RR	2	7	2	0	12	0	7
M Turk Requester	*HR*	50	11	3	0	0	0	16
	RR	262	77	0	0	0	0	14

Table 5.6: Number of changes performed on highly referenced (*HR*) and rarely referenced (*RR*) data types categorized into seven types.

data types were selected based on the frequency they were referenced. From this manual analysis we conclude that the highly referenced data types are the ”building blocks” of the rarely referenced data types. The rarely referenced data types are more complex and consequently less frequently re-used. To be more specific, the rarely referenced data types are composed by other data types, some of which are the highly referenced data types. Very rarely, the highly used data types are used directly as the data type of an operation message.

For example, the *ClientDetail* is a complex data type defined in the *ShipService* WSDL. *ClientDetail* is referenced several times as it is shown in Table 5.7. The manual analysis shows that it contains descriptive data about the client submitting the transaction. Table 5.7 also illustrates the low complexity of this data type. The evolution of this type involves only an element addition in version 5. In addition, there are several other data types like the *DeleteShipmentRequest*, the sub-elements of which include a reference to the *ClientDetail* data type. Therefore, a change to *ClientDetail* automatically affects all the complex types that reference it.

Table 5.8 presents information about the number of changes performed in subsequent revisions and the complexity of the *DeleteShipmentRequest* complex data type which contains a reference to *ClientDetail* and other highly referenced data types. A total of 27 changes was observed. One can notice that this data type is significantly bigger than the *ClientDetail*, with maximum *Branching Factor* equal to 7 and 21-26 tree nodes. The larger *Branching Factor* value verifies that the *DeleteShipmentRequest* data type is composed mainly by other complex data types.

Another observation which resulted from the manual analysis is that the value of the *minOccurs* attribute is equal to one for the vast majority of the newly added elements. This means that developers try to maintain backwards compatibility of their interfaces by making the new elements optional. This practice results in not breaking the clients' systems whenever a new element is inserted. Especially, there are cases where a new element is inserted with *minOccurs* equal to 0, only to be changed to mandatory in the next revision. In addition, we noticed that in 5 out of the 10 examined WSDLs there is at least one revision

Version	# Times Referenced	Tree Height	Branching Factor	Number Of Nodes
2	1	3	3	6
5	9	3	4	7
6	9	3	4	7
7	7	3	4	7
8	7	3	4	7
9	7	3	4	7
10	7	3	4	7

Table 5.7: Information details of the highly referenced *ClientDetail* data type of the *ShipService* WSDL.

Version	# Of Changes	Tree Height	Branching Factor	Number Of Nodes
2	2	4	7	21
5	7	4	7	22
6	2	4	7	25
7	7	4	7	25
8	3	4	7	26
9	6	4	7	26
10	0	4	7	26

Table 5.8: Information details of the *DeleteShipmentRequest* data type of the *ShipService* WSDL.

in which more than 30% of elements are new. This indicates that there are some versions where major maintenance with many element and operation additions occurs. In general we conclude that the developers introduce new functionality while trying to keep backwards compatibility to the existing WSDL interfaces.

5.6 Summary

In this chapter we presented our data set and the conducted experiments. Our goal is to investigate how the data type complexity and cohesion is correlated with the change-proneness of ten publicly available WSDL-interfaces. We used the Spearman rank correlation coefficient to perform a thorough investigation of several formulated hypotheses.

The first experiment investigates the relationship between three complexity metrics (*Branching Factor*, *Tree Height* and *Number of Nodes*) and the number of changes performed on the data types in all subsequent revisions. All complexity metrics exhibit some correlation with the change-proneness. Especially, the *Number of Nodes* metric presents the best correlation compared to the other two metrics. Nevertheless, we conclude that the complexity results verify something we all expect: the bigger the data type, the more change-prone it is.

The second experiment aims to analyze the correlation between four complexity metrics (*LCOS*, *SFCI*, *SIDC* and our newly defined *DTC*) and the change-proneness of a WSDL interface. The results show that the LCOS metric is not very indicative of service cohesion despite the fact that it has substantial correlation. This is due to the fact that developers tend to create separate messages and data types for each operation. The *SIDC* and *SFCI* metrics do not produce any significant results. The newly defined *DTC* metric reports substantial correlation with the number of future changes performed on the examined WSDLs. In general, the experiments show that the more cohesive services are less change-prone.

The third experiment intends to investigate the relationship between the times a data type is referenced by other types and its change-proneness. The results show that the more frequently a data type is referenced, the less change-prone it is. Then, a manual investigation is performed to support the previous results. The manual analysis reveals that the highly referenced complex types are the ”building blocks” of the rarely referenced data types, among a variety of other conclusions about the service provider's common practices.

In the next chapter we will discuss the results and the threats to validity applicable to our research design.

Chapter 6

Discussion

In the previous chapter we presented the results of this empirical study. Based on these results, the formulated hypotheses were accepted or rejected. In this chapter we answer to the research questions defined in Chapter 1 and we discuss the implications of the results from three different perspectives: the service providers, the service consumers and the researchers. We also address the several threats to validity that apply to this work.

6.1 Implications of the results

At this point we have to address our initial research questions:

RQ1: Which complexity metrics can be used to identify the change-prone WSDL interfaces?
RQ2: Which cohesion metrics can be used to identify the change-prone WSDL interfaces?

Our first findings concerning *RQ1* confirm that the data type complexity metrics defined by Thi *et. al.* [14] are correlated with the overall change-proneness of a data type defined in a WSDL interface. Furthermore, after testing several hypotheses we conclude that among the *Tree Height*, the *Branching Factor* and the *Number of Nodes* metrics, the last one has the highest correlation to the data type change-proneness. Besides, all the complexity metric results show that the bigger the data type tree, the more change-prone it is.

Regarding *RQ2*, our cohesion results show that *LCOS* [10] and *SFCI* [10] can not express the cohesion of a WSDL interface. In addition, the *SIDC* metric [11] does not seem to be correlated to the overall change-proneness of the interfaces. On the contrary, the newly defined *DTC* metric exhibits substantial correlation to the total number of future WSDL changes. Furthermore, we came to the conclusion that the more cohesive interfaces are less change-prone.

This conclusion agrees with findings from testing the correlation between the frequency a data type is referenced by other types and the number of future changes. Regarding that, we conclude that the highly referenced data types are less change-prone. Moreover, the manual analysis of the WSDLs revealed that the highly referenced data types are simple

types which remain relatively stable and are used as "building blocks" for the rarely referenced data types. In general, the rarely referenced data types are more complex with a relatively high number of sub-elements compared to the highly referenced data types.

Some other useful observations obtained from the manual analysis of WSDLs. Developers introduce new functionality in some versions while they are trying to keep backwards compatibility for their interfaces. There are cases where elements are introduced as optional in one version and the in future versions changes performed to make them required in messages.

The results of our study have several implications for researchers and software engineers. In SOA there are two types of software engineers involved in the development: the developer who maintains the provided web service (service provider) and developer who incorporates the offered service into his system (service consumer). Except developers, managers are also interested in mediums that help the evaluation of interfaces' stability. In addition, researchers may interested in the results from the perspective of extending this empirical analysis using our findings. Hence, the implications should be discussed under these three different perspectives.

From the perspective of a service provider developer, the complexity results imply that the developers should model their types aiming to reduce the unnecessary complexity. The simpler data types are less change-prone and can easily be reused. Like classes in object oriented programming, the data types should be autonomous and represent real-world ontologies according to their properties. The increased re-usability of the data types leads to increased cohesion that reduces the number of future changes as we discussed. Thus, developers must always target low data type complexity and high cohesion while they design WSDL interfaces. That way they can reduce the risk of interface-breaking changes to their clients. The managers who are the owners of a web service can assess the stability and therefore the quality of the offered WSDL interface using the *Number of Nodes* metric in conjunction with the *DTC* cohesion metric. The practice of keeping backwards compatibility is very profitable for the service consumers because it requires less maintenance effort from their side.

Managers and software developers as service consumers need to be able to choose the more stable web services. Another important aspect for them is to predict the cost and effort that is needed to add a dependency from an unknown web service to their systems. From their perspective, metrics like *DTC*, which are correlated with the change-proneness of interfaces, can be a powerful tool in the web service assessment process. Also, complexity metrics can be used by service consumers in order to assess the data type complexity. However, complexity metrics alone are not indicative of change-proneness. They have to be combined with an assessment of service cohesion.

From researchers' perspective, these are interesting results for several reasons. First, we verify the correlation between the complexity metrics defined in [14] and the data type change-proneness using an empirical study. Second, we present a new cohesion metric that exhibits substantial correlation with the number of changes. Still, we believe that our metric should be further tested in different commercial environments. Finally, researchers may also be interested in extending our research by verifying the results with a qualitative analysis.

6.2 Threats to validity

To support the validity of the conclusions we have to evaluate the validity of our research design and determine if threats are true or implausible. In this section we discuss the several types of threats to validity using the Campbell-Cook-Shadish-Stanley validity framework [56][57]. This framework is a result of studying how research designs can reach incorrect conclusions and it identifies four types of validity threats in which a research design may be more or less vulnerable to:

- *Statistical conclusion validity*: Is the statistical analysis performed correctly in order to identify the correlation between two variables in this study?
- *Internal validity*: Is the relation between the dependent and the independent variables causal or are there any confounding variables?
- *External validity*: Can the conclusions of the study be generalized to other systems?
- *Construct validity*: Are the dependent and independent variables well chosen to reflect the theoretical concepts?

Concerning the statistical conclusion validity threat, we used the Spearman's correlation coefficient which is appropriate for both continuous and discrete variables and does not make any assumptions about the underling frequency distribution of the data, variances and the type of the relationship [53]. Additionally, we used a strict level of significance while analyzing the correlation results ($p\text{-}value \leq 0.01$). Finally, to mitigate the risk of our correlation results having low statistical power, we used several different interpretations in the *rho-values*. That way we differentiate between weak, substantial and strong correlation.

The threats to internal validity are related to the existence of an external variable (confound variable) that directly affects the independent variables. An example of this can be the correlation between the number of operations and the *SFCI* cohesion metric, which was discovered and investigated. Furthermore, both the independent and dependent variables of our analysis were extracted using deterministic algorithms. Thus, the results can be re-produced using the same data set. Furthermore, the risk of a fault in the implemented system should always considered as a threat in research. In order to reduce this risk we implemented a suite of 14 unit tests in order to exhaustively test the measurement of the several metrics.

The threat to external validity is the most serious for this work since every result obtained through empirical studies is threatened by the bias of their datasets [58]. There is a variety of threats mostly related with the selected data set which belong to this category. The first threat is that only open-source WSDL interfaces were tested. In order to mitigate this threat used a diverse data set including different systems of different size coming from different domains. The fact that we used systems developed by big companies which are probably relatively stable, can be also considered as a threat. We believe that this fact reduces the noise in the number of changes because of rapid system changes. In newly formed companies many successive changes happen due to system requirements alterations coming

from business case adjustments. In the future, it would be interesting to validate the results of this study using commercial web services from companies of different sizes.

The threats to construct validity challenge the relationship between the theoretical concepts and their representation in our study. In this work, a threat of this type can be way the changes are measured. This is why we selected to test both the number of fine grained changes between two subsequent revisions as well as the overall number of changes over all revisions. In all the cases we received the same results. Another threat that can be associated with this category, is the measurement of cohesion and complexity with the help of metrics. In order to mitigate this threat we tested a variety of cohesion and complexity metrics applicable to web services appearing in the literature. Furthermore, we designed and tested a new cohesion metric used to represent better the cohesion of a WSDL interface.

6.3 Summary

In this chapter we discussed the contribution of this work and the threats to validity which threaten our research design. We discuss the implication of the results presented in the previous chapter from three different perspectives: service providers, service consumers and researchers. Our conclusions are useful to service providers helping them produce more stable interfaces with high re-usability. From service consumers' perspective, our results are helpful in their selection of the most appropriate service for their systems. On the other hand, researchers can interested in extending this work with a more extensive empirical study that includes commercial interfaces or with a qualitative analysis.

Also we identified several threats to validity categorized them to four categories: internal, external, statistical conclusion and construct validity threats. Some of them are the following: statistical test choice, the correlation and significance levels interpretation, confound variables existence, faults in system's implementation, bias of the examined data set and the various concepts representation. In addition, for each threat we present the measures we took in order to mitigate the risks.

Chapter 7

Conclusions and Future Work

This chapter gives an overview of the contributions of this work. In Section 7.1 we reflect on the results and we draw some conclusions. Section 7.2 discusses some ideas for future work.

7.1 Conclusions

In the previous chapter we presented the answers to the posed research questions. According to our study, metrics can be used as indicators of stability of service interfaces. Complexity and cohesion are two factors that can be measured directly in interfaces without any knowledge of the underlining implementation. To goal was to examine a potential correlation between the interface change-proneness and some complexity and cohesion metrics appeared in the literature.

From the complexity results we conclude that bigger data types lead to more change-prone interfaces in the long-term. From the manual analysis of the data set, we concluded that the complexity of an interface lies in the complexity of its data types. Finally, an important finding is that the data type complexity can be efficiently expressed as the number of nodes in the tree representation of the data type.

Regarding service cohesion, we conclude that metrics based on the messages and their types do not have discriminating power in our data set. This happens because we noticed that a new data type was declared for every input or output message. Our observation that many data types are reused by reference inside the same service lead us to design a new cohesion metric which takes into account the commonality of elements between two data types. This metric shows substantial correlation with the future change-proneness. To be more specific, the results have shown that cohesive interfaces are less change-prone. This means that the more the data types are reused instead of being copied, the less XML elements are needed to change in a potential refactoring.

We went a step further in our tests examining the correlation between the frequency a data type is referenced by other types and the change-proneness of the WSDL. The results have shown that highly referenced data types are less change-prone in long term. By manually analyzing these data types we conclude that the highly referenced types are simple,

small types with low complexity that are used as "building blocks" for the more complex and therefore less reused data types. So the developers' intention should be to split the core data types of the designed services to small autonomous components that can be reused by other data types.

Another valuable result coming from this study is that developers are trying to keep backwards compatibility when they introduce new functionality. This is an observation for the systems we studied. Sometimes, in the same newly-introduced elements developers perform necessary breaking changes in future versions. In that way they give time to their clients to adapt their systems. This is a very valuable practice for the service consumers because they have more time to perform the necessary changes.

We argue that the change-proneness of the service interfaces is dependent on the initial service design. As shown in our study, there are several factors affecting the change-proneness of an interface like the complexity and the design of its constituent data types. According to our research, developers should design their types creating as much as possible, small autonomous data types which can be reused or extent with non-required elements. In that way they will succeed not coping the same types in several places avoiding multiple points of change when new functionality is added.

7.2 Future work

The system we proposed to study the WSDL change-proneness has a lot of potential. The correlation of metrics values with the number of future changes is a good step forward in the assessment of the interfaces changeability. Still, there is room for improvement.

We have selected several open source WSDL interfaces from different communities and businesses but we need to perform the same study on proprietary systems of different sizes. Besides, examining more projects would result in a larger and diverse data set which would reduce some of the threats discussed in the previous chapter. Also we need to make sure that we study interfaces coming directly from the underlying implementation and not some wrapper service interfaces.

Other improvements in this research include investigation of other metrics which can be computed in WSDL interfaces extending the tests we performed. There are coupling metrics which can be computed if we have knowledge about the dependencies between services. Also, there are reusability metrics mentioned Chapter 3 which can be computed with the help of information about the service usage. Furthermore, the co-evolution of some services is an interesting field to be studied. It is very important to examine the role of other SOA characteristics (coupling, granularity, reusability) in the interface change-proneness assessment.

Based on our results, we think that it would be interesting to examine further the evolution of less referenced data types. These types present the higher evolution activity in most WSDLs. Also, it could be interesting to extend our research examining the co-evolution of some data types trying to identify occasions where one single change needs multiple updates in order to be implemented. Another improvement could be the classification of changes to breaking and non-breaking changes for the service consumers. In that way, we

could examine the correlation between metrics and client breaking changes. The existence of anti-patterns is also another factor that can cause high evolution activity and worths to be investigated.

Finally, a qualitative analysis can be used to show the strength of the newly designed cohesion metric in the software development community. Such a study can support the findings of the empirical study but also reveal new hypotheses for interface characteristics which may affect their change-proneness. My team has already started performing a qualitative analysis using questionnaires, the results of which will be published in a later work.

Bibliography

[1] Shari Lawrence Pfleeger. The nature of system change [software]. *Software, IEEE*, 15(3):87–90, 1998.

[2] Nazim H Madhavji, Juan Fernandez-Ramil, and Dewayne Perry. *Software evolution and feedback: Theory and practice*. John Wiley & Sons, 2006.

[3] Meir M Lehman, Juan F Ramil, and G Kahen. Evolution as a noun and evolution as a verb. In *SOCE 2000 Workshop on Software and Organisation Co-evolution*, pages 12–13, 2000.

[4] Keith H Bennett and Václav T Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 73–87. ACM, 2000.

[5] Robert Daigneau. *Service Desing Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. Addison-Wesley, 2011.

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

[7] Tom DeMarco. *Controlling software projects: Management, measurement, and estimates*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1986.

[8] Michele Lanza and Stéphane Ducasse. Understanding software evolution using a combination of software visualization and software metrics. In *In Proceedings of LMO 2002 (Langages et Modèles à Objets*. Citeseer, 2002.

[9] Frank Simon, Frank Steinbruckner, and Claus Lewerentz. Metrics based refactoring. In *Software Maintenance and Reengineering, 2001. Fifth European Conference on*, pages 30–38. IEEE, 2001.

[10] Renuka Sindhgatta, Bikram Sengupta, and Karthikeyan Ponnalagu. Measuring the quality of service oriented design. In *Service-Oriented Computing*, pages 485–499. Springer, 2009.

[11] Mikhail Perepletchikov, Caspar Ryan, and Zahir Tari. The impact of service cohesion on the analyzability of service-oriented software. *Services Computing, IEEE Transactions on*, 3(2):89–103, 2010.

[12] Mikhail Perepletchikov, Caspar Ryan, Keith Frampton, and Zahir Tari. Coupling metrics for predicting maintainability in service-oriented designs. In *Software Engineering Conference, 2007. ASWEC 2007. 18th Australian*, pages 329–340. IEEE, 2007.

[13] Si Won Choi and Soo Dong Kim. A quality model for evaluating reusability of services in soa. In *E-Commerce Technology and the Fifth IEEE Conference on Enterprise Computing, E-Commerce and E-Services, 2008 10th IEEE Conference on*, pages 293–298. Ieee, 2008.

[14] Quynh Pham Thi, Dung Ta Quang, and Thang Huynh Quyet. A complexity measure for web service. In *Knowledge and Systems Engineering, 2009. KSE'09. International Conference on*, pages 226–231. IEEE, 2009.

[15] Daniele Romano and Martin Pinzger. Analyzing the evolution of web services using fine-grained changes. In *Web Services (ICWS), 2012 IEEE 19th International Conference on*, pages 392–399. IEEE, 2012.

[16] Chris F. Kemerer and Sandra Slaughter. An empirical approach to studying software evolution. *Software Engineering, IEEE Transactions on*, 25(4):493–509, 1999.

[17] Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[18] Stephen Cook, Rachel Harrison, Meir M Lehman, and Paul Wernick. Evolution in software systems: foundations of the spe classification scheme. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(1):1–35, 2006.

[19] Meir M Lehman and Juan F Ramil. Rules and tools for software evolution planning and management. *Annals of software engineering*, 11(1):15–44, 2001.

[20] Meir M Lehman, Juan F Ramil, Paul D Wernick, Dewayne E Perry, and Wladyslaw M Turski. Metrics and laws of software evolution-the nineties view. In *Software Metrics Symposium, 1997. Proceedings., Fourth International*, pages 20–32. IEEE, 1997.

[21] M.M. Lehman and V Stenning. Feast/1: Case for support. Technical report, Department of Computing, Imperial College, London, UK, Mar. 1996. Available from the FEAST web site, http://www.dse.doc.ic.ac.uk/ mml/feast/.

[22] M.M. Lehman. Feast/2: Case for support. Technical report, Department of Computing, Imperial College, London, UK, Jul. 1998. Available from links at the FEAST project web site UK, http://www.dse.doc.ic.ac.uk/ mml/feast/.

[23] Norman E Fenton and Shari Lawrence Pfleeger. *Software metrics: a rigorous and practical approach*. PWS Publishing Co., 1998.

[24] M.P. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, pages 3–12. IEEE, 2003.

[25] Daigneau Robert. *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. Addison-Wesley, November 2011.

[26] CM MacKenzie, K. Laskey, F. McCabe, PF Brown, R. Metz, and BA Hamilton. Reference model for service oriented architecture 1.0. oasis soa reference model technical committee, 2006.

[27] Shah Md Emrul Islam. *An Approach to Implement Security in Service Oriented Architecture Using Deception Technique*. ProQuest, 2008.

[28] Mike P Papazoglou. The challenges of service evolution. In *Advanced Information Systems Engineering*, pages 1–15. Springer, 2008.

[29] Thomas Erl. *Soa: principles of service design*, volume 1. Prentice Hall Upper Saddle River, 2008.

[30] Erl Thomas. Service-oriented architecture: concepts, technology, and design. *Prentice Hall*, 31:W3C, 2005.

[31] DJN Artus. Soa realization: Service design principles. *IBM developerWorks(February 2006) http://www-128. ibm. com/developerworks/webservices/library/ws-soa-design*, 2006.

[32] Vinay Kumar Reddy, Alpana Dubey, Sala Lakshmanan, Srihari Sukumaran, and Rajendra Sisodia. Evaluating legacy assets in the context of migration to soa. *Software Quality Journal*, 17(1):51–63, 2009.

[33] Linda Westfall. 12 steps to useful software metrics, 2005.

[34] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476–493, 1994.

[35] C Ravindranath Pandian. Software metrics: A guide to planning. *Analysis, and Application. Auerbach Publications*, 46, 2004.

[36] ISO/IEC 9126-1:2001 ISO/IEC. *Software engineering-Product Quality-Quality Model*. 2001.

[37] Juergen Rilling and Tuomas Klemola. Identifying comprehension bottlenecks using program slicing and cognitive complexity metrics. In *Program Comprehension, 2003. 11th IEEE International Workshop on*, pages 115–124. IEEE, 2003.

[38] Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[39] Mikhail Perepletchikov and Caspar Ryan. A controlled experiment for evaluating the impact of coupling on the maintainability of service-oriented software. *Software Engineering, IEEE Transactions on*, 37(4):449–465, 2011.

[40] Wang Xiao-jun. Metrics for evaluating coupling and service granularity in service oriented architecture. In *Information Engineering and Computer Science, 2009. ICIECS 2009. International Conference on*, pages 1–4. IEEE, 2009.

[41] Daniele Romano and Martin Pinzger. Using source code metrics to predict change-prone java interfaces. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 303–312. IEEE, 2011.

[42] Zhenchang Xing and Eleni Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 54–65. ACM, 2005.

[43] Nikolaos Tsantalis, Natalia Negara, and Eleni Stroulia. Webdiff: A generic differencing service for software artifacts. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 586–589. IEEE, 2011.

[44] Beat Fluri, Michael Wursch, Martin PInzger, and Harald C Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *Software Engineering, IEEE Transactions on*, 33(11):725–743, 2007.

[45] Sudarshan S Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *ACM SIGMOD Record*, volume 25, pages 493–504. ACM, 1996.

[46] Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing*, 18(6):1245–1262, 1989.

[47] Rimon Mikhaiel and Eleni Stroulia. Examining usage protocols for service discovery. In *Service-Oriented Computing–ICSOC 2006*, pages 496–502. Springer, 2006.

[48] Shuying Wang and Miriam AM Capretz. A dependency impact analysis model for web services evolution. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pages 359–365. IEEE, 2009.

[49] Lerina Aversano, Marcello Bruno, Massimiliano Di Penta, Amedeo Falanga, and Rita Scognamiglio. Visualizing the evolution of web services using formal concept analysis. In *Principles of Software Evolution, Eighth International Workshop on*, pages 57–60. IEEE, 2005.

[50] Zaiwen Feng, Keqing He, Rong Peng, and Yutao Ma. Taxonomy for evolution of service-based system. In *Services (SERVICES), 2011 IEEE World Congress on*, pages 331–338. IEEE, 2011.

[51] Martin Treiber, Hong-Linh Truong, and Schahram Dustdar. On analyzing evolutionary changes of web services. In *Service-Oriented Computing–ICSOC 2008 Workshops*, pages 284–297. Springer, 2009.

[52] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2012. ISBN 3-900051-07-0.

[53] Shirley Dowdy, Stanley Wearden, and Daniel Chilko. *Statistics for research*, volume 512. Wiley. com, 2011.

[54] Will G Hopkins. *A New View of Statistics*. Internet Society for Sport Science: http://www.sportsci.org/resource/stats/, 2000.

[55] David Lorge Parnas, Paul C Clements, and David M Weiss. The modular structure of complex systems. In *Proceedings of the 7th international conference on Software engineering*, pages 408–417. IEEE Press, 1984.

[56] William R Shadish, Thomas D Cook, and Donald Thomas Campbell. Experimental and quasi-experimental designs for generalized causal inference. 2002.

[57] Donald Thomas Campbell, Julian C Stanley, and Nathaniel Lees Gage. *Experimental and quasi-experimental designs for research*. Houghton Mifflin Boston, 1963.

[58] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *Software Engineering, IEEE Transactions on*, 33(1):2–13, 2007.