Intelligent Web Services Architecture Evolution Via An Automated Learning-Based Refactoring Framework

by

Hanzhang Wang

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer and Information Science)
in the University of Michigan Dearborn
2018

Doctoral Committee:

Assistant Professor Marouane Kessentini, Chair
Professor William Grosky
Professor Bruce Maxim
Associate Professor Brahim Medjahed
Professor Armen Zakarian
Professor Qiang Zhu

Hanzhang Wang

wanghanz@umich.edu

Acknowledgements

Many people have offered me valuable supports during my Ph.D. journey, I am extremely privileged to have them in my life.

Firstly, I would like to give my sincere gratitude to Dr. Marouane Kessentini, my Ph.D. advisor who, with extraordinary patience and consistent encouragement, gave me the great help by providing me with important advices and insights of great value. As a professor who really cares about his student, he always watches my back and thinks for my future. Under his wise guidance, not only I successfully accomplished my Ph.D., but also learnt how to success, to love your family and to help the people around you.

My heartfelt thanks also go to Dr. William Grosky, Dr. Bruce Maxim, Dr. Brahim Medjahed, and Dr. Qiang Zhu, for their insightful comments and encouragement, but also for the interesting questions which incented me to widen my research from various perspectives.

I want to thank my fellow labmates for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last three years. I also want to thank my friends in US and China, for making my life so vividly beautiful.

A special thank goes to Guozhang Yuan, who truly has faith in me and convinced me to start doing a Ph.D., his wisdom benefits me in so many ways.

Finally, I would like to thank my parents, Jing Wang and Tianxia Zhang, for their endless love and selflessly supporting me spiritually and materially throughout my ten years study in American.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

AC	Automatic Correctness
ANN	Artificial Neural Networks
API	Application Programming Interface
AR	Auto-Regressive
ARIMA	Auto-Regressive Integrated Moving Average
ARMA	Auto-Regressive Moving-Averagedels
AWS	Ambiguous Web Service or Amazon Web Services
BLOP	Bi-Level Optimization Problems
BP	Back-Propagation
CI	CRUDy Interface
CRUD	Create, Read, Update, and Delete
CWS	Chatty Web Service
DSL	Digital Subscriber Line
DWS	Data Web Service
EA	Evolutionary Algorithms
FGWS	Fine grained Web Service
GA	Genetic Algorithms
GOWS	God object Web Service
GP	Genetic Programing
IBEA	Indicator based Evolutionary Algorithm
IDE	Integrated Development Environment
MA	Moving-Average
MC	Manual Correctness
MLP	Multi-Layer Perceptron
MNR	Maybe It is Not RPC
MOEA/D	Multiobjective Evolutionary Algorithm Based on Decomposition
MOGP	Multi-Objective Genetic Programming
NLP	Natural Language Processing
NP-hard	Non-deterministic Polynomial-Time Hard

NSGA	Non-dominated Sorting Genetic Algorithm
OO	Object-Oriented
OOP	Object-Oriented Programming
PCA	Principal Components Analysis
QoS	Quality of Service
REST	Representational State Transfer
RPC	Remote Procedure Call
RPT	Redundant PortTypes
RQ	Research Question
RS	Random Search
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SOC	Service-Oriented Computing
SPEA	Strength Pareto Evolutionary Algorithm
SUS	Stochastic Universal Sampling
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium
WS	Web Services
WSDL	Web service Description Language
XML	Extensible Markup Language

Abstract

Architecture degradation can have fundamental impact on software quality and productivity, resulting in inability to support new features, increasing technical debt and leading to significant losses. While code-level refactoring is widely-studied and well supported by tools, architecture-level refactorings, such as repackaging to group related features into one component, or retrofitting files into patterns, remain to be expensive and risky. Serval domains, such as Web services, heavily depend on complex architectures to design and implement interface-level operations, provided by several companies such as FedEx, eBay, Google, Yahoo and PayPal, to the end-users.

The objectives of this work are: (1) to advance our ability to support complex architecture refactoring by explicitly defining Web service anti-patterns at various levels of abstraction, (2) to enable complex refactorings by learning from user feedback and creating reusable/personalized refactoring strategies to augment intelligent designers' interaction that will guide low-level refactoring automation with high-level abstractions, and (3) to enable intelligent architecture evolution by detecting, quantifying, prioritizing, fixing and predicting design technical debts.

We proposed various approaches and tools based on intelligent computational search techniques for (a) predicting and detecting multi-level Web services antipatterns, (b) creating an interactive refactoring framework that integrates refactoring path recommendation, design-level human abstraction, and code-level refactoring automation with user feedback using interactive mutli-objective search, and (c) automatically learning reusable and personalized refactoring strategies for Web services by abstracting recurring refactoring patterns from Web service releases.

Based on empirical validations performed on both large open source and industrial services from multiple providers (eBay, Amazon, FedEx and Yahoo), we found that the proposed approaches advance our understanding of the correlation and mutual impact between service antipatterns at different levels, revealing when, where and how architecture-level anti-patterns the quality of services. The interactive refactoring framework enables, based on several controlled experiments, human-based, domain-specific abstraction and high-level design to guide automated code-level atomic refactoring steps for services decompositions. The reusable refactoring strategy packages recurring refactoring activities into automatable units, improving refactoring path recommendation and further reducing time-consuming and error-prone human intervention.

KEYWORDS: Technical debt, Quality of Services, Design Defects, Architecture Evolution, Web Service Refactoring, Search-based Software Engineering.

Chapter 1 **Introduction**

1.1 **Research Context**

1.1.1 **Service-oriented Computing**

The Service-Oriented Computing (SOC) is becoming the leading edge of modern software engineering and it is increasingly adopted in the software industry. Services are, in general, provided by third-parties who only expose services interfaces to the outer world. These services are commonly treated as "black-boxes" with abstract interfaces constituting the only visible part of the system. The interfaces are the main source of interactions with the user to adopt the services in real-world applications. Thus, poorly designed service interfaces may have a negative effect on all these applications using the services. A well-designed interface can accelerate project schedules and make the Service Oriented Architecture (SOA) solution more responsive to business needs. Indeed, service interfaces with well-defined abstractions and cohesive operations are easy to comprehend and reuse in business processes [1].

A Web Service is defined according to the W3C (World Wide Web Consortium), as "a software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artefacts. For SOAP web service, its interface is described as a WSDL (Web service Description Language) document that contains structured information about the Web service's location, its offered operations, the input/output parameters, etc." A Web service interface corresponds to a WSDL port type, which is the most important WSDL element. A Web service has at least one interface. This WSDL element describes a Web

service, the operations that can be performed, and the messages that are involved. It can be compared to a function library (or a module or a class) in a traditional programming language.

In the context of implementing SOA solutions, the structure of a service interface is critical. Like any other software systems, Web services need to be changed and updated frequently to add new functionalities in response to client needs [2]. For example, a hotel management Web service must, over time, offer new features, become more reliable and respond faster. However, these continuous changes may lead to increase the complexity of the service interfaces and even taking them away from their original design [2]. This may in turn introduce side effects known as antipatterns – symptoms of bad design and implementation practices that often lead to several usability, understandability and maintainability problems as well as runtime errors [3]. YouTube, eBay, Google, FedEx, PayPal, and many other companies are leveraging these Web Services in a reusable, distributed and portable fashion that can be invoked by the users [4]. SBSs evolve over time to meet new requirements or to fix bugs. Such continuous changes may have a negative impact on the quality of the services.

1.1.2 **Web Service Antipatterns**

A common bad design practice, i.e., antipattern, that often appear in real-world Web services is to group together a large number of semantically unrelated operations in a single interface [3], [5]. Most of existing interfaces tend to cover several distinct core abstractions and [3]processes, leading to many operations associated with each abstraction. This inappropriate service modularization will result in poorly designed applications that tend to be hard to use, implement, maintain and evolve [6]–[8].

Best practice for service design suggests that services should expose their operations in a modular way, where each module, i.e., interface, defines operations that handle one abstraction at

a time [6], [9]. Service interfaces will consequently exhibit low coupling and high cohesion [10]. Low coupling means that a service interface is independent to other interfaces, allowing an effective reuse. Cohesion refers to how strongly related the operations themselves are. High cohesion means that the service operations are related as they operate on the same, underlying core abstraction.

Service's providers always try to improve the quality of their service interface descriptions to ensure best practice of third-party reuse [3]. Although this observation might sound obvious, developers tend to take little care of their service WSDL descriptions as several researchers have pointed out [5] Search-based web service antipatterns detect [3], [11]. Most of these existing descriptions are designed in only one interface regrouping all the operations together.

Web service bad design practices and antipatterns have been recently studied, and different approaches have been proposed to discover Web service interfaces suffering from bad design practices [3], [11]–[13]. However, fixing these antipatterns is still unexplored and it is a manual, complex, time-consuming and error-prone task. Indeed, designing a service interface with the right number of interfaces, i.e., port types, and an appropriate assignment of operations to port types is a non-trivial task especially when developers are under pressure and stress to meet several release deadlines [5], [14], [15]. In fact, the number of operation combinations to explore is exponentially high, leading to a large and complex search space.

Unlike the area of object oriented design [16]–[25], there has been recently few studies focusing on the study of bad design practices for web services interface [3], [11], [13], [26]. The vast majority of these work relies on declarative rule specification. In these settings, rules are manually defined to identify the key symptoms that characterize an interface design defect using combinations of mainly quantitative metrics. For each possible interface design defect, rules that

are expressed in terms of metric combinations need high calibration efforts to find the right threshold value for each metric. Another important issue is that translating symptoms into rules is not obvious because there is no consensual symptom-based definition of design defects [13]. These difficulties explain a large portion of the high false-positive rates reported in existing research [11]. Recently, a heuristic-based approach based on genetic programming [3] is used to generate design defects detection. However, such approaches require a high number of interface design defect examples (data) to provide efficient detection rules solutions. In fact, design defects are not usually documented by developers. In addition, it is challenging to ensure the diversity of the examples to cover most of the possible bad-practices.

1.1.3 **Web Service Refactoring**

Software refactoring is defined by Fowler [27] as "the process of changing the internal structure of a software to improve its quality without altering the external behavior". Refactoring is recognized as an essential practice to improve software quality. Dudney et al. [10] have defined an initial catalog of refactoring operations for Web services including Interface Partitioning, Interface Consolidation, Bridging Schemas or Transforms and Web Service Business Delegate. Despite being commonly used in the Object-Oriented Programming (OOP) paradigm and widely supported by OOP integrated development environments (IDEs), refactoring is still unexplored in the context of service-oriented computing (SOC). In fact, SOC refactoring is not a trivial case of recoding existing OOP refactoring techniques.

Despite the extensive adoption of Web service technologies, very few studies has been proposed for the first step of the refactoring process which is the detection of antipatterns [26]. Indeed, the vast majority of existing work in Web services antipattern detection merely attempts to provide definitions and/or the key symptoms that characterize common antipatterns. Recent

works [11], [12] rely on a declarative rule-based language to specify antipattern symptoms at a higher-level of abstraction using combinations of quantitative (metrics), structural, and/or lexical information. However, in an exhaustive scenario, the number of possible antipatterns to be characterized manually and formulated with rules can be large. To make the situation worse, it is difficult to find a consensus to characterize and formulate such symptoms. For these reasons, the detection task is still mainly a manual, time-consuming and subjective process.

1.2 **Research Contributions**

Figure 1 summarizes the different contributions of this work, published in 11 venues (3 journals: IEEE Transaction on Services Computing [28], ACM Transactions on Internet Technology [29], and ACM Transactions on the Web [30], and 8 conferences: 3*ICWS2017 [31]–[33], 1*ICSOC2017 [34], 2*ICSOC2016 [35], [36], 1*ICWS2016 [37], 1*MEDES2015 [38]) related to the prediction, detection and correction of Web services design defects based on various intelligent computational search and machine learning techniques. In the following, we will summarize the aims of each contribution.

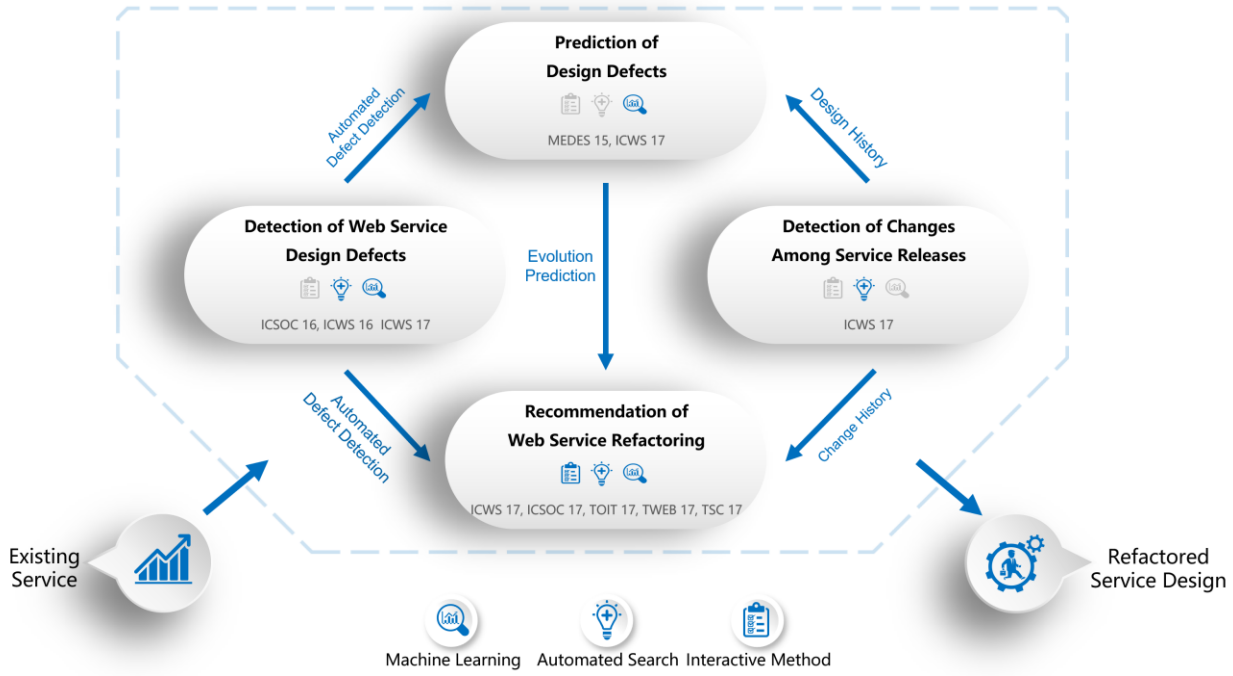


Figure 1 Overview of the proposed contributions

1.2.1 **Contribution 1: Detection of Web Service Design Defects**

Several quality metrics can be used to capture the structural and semantic attributes of the Web services, and can be a reliable indicator of the quality of design [12]. These quality indicators can then be used to quantitatively estimate and reflect the design signatures of Web Services architecture in terms of many metrics. The antipatterns detection process usually involves finding the fragments of the design which violate these metrics. In this contribution, we used a set of static and Web service and dynamic QoS metrics [39]. Static metrics aim at measuring the structural properties of Web services in both the interface (WSDL) and code levels, whereas QoS metrics aim at invoking the Web services and measuring different properties, e.g., response time.

Many metric combinations are possible, so the detection rules generation process is, by nature, a combinatorial optimization problem. The number of possible solutions quickly becomes huge as the number of metrics and possible threshold values increases. A deterministic search is not practical in such cases, and hence the use of heuristic search is warranted. The dimensions of

the solution space are set by the metrics, their threshold values, and logical operations between them, e.g., union (metric1 OR metric2) and intersection (metric1 AND metric2). A solution is determined by assigning a threshold value to each metric.

The manual definition of rules to identify maybe difficult and can be time-consuming. The main issue with Web service antipattern detection is that there is no general consensus on how to decide if a particular design violates a quality heuristic. Indeed, there is a difference between detecting symptoms and asserting that the detected situation is an actual antipattern. Deciding which Web services are antipattern candidates heavily depends on the interpretation of each analyst. In some contexts, an apparent violation of a design principle may be consensually accepted as normal practice. For example, a translation Web service1 may have in its interface only a single operation translates which is responsible for translating text from one language to another language. Although this service might be designed properly, from a strict antipattern definition, it may be considered as a fine-grained Web service.

Another inherent problem is related to the definition of threshold values when dealing with quantitative information. Indeed, there is no general agreement on extreme manifestations of Web service antipatterns [13]. That is, for each antipattern, rules that are expressed in terms of metrics need substantial calibration efforts to find the right threshold value for each metric, above which an antipattern is said to be detected.

To address or circumvent the above-mentioned issues and challenges, we used multi-objective heuristic-based and bi-level approaches and integrate QoS information to automatically detect Web service antipatterns.

1.2.2 **Contribution 2: Detection of Changes among Service Releases**

In contrast to change tracking approaches, state-based refactoring detection approaches aim to reveal refactorings some posteriori on the base of the two successively modified versions of a software artifact. The detection of atomic changes on program code has a long history in computer science as pointed out by [4], but is still an ongoing research topic [40]. In [11], [41], a very recent approach for detecting refactorings improving several open issues of previous approaches has been proposed. In particular, REF-FINDER tool is presented which is capable of detecting complex refactorings, which comprise a set of atomic refactorings using logic-based rules executed by a logic programming engine.

In this contribution, we propose a genetic algorithm approach [42] to detect composite changes between multiple Web service releases. Our approach takes as input an exhaustive list of possible change types, the initial release and the revised one, and generates as output a list of detected changes in terms of refactorings (composite changes). A solution is defined as the combination of refactoring operations that should maximize the structural and textual similarity between the expected new Web service interface release and the generated one after applying the refactoring sequence on the initial release. Due to the large number of possible solutions, a search-based method, based on Genetic Algorithms (GA) is used instead of an enumerative one to explore the space of possible solutions.

1.2.3 **Contribution 3: Prediction of Web Services Evolution**

Service-based systems heavily depend on the interface of selected services used to implement specific features. However, service providers do not know, in general, the impact of their changes, during the evolution Web services, on the applications of subscribers. The subscribers are reluctant, in general, to use Web services that are risky and not stable [2]. Thus,

analyzing and predicting Web service changes is critical but also challenging because of the distributed and dynamic nature of services.

We propose a machine learning approach based on Artificial Neural Networks (ANN) [43] to predict the evolution of Web services interface from the history of previous releases' metrics. The predicted interface metrics value are used to predict and estimate the risk and the quality of the studied Web services. We evaluated our approach on a set of 6 popular Web services including more than 90 releases. We reported the results on the efficiency and effectiveness of our approach to predict the evolution of Web services interfaces and provide useful recommendations for both service providers and subscribers. The results indicate that the prediction results of several Web service metrics, on the different releases of the 6 Web services, were similar to the expected ones with very low deviation rate. Furthermore, most of the quality issues of Web service interfaces were accurately predicted, for the next releases, with an average precision and recall higher than 82%. The survey conducted with a set of developers also shows the relevance of prediction technique for both service providers and subscribers.

1.2.4 **Contribution 4: Recommendation of Web Services Design Refactoring**

The structure of a service interface is critical in SOA. However, developers tend to take little care of their service WSDL descriptions as several researchers have pointed out [3], [5], [11], [41]. Most of these existing descriptions are designed in only one interface grouping all the operations together. To this end, Web service bad design practices and antipatterns have been recently studied, and different approaches found that several of existing Web service interfaces are suffering from bad design practices and proposed solutions to detect them [3], [11]–[13]. In this work, we propose three search-based approaches to contribute Web services refactoring research:

The first approach uses Genetic Algorithm (GA)-based interactive learning algorithm [44] for Web services interface modularization based on Artificial Neural Networks (ANN) [45]. The proposed approach is based on the important feedback of the user to guide the search for relevant Web services modularization solutions using predictive models. To the best of our knowledge, the use of predictive models has not been used to improve the quality of Web services design. In the proposed approach, we are modeling the user's design preferences using ANN as a predictive model to approximate the fitness function for the evaluation of the Web services modularization solutions. The user is asked to evaluate manually Web services interface modularization solutions suggested by a Genetic Algorithm (GA) for few iterations then these examples are used as a training set for the ANNs to evaluate the solutions of the GA in the next iterations.

The second approach is based on the PCA-NSGA-II methodology [46], aims at finding the best and reduced set of objective that represents the quality metrics of interest to the domain expert. A regular multi-objective NSGA-II algorithm [47] with an initial set of exhaustive metrics is executed for a number of iterations then a PCA component analyzes the correlation between the different objectives using the execution traces. The number of objectives maybe reduced during the next iterations based on the PCA results. The process is repeated several times until a maximum number of iterations is reached to generate a set of non-dominated Web services modularization solutions.

Finally, a recommendation approach is proposed that dynamically adapts and interactively suggests a possible modularization, also called refactoring [27], of the Web services interface to developers and takes their feedback into consideration. Our approach uses an interactive multi-criteria decision-making algorithm, based on interactive non-dominated sorting genetic algorithm (NSGA-II) [40], to find a set of good design interface modularization solutions that provide a

trade-off between (1) improving several interface design quality metrics (e.g. coupling, cohesion, number of portTypes and number of antipatterns), (2) maximizing the satisfaction of the interaction constraints learnt from the user feedback during the execution of the algorithm, while (3) minimizing the deviation from the initial design. To find a trade-off between these different conflicting objectives, there is no single possible modularization solution but a set of optimal, i.e., non-dominated, solutions, so-called Pareto front [40]. The challenge at this step is how to choose one solution from this front to present to the Web service's user or developer? The traditional approach is to seek a 'knee point' [40] from the front that presents the maximum trade-off between the different objectives. However, this may ignore the preferences of the user. To address this issue, we propose to analyze and explore the Pareto front of possible remodularization solutions interactively and implicitly with the developer.

1.3 **Roadmap**

The remainder of this thesis is structured as follows: Chapter 2 reviews the related work on software codes smells, software refactoring approaches, service design defects, detection of service changes, investigation of service evolution, and service refactoring. Chapter 3 introduces our contributions for detecting Web Service design defects. Chapter 4 reports our contribution related to the detection of changes among service releases. Chapter 5 reports our work on the prediction of software and services architecture evolution. Chapter 6 describes three contributions related to the recommendation of Web Service Refactoring. Finally, Chapter 7 presents the conclusions of this dissertation and outlines the future directions to expand our current work.

Chapter 2 **State of the Art**

2.1 **Introduction: Software and Web Service Design Defects**

2.1.1 **Software Code smells**

Code-smells, also called anti-patterns, anomalies, design flaws or bad smells, are problems resulting from bad design practices and refer to design situations that adversely affect the software maintenance. According to Fowler [27], bad-smells are unlikely to cause failures directly, but may do it indirectly. In general, they make a system difficult to change, which may in turn introduce bugs. Different types of code-smells, presenting a variety of symptoms, have been studied with the intent of facilitating their detection and suggesting improvement solutions. In [27], the authors define 22 sets of symptoms of code smells. These include large classes, feature envy, long parameter lists, and lazy classes. Each code-smell type is accompanied by refactoring suggestions to remove it. In this work, we focus on the following seven code-smell types to evaluate our approach:

- **Blob**: This is found in designs where one large class monopolizes the behavior of a system (or part of it), and the other classes primarily encapsulate data. It is a large class that declares many fields and methods with a low cohesion and has almost no parents and no children.
- **Data Class**: This is a class that contains only data and performs no processing on these data. It is typically composed of highly cohesive fields and accessors.

- **Spaghetti Code**: This is a code with a complex and tangled control structure. This code-smell is characteristic of procedural thinking in object-oriented programming. Spaghetti Code is revealed by classes with no structure, declaring long methods with no parameters, and utilizing global variables. Names of classes and methods may suggest procedural programming. Spaghetti Code does not exploit and prevents the use of object-orientation mechanisms, polymorphism, and inheritance.
- **Functional Decomposition**: This occurs when a class is designed with the intent of performing a single function. This is found in code produced by non-experienced object-oriented developers.
- **Schizophrenic class:** This occurs when a public interface of a class is large and used non-cohesively by client methods i.e., disjoint groups of client classes use disjoint fragments of the class interface in an exclusive fashion.
- **Shotgun Surgery:** This occurs when a method has a large number of external operations calling it, and these operations are spread over a significant number of different classes. As a result, the impact of a change in this method will be large and widespread.
- **Feature Envy:** This is found when a method heavily uses attributes and data from one or more external classes, directly or via accessor operations. Furthermore, in accessing external data, the method is intensively using data from at least one external capsule.

2.1.2 **Web Service Design Defects**

A SOAP or Restful Web service has at least one interface. For example, a typical SOAP Web service interface corresponds to a WSDL port type, which is the most important WSDL element. This WSDL element describes a Web service, the operations that can be performed, and

the messages that are involved. It can be compared to a function library (or a module or a class) in a traditional programming language.

Antipatterns are symptoms of poor design and implementation practices that describe bad solutions to recurring design problems. They often lead to software which is hard to maintain and evolve [48]. Different types of antipatterns presenting a variety of symptoms have been recently studied with the intent of improving their detection and suggesting improvements paths [10], [11], [13]. Web service interface antipatterns/defects are defined as bad design choices that can have a negative impact on the interface quality such as maintainability, changeability and comprehensibility which may impacts the usability and popularity of services [10], [13]. They can be also considered as structural characteristics of the interface that may indicate a design problem that makes the service hard to evolve and maintain, and trigger refactoring [4]. In fact, most of these defects can emerge during the evolution of a service and represent patterns or aspects of interface design that may cause problems in the further development of the service. In general, they make a service difficult to change, which may in turn introduce bugs. It is easier to interpret and evaluate the quality of the interface design by identifying different defects definition than the use of traditional quality metrics. To this end, recent studies defined different types of Web services design defects [4], [10], [13]. In our experiments, we focus on the eight following Web service defect types:

- **God object Web service (GOWS)**: implements a high number of operations related to different business and technical abstractions in a single service.
- **Fine grained Web service (FGWS)**: is a too fine-grained service whose overhead (communications, maintenance, and so on) outweighs its utility.

- **Chatty Web service (CWS)**: represents an antipattern where a high number of operations are required to complete one abstraction.
- **Data Web service (DWS)**: contains typically accessor operations, i.e., getters and setters. In a distributed environment, some Web services may only perform some simple information retrieval or data access operations.
- **Ambiguous Web service (AWS)**: is an antipattern where developers use ambiguous or meaningless names for denoting the main elements of interface elements (e.g., port types, operations, messages).
- **Redundant PortTypes (RPT)**: is an antipattern where multiple portTypes are duplicated with the similar set of operations.
- **CRUDy Interface (CI)**: is an antipattern where the design encourages services the RPC-like behavior by declaring create, read, update, and delete (CRUD) operations, e.g., *createX()*, *readY()*, etc.
- **Maybe It is Not RPC (MNR)**: is an antipattern where the Web service mainly provides CRUD-type operations for significant business entities.

We choose these defect types in our experiments because they are the most frequent and hard to detect [3], [11], [26], cover different maintainability factors, due to the availability of defect examples and to compare the performance of our detection technique to existing studies [3][11]. However, the proposed approach in this work is generic and can be applied to any type of defects. The defects detection process consists in finding interface design fragments that violate structural or semantic properties such as the ones related to coupling and complexity. In this setting, internal attributes used to define these properties, are captured through several metrics, and properties are expressed in terms of valid values for these metrics. The list of metrics is described in Table 1.

Table 1 List of quality metrics

Metric Name	Definition
NPT	Number of port types
NOD	Number of operations declared
NAOD	Number of accessor operations declared
NOPT	Average number of operations in port types
ANIPO	Average number of input parameters in operations
ANOPO	Average number of output parameters in operations
NOM	Number of messages
NBE	number of elements of the schemas
NCT	Number of complex types
NST	Number of primitive types
NBB	Number of bindings
NBS	Number of services
NPM	Number of parts per message
NIPT	Number of identical port types
NIOP	Number of identical operations
COH	Cohesion
COU	Coupling
AMTO	Average meaningful terms in operation names
AMTM	Average meaningful terms in message names
AMTMP	Average meaningful terms in message parts
AMTP	Average meaningful terms in port-type names
ALOS	Average length of operations signature
ALPS	Average length of port-types signature
ALMS	Average length of message signature

In the following, we introduce some issues and challenges related to the detection of the Web service defects. Overall, there is no consensus on how to decide if a design violates a quality heuristic. In fact, there is a difference between detecting symptoms and asserting that the detected situation is an actual design defect. Another issue is related to the definition of thresholds when dealing with quantitative information. For example, the *GOWS* defect detection involves information such as the interface size as illustrated in Figure 2. Although we can measure the size of an interface, an appropriate threshold value is not trivial to define. An interface considered large in each service/community of users could be considered average in another. The generation of detection rules requires a large defect example set to cover most of the possible bad-practice

behaviors. Defects are not usually documented by developers (unlike bugs report and object-oriented design). Thus, it is time-consuming and difficult to collect defects and inspect manually large Web services. In addition, it is challenging to ensure the diversity of the defect examples to cover most of the possible bad-practices then using these examples to generate good quality of detection rules.

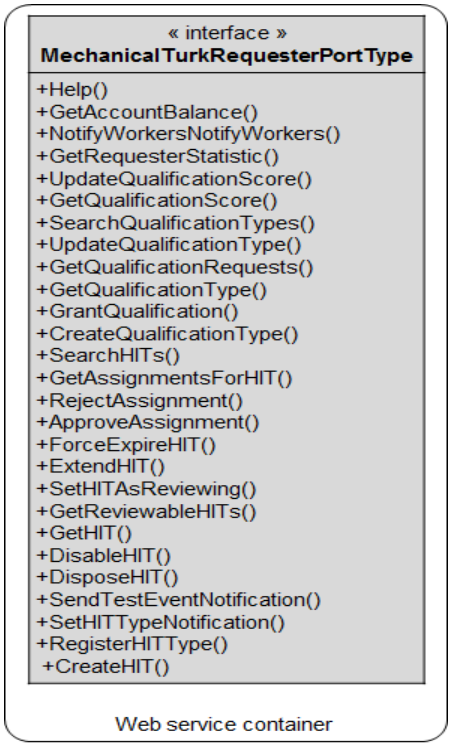


Figure 2 A god object Web service (GOWS) example

2.2 **Detection of Web Service Design Defects**

2.2.1 **Software Code Smell Detection**

The code-smell detection process consists in finding code fragments that violate structural or semantic properties such as the ones related to coupling and complexity. In this setting, internal attributes used to define these properties, are captured through software metrics, and properties are

expressed in terms of valid values for these metrics. This follows a long tradition of using software metrics to evaluate the quality of the design including the detection of code-smells. The most widely-used metrics are the ones defined by Chidamber and Kemerer [49]. In this work, we use variations of these metrics and adaptations of procedural ones as well including: Weighted Methods per Class (WMC), Response for a Class (RFC), Lack of Cohesion of Methods (LCOM), Number of Attributes (NA), Attribute Hiding Factor (AH), Method Hiding Factor (MH), Number of Lines of Code (NLC), Coupling Between Object classes (CBO), Number of Association (NAS), Number of Classes (NC), Depth of Inheritance Tree (DIT), Polymorphism Factor (PF), Attribute Inheritance Factor (AIF), and Number of Children (NOC). Kessentini et al. [22] allows detecting code-smells using metric-based detection rules independently to their severity, risk or importance and without predicting the evolution of the code smells. Detection rules are expressed in terms of metrics and threshold values. Each rule detects a specific defect type and is expressed as a logical combination of a set of quality metrics/threshold values. These detection rules are generated/learned from real instances of code-smells using genetic algorithm.

One of the well-known development activities that can help fix code-smells and reduce the increasing complexity of a software system is refactoring. Fowler [27] defines refactoring as a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. The idea is to reorganize variables, classes and methods to facilitate future adaptations and extensions. This reorganization is used to improve different aspects of software-quality such as maintainability, extensibility, reusability, etc. [50]. For these precious benefits on design quality, some modern integrated development environments (IDEs), such as Eclipse, NetBeans, and Refactoring Browser, provide semi-automatic support for applying the most commonly used refactorings, e.g., move method, rename class, etc. However,

automatically suggesting/deciding where and which refactorings to apply is still a real challenge in software engineering.

2.2.2 **Web Service Design Defect Detection**

Comparing to object-oriented code smell, Web service antipattern is a relatively new field. Few works have proposed to address the problem of SOA antipatterns [10] was the first book related to this topic in the literature, it provides informal definitions of a list of Web service antipatterns. Later, Rotem-Gal-Oz described the symptoms of a set of SOA antipatterns in [4]. Then, [13] describes seven popular antipatterns which violate the SOA principles [42], [51], [52] provide a set of guidelines for service providers to avoid bad practices while writing WSDL documentations and able to locate eight bad practices in writing WSDL for Web services. Beside the definition and guidelines of service antipattern, there are also several studies related to the detection part. Moha et al. proposed SODA [53], a rule-based approach for SCA systems (Service Component Architecture) in . Then, [11] extended this work for service antipatterns in SODA-W. Similar to DECOR [54], the proposed approach rely on a declarative rule specification using a DSL to identify the key antipattern symptoms of antipatterns. In another work [55], authors created and reviewed a repository of 45 general antipatterns in SOA, and aim to help developers understand and avoid potential problems. [56] has proposed an approach to prevent antipattern during the phase of WSDL documentation generation. Recently, several search-based approaches have been proposed. [12] uses genetic programming to generate detection rules based on the interface metrics of service antipattern examples, and later extend in [3] which use cooperative parallel evolutionary algorithms and brings the code-level metrics into the process.

The first limitation of exsiting work is lack of alternative solutions based on conflict objectives(e.g. generalirity and correctness) and users' preference. Second, the detection rules are

generated based on the exsiting services, the antipattern examples are limited to study. It is difficult to find the best detection rule without generating artificial defects due to the limited training set. Another limitation of the state-of-the-art approaches is the limited use of dynamic QoS metrics to measure service quality, more specifically, only response time is being used in [3]. However, in the dynamic environment of Web services, the QoS performance is critical to both sides of the service provider and user. The information extracted which is only based on code-level or structural-level is not enough to understand the full characteristics of Web services. Therefore, the Web service antipattern detection approaches in the literature are not able to provide a comprehensive service antipattern detection framework.

2.3 **Detection of Changes among Service Releases**

Fokaefs *et al.* [57] used the *VTracker* tool to calculate the minimum edit distance between two trees representing two WSDL files. The outcome of the tool is the percentage of interface changes such as added, changed and removed elements among the XML models of two WSDL interfaces. Romano *et al.* [2] proposed a similar tool called *WSDLDiff* that can identify fewer types of change than *VTracker* that may help to analyze the evolution of a WSDL interface without manually inspecting the XML changes. Aversano *et al*. [58] analyzed the relationships between sets of services change during the service evolution based on formal concept analysis. The focus of the study is to extract relationships among services.

Several studies have been proposed to measure the similarity between different Web services to search for relevant ones or classify them but not to analyze their evolution. Xing *et al.* [59] suggested a tool, called *UMLDiff* to detect differences between different UML diagram versions to understand their evolution. Zarras et *al.* [60] detected evolution patterns and regularities

by adapting Lehman's laws of software evolution. The study was focused only on Amazon Web Services (AWS).

Based on this overview of existing work in the area of Web services evolution, the problem of predicting the evolution of Web services was not addressed before. In addition, the use of machine learning algorithms in Web services was limited to the classification of Web Services and their messages into ontologies [61]. These existing machine learning-based studies are not concerned with the analysis of the releases within the same Web service but more about mining different Web services (one release per service) to classify them in order to help the composition of services process for the subscribers based on their requirements.

2.4 **Recommendation of Software and Web Service Refactoring**

2.4.1 **Search-based Software Refactoring Recommendation**

Software refactoring is defined by Fowler [27] as "the process of changing the internal structure of a software to improve its quality without altering the external behavior". Refactoring is recognized as an essential practice to improve software quality. Dudney et al. [10] have defined an initial catalog of refactoring operations for Web services including Interface Partitioning, Interface Consolidation, Bridging Schemas or Transforms and Web Service Business Delegate. Despite being commonly used in the Object-Oriented Programming (OOP) paradigm and widely supported by OOP integrated development environments (IDEs), refactoring is still unexplored in the context of service-oriented computing (SOC). In fact, SOC refactoring is not a trivial case of recoding existing OOP refactoring techniques.

Several studies are proposed in the literature to address the refactoring problem. We focus mainly in this related work on existing search-based refactoring work. These studies are based on the use of mono, multi and many-objective optimization techniques. The GA was the most used

metaheuristic search algorithm according to a recent survey [62] and recently there has been also many other algorithms such as NSGA-II [40] and NSGA-III [46]. Hence, we classify those approaches into two main categories: (1) *mono-objective approaches*, and (2) *multi/many-objective ones*.

In the first category, the majority of existing work combines several metrics in a single fitness function to find the best sequence of refactorings. In [63], Qayum et. al. considered the problem of refactoring scheduling as a graph transformation problem. They expressed refactorings as a search for an optimal path, using Ant colony optimization, in the graph where nodes and edges represent respectively refactoring candidates and dependencies between them. Recently, Kessentini et. al. [22] have proposed a single-objective combinatorial optimization using genetic algorithms to find the best sequence of refactoring operations that improve the quality of the code by minimizing as much as possible the number of design defects detected on the source code. Kilic et. al. [64] explore the use of a variety of population-based approaches to search-based parallel refactoring, finding that local beam search could find the best solutions.

In the second category of work, Harman et. al. [65] have proposed a search-based approach using Pareto optimality that combines two quality metrics, CBO (coupling between objects) and SDMPC (standard deviation of methods per class), in two separate fitness functions. The authors start from the assumption that good design quality results from good distribution of features (methods) among classes. Their Pareto optimality-based algorithm succeeded in finding good sequence of move method refactorings that should provide the best compromise between CBO and SDMPC to improve code quality. Ó Cinnéide et. al. [66] have proposed a multi-objective search-based refactoring to conduct an empirical investigation to assess some structural metrics and to explore relationships between them. To this end, they have used a variety of search techniques

(Pareto-optimal search, semi-random search) guided by a set of cohesion metrics. Furthermore, Ouni et al. [67] have proposed a new multi-objective refactoring to find the best compromise between quality improvement and semantic coherence using two heuristics related to the vocabulary similarity and structural coupling.

2.4.2 **Web Service Refactoring**

One of the first attempts to address service interface refactoring was by Athanasopoulos et al. [5] (Greedy). The approach was able to improve the cohesion of the Web service interface. However, the approaches limitation is not being able to perfectly adjust to the developers' needs [5]. The reasons could be ignoring the coupling between interfaces which results in much cohesive but highly connected interfaces, and the greedy algorithm provides only one specific remodularization solution with fixed interface size which might be not suitable for the developers. In another study, Mateos et al. [42] and Rodriguez et al. [51], [52] have proposed a set of guidelines for the providers of Web service to avoid introducing antipattern while constructing the WSDL files. Based on the heuristics, the authors detected eight bad practices happens while writing the WSDLs.

A lot of efforts has been devoted to refactoring of object-oriented (OO) applications. My contributions to this domain contain refactorings that are similar in OO systems, such as *Extract Class* which employs metrics to split a large class into smaller, more cohesive [27]. Bavota et al. [51], [52] have proposed an approach for software refactoring to split a large class into smaller cohesive classes using structural and semantic similarity measures. Fokaefs et al. [68] proposed an automated refactoring approach to *Extract Class* based on a hierarchical clustering algorithm to locate cohesive subsets of class methods and attributes. However, the *Extract Class* refactoring is not applicable in the context of Web services, because of the development paradigm, used

technologies and metrics are different. For example, Web service source code is not publicly available typically. The Web service providers only expose their interfaces to the clines, compare to general software refactoring, the main challenge is that less information can be extracted or gathered; this is also a key reason that we introduce client application releases.

2.5 **Conclusion**

In this Chapter, several related works are described, as well as their limitations. To summarize, Table 2 reports the limitations of the existing work. These limitations are addressed by my research works, and detailly reported in the following chapters.

Table 2 State of the art summary

<table>
<tr><th>Existing Work</th><th>Main Limitations</th><th>Problem Domain</th></tr>
<tr><td>Rotem-Gal-Oz et al. [4]</td><td rowspan="2">• Manual support to detect antipatterns
• Limited to three types of antipatterns
• Limited set of quality metrics to describe the symptoms</td><td rowspan="3">Detection Service Antipatterns</td></tr>
<tr><td>Rodriguez et al. [41]</td></tr>
<tr><td>Ouni et al. [3]</td><td rowspan="2">• Limited to interface-level metrics
• Aggregating several conflicting quality metrics and objectives
• Limited training set and types of antipatterns and metrics</td></tr>
<tr><td>Palma et al. [11]</td><td></td></tr>
<tr><td>Fokaefs et al. [68]</td><td rowspan="3">• Limited to the detection of atomic changes (not complex refactorings)
• Based only on structural similarities
• Limited types of changes to detect</td><td rowspan="3">Detection of Service changes</td></tr>
<tr><td>Aversano et al. [58]</td></tr>
<tr><td>Xing et al. [59]</td></tr>
<tr><td>Athanasopoulos et al. [5]</td><td rowspan="2">• Used only cohesion as a metric to evaluate the refactoring solutions
• Used only one type of refactoring to decompose portTypes.
• Generate only one decomposition solution and do not support the consideration user preferences.</td><td rowspan="2">Web Service Refactoring</td></tr>
<tr><td>Ouni et al. [69]</td></tr>
</table>

Chapter 3 **Detection of Web Service Design Defects**

3.1 **Multi-objective Web Service Design Defect Detection**

3.1.1 **Introduction**

Web services must be carefully designed and implemented to adequately fit in the required system's design whilst achieving good quality of services [13]. Indeed, there is no exact recipe to follow for proper service design. A set of guiding quality principles for service-oriented design exists, including such principles as service flexibility, operability, composability, and loose coupling. However, the design of services is strongly influenced by the context, environment and other decisions the service designers take, and such factors may lead to violations of quality principles. The presence of programming patterns associated with bad design and programming practices, known as *antipatterns*, are indications of such violations [70]. Furthermore, it is widely believed that such antipatterns lead to various maintenance and evolution problems including an increased bug rate, fragile design and inflexible code.

Despite the extensive adoption of Web service technologies, very few studies has been proposed for the first step of the refactoring process which is the detection of antipatterns [26]. Indeed, the vast majority of existing work in Web services antipattern detection merely attempts to provide definitions and/or the key symptoms that characterize common antipatterns. Recent works [11], [12] rely on a declarative rule-based language to specify antipattern symptoms at a higher-level of abstraction using combinations of quantitative (metrics), structural, and/or lexical

information. However, in an exhaustive scenario, the number of possible antipatterns to be characterized manually and formulated with rules can be large. To make the situation worse, it is difficult to find a consensus to characterize and formulate such symptoms. For these reasons, the detection task is still mainly a manual, time-consuming and subjective process.

To address the above-mentioned limitations, we propose in this work a multi-objective search-based approach for the generation of antipatterns detection rules from both bad and well-designed service examples. The process aims at finding the optimal combination of quality metrics, from an exhaustive list of possible metric combinations, that: 1) *maximizes the coverage of a set of antipattern examples collected from different systems*; and 2) *minimizes the detection of examples of good-design practices*. In fact, it is difficult to ensure that the used design defect examples cover all possible bad-design practices. Thus, we used good-design practices as another objective to detect antipatterns that are not similar to the well-designed service examples and design defect examples. To this end, a multi-objective genetic programming (MOGP) [71] is used to generate the antipatterns detection rules that find trade-offs between the two above-mentioned objectives. MOGP is a powerful evolutionary metaheuristic which extends the generic model of learning to the space of programs [71].

To validate our proposal, we present an empirical evaluation of our approach on a benchmark of 415 Web services from ten different application domains and we considered 8 common Web service antipattern types. We compared our multi-objective approach with random search, an existing mono-objective technique [3], [12] and a rule-based approach [11] not based on heuristic search techniques. Statistical analysis demonstrates the efficiency of our approach in detecting Web service antipatterns, on average, with a precision score of 94% and a recall score of

92%. To the best of our knowledge, this is the first work to use multi-objective evolutionary algorithms for the detection of Web service antipatterns.

3.2.1 Multi-Objective Genetic Programming

Genetic Programming (GP) is a powerful evolutionary metaheuristic which extends the generic model of learning to the space of programs [71]. Differently to other evolutionary approaches, in GP, population individuals are themselves programs following a tree-like structure instead of fixed length linear string formed from a limited alphabet of symbols. GP can be seen as a process of program induction that allows automatically generating programs that solve a given task. Most exiting work on GP makes use of a single objective formulation of the optimization problem to solve using only one fitness function to evaluate the solution. Differently to single-objective optimization problems, the resolution of Multi-Objective Optimization Problems (MOPs) yields a set of trade-off solutions called non-dominated solutions and their image in the objective space is called the Pareto front.

1: Create an initial population P_0
2: Create an offspring population Q_0
3: $t = 0$
4: **while** stopping criteria not reached **do**
5: $R_t = P_t \cup Q_t$
6: F = fast-non-dominated-sort(R_t)
7: $P_{t+1} = \emptyset \; and \; i = 1$
8: **while** $\mid P_{t+1} \mid + \mid F_i \mid \leqslant N$ **do**
9: Apply crowding-distance-assignment(F_i)
10: $P_{t+1} = P_{t+1} \cup F_i$
11: $i = i + 1$
12: **end while**
13: $Sort(F_i, \prec n)$
14: $P_{t+1} = P_{t+1} \cup F_i[N- \mid P_{t+1} \mid]$
15: Q_{t+1} = create-new-pop(P_{t+1})
16: t = t+1
17: **end while**

Figure 3 High level pseudo code for MOGP

A high-level view of MOGP is depicted in Figure 3. The algorithm starts by randomly creating an initial population P_o of individuals encoded using a specific representation (line 1). Then, a child

population Q_o is generated from the population of parents P_o (line 2) using genetic operators (crossover and mutation). Both populations are merged into an initial population R_o of size N (line 5). Fast non-dominated-sort [40] is the technique used by MOGP to classify individual solutions into different dominance levels (line 6). Indeed, the concept of non-dominance consists of comparing each solution x with every other solution in the population until it is dominated (or not) by one of them. According to Pareto optimality: "A solution x_1 is said to dominate another solution x_2, if x_1 is no worse than x_2 in all objectives and x_1 is strictly better than x_2 in at least one objective". Formally, if we consider a set of objectives $f_i, i \in 1..n$, to maximize, a solution x_1 dominates x_2 :

$$if\ \forall i,\ f_i(x_2) \leqslant f_i(x_1) \text{ and } \exists j \mid f_j(x_2) < f_j(x_1)$$

The whole population that contains N individuals (solutions) is sorted using the dominance principle into several fronts (line 6). Solutions on the first Pareto-front F_0 get assigned dominance level of 0 Then, after taking these solutions out, fast-non-dominated-sort calculates the Pareto front F_1 of the remaining population; solutions on this second front get assigned dominance level of 1, and so on. The dominance level becomes the basis of selection of individual solutions for the next generation. Fronts are added successively until the parent population P_{t+1} is filled with N solutions (line 8). When NSGA-II must cut off a front F_i and select a subset of individual solutions with the same dominance level, it relies on the crowding distance to make the selection (line 9). This parameter is used to promote diversity within the population. This front F_i to be split, is sorted in descending order (line 13), and the first $(N - |P_{t+1}|)$ elements of F_i are chosen (line 14). Then a new population Q_{t+1} is created using selection, crossover and mutation (line 15). This process will be repeated until reaching the last iteration according to stop criteria (line 4).

3.1.2 **Multi-Objective Optimization and NSGA-II**

An optimization problem consists in searching for an optimal or near-optimal solution within a predefined search space where the goal is to maximize or minimize a quality function called objective function. As opposed to single-objective optimization problems where we are looking for a single optimal solution, the resolution of a multi-objective problem (MOP) yields a set of compromise solutions, called non-dominated solutions, and their image in the objective space is called the Pareto front. In what follows, we give some background definitions related to this topic:

Definition - MOP. An MOP consists in minimizing or maximizing a set of objective functions under some constraints [40]. An MOP could be expressed as:

$$\begin{cases} Min\, f(x) = [f_1(x), f_2(x),..., f_M(x)]^T \\ g_j(x) \geq 0 & j = 1,...,P; \\ h_k(x) = 0 & k = 1,...,Q; \\ x_i^L \leq x_i \leq x_i^U & i = 1,...,n. \end{cases}$$

where M is the number of objective functions, P is the number of inequality constraints, Q is the number of equality constraints, x_i^L and x_i^U correspond to the lower and upper bounds of the variable x_i. A solution x_i satisfying the (P+Q) constraints is said to be feasible and the set of all feasible solutions defines the feasible search space denoted by Ω. In this formulation, we consider a minimization MOP since maximization can be easily turned to minimization based on the duality principle by multiplying each objective function by -1. The resolution of a MOP consists in *approximating the whole Pareto front*.

Definition - Pareto optimality. A solution $x^* \in \Omega$ is Pareto optimal if there does not exist any solution *x such that* $f_m(x) < f_m(x^*)$ *for all m.*

The definition of Pareto optimality states that x^* is Pareto optimal if no feasible vector x exists which would improve some objective without causing a simultaneous worsening in at least another one. Other important definitions associated with Pareto optimality are essentially the following:

Definition - Pareto dominance. A solution $u = (u_1,u_2,..., u_n)$ is said to dominate another solution $v = (v_1,v_2,...,v_n)$ (denoted by $f(u) \preceq f(v)$) if and only if $f(u)$ is partially less than $f(v)$. In other words, $\forall m \in \{1,..., M\}$ we have $f_m(u) \leq f_m(v)$ and $\exists m \in \{1,..., M\}$ where $f_m(u) < f_m(v)$.

Definition - Pareto optimal set. For a MOP $f(x)$, the Pareto optimal set is $P^* = \{x \in \Omega | \neg \exists x' \in \Omega, f(x') \preceq f(x)\}$.

Definition - Pareto optimal front. For a given MOP $f(x)$ and its Pareto optimal set P^*, the Pareto front is $PF^* = \{f(x), x \in P^*\}$.

Several methods were proposed in the literature to solve MOPs. Due to their population-based nature, Evolutionary Algorithms (EAs) have shown their effectiveness and efficiency in providing a well-converged and well-diversified approximation of the Pareto front independently of its geometrical nature which is not the case for classical mathematical methods. Among the most used Multi-Objective EAs (MOEAs), we cite NSGA-II, SPEA2, IBEA and MOEA/D. Since the most used MOEA within the SBSE community is NSGA-II [40], we choose to use it in this study.

NSGA-II is one of the most used and effective MOEAs. It begins by generating an offspring population from a parent one by means of variation operators (crossover and mutation) such that both populations have the same size. After that, it ranks the merged population (parents and children) into several non-dominated layers, called fronts, as depicted by Figure 3. Non-

dominated solutions are assigned a rank of 1 and constitute the first layer. Non-dominated solutions according to the population truncated of the layer 1 are assigned a rank of 2 and constitute the layer 2. This process is continued until the ranking of all parent and children individuals. After that, each solution is assigned a diversity score, called crowding distance, front wise. This distance corresponds to the half of the perimeter of the cuboid having the two closest neighboring solutions to the considered individual as vertices. It is important to note that extreme solutions are assigned an infinite crowding score since they are of great importance for diversity. The fitness in NSGA-II is not a scalar value. In fact, it is a couple (*rank*, *crowding distance*). Solutions having better ranks are emphasized. Among solutions having the same rank (belonging to the same layer), solutions having larger crowding distances are emphasized since they are less crowded than the others. Once all individuals of the merged population are assigned a rank and a diversity score, we perform the environmental selection to form the parent population for the next generation. Indeed, solutions belonging to the best layers are selected. Figure 4 illustrates this process where the last selected layer is the 4^{th} one. Usually, the cardinality of the last layer (layer 4 in Figure 4) is greater than the number of available slots in the parent population of the next generation. As denoted by Figure 4, solutions of the 4^{th} layer are selected based on their crowding distance values. In this way, most crowded solutions are discouraged to remain in the race; thereby emphasizing population diversification. To sum up, the Pareto ranking encourages convergence and the crowding factor procedure emphasizes diversity, therefore NSGA-II is an elitist multi-objective EA which is today the most used metaheuristic in multi-objective applied optimization.

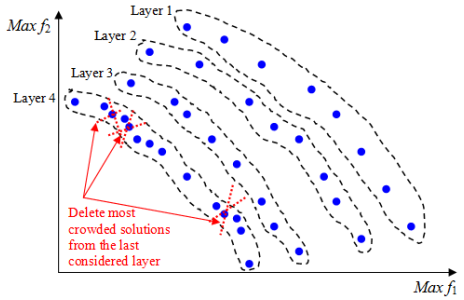


Figure 4 NSGA-II replacement scheme for a bi-objective maximization case.

One of the widely used multi-objective search techniques is NSGA-II [40] that has shown good performance in solving several software engineering problems [62].

As in Figure 5, the algorithm starts by randomly creating an initial population P_0 of individuals encoded using a specific representation (line 1). Then, a child population Q_0 is generated from the population of parents P_0 (line 2) using genetic operators (crossover and mutation). Both populations are merged into an initial population R_0 of size N (line 5). *Fast-non-dominated-sort* [20] is the technique used by NSGA-II to classify individual solutions into different dominance levels (line 6). Indeed, the concept of non-dominance consists of comparing each solution x with every other solution in the population until it is dominated (or not) by one of them. According to Pareto optimality: "A solution x_1 is said to dominate another solution x_2, if x_1 is no worse than x_2 in all objectives and x_1 is strictly better than x_2 in at least one objective". Formally, if we consider a set of objectives f_i , $i \in 1..n$, to maximize, a solution x_1 dominates x_2.

The whole population that contains N individuals (solutions) is sorted using the dominance principle into several fronts (line 6). Solutions on the first Pareto-front F_0 get assigned dominance level of 0 Then, after taking these solutions out, *fast-non-dominated-sort* calculates the Pareto-front F_1 of the remaining population; solutions on this second front get assigned dominance level of 1, and so on. The dominance level becomes the basis of selection of individual solutions for the

next generation. Fronts are added successively until the parent population P_{t+1} is filled with N solutions (line 8). When NSGA-II has to cut off a front F_i and select a subset of individual solutions with the same dominance level, it relies on the crowding distance [20] to make the selection (line 9). This parameter is used to promote diversity within the population. This front F_i to be split, is sorted in descending order (line 13), and the first (N- $|P_{t+1}|$) elements of F_i are chosen (line 14). Then a new population Q_{t+1} is created using selection, crossover and mutation (line 15). This process will be repeated until reaching the last iteration according to stop criteria (line 4).

```
1: Create an initial population P_0
2: Create an offspring population Q_0
3: t = 0
4: while stopping criteria not reached do
5:    R_t = P_t ∪ Q_t
6:    F = fast-non-dominated-sort(R_t)
7:    P_{t+1} = ∅ and i = 1
8:    while | P_{t+1} | + | F_i | ⩽ N do
9:       Apply crowding-distance-assignment(F_i)
10:      P_{t+1} = P_{t+1} ∪ F_i
11:      i = i + 1
12:   end while
13:   Sort(F_i, ≺ n)
14:   P_{t+1} = P_{t+1} ∪ F_i[N− | P_{t+1} |]
15:   Q_{t+1} = create-new-pop(P_{t+1})
16:   t = t+1
17: end while
```

Figure 5 High level pseudo code for NSGA-II

3.1.3 **NSGA-II Adaptation**

1) Problem Formulation

The Web service antipatterns detection problem involves searching for the best metric combinations among the set of candidate ones, which constitutes a huge search space. A solution of our antipatterns detection problem is a set of rules (metric combination with their thresholds values) where the goal of applying these rules is to detect design defects in a web service. We propose a multi-objective formulation of the Web service antipatterns rules generation problem. Consequently, we have two objective functions to be optimized: (1) *maximizing the coverage of*

antipattern examples, and (2) *minimizing the detection of good design practice examples of Web services*. The collected examples of well-designed Web services and antipatterns are taken as an input for our approach. Analytically speaking, the formulation of the multi-objective problem can be stated as follows:

$$\begin{cases} \max\ f_1(x) = \dfrac{\dfrac{|DCS(x)| \cap |ECS|}{|ECS|} + \dfrac{|DCS(x)| \cap |ECS|}{|DCS(x)|}}{2} \\ \min\ f_2(x) = \dfrac{\dfrac{|DCS(x)| \cap |EGE|}{EGE} + \dfrac{|DCS(x)| \cap |EGE|}{|DCS(x)|}}{2} \end{cases}$$

where $|DCS(x)|$ is the cardinality of the set of detected antipatterns by the metric combination x, $|ECS|$ is the cardinality of the set of existing antipatterns, and $|EDE|$ is the cardinality of the set of existing good examples. Once the bi-objective trade-off front is obtained, the developer can navigate through this front in order to select his/her preferred solution (metric combination).

The basic idea of the algorithm is to explore the search space by making a population of candidate solutions, also called individuals, and evolve this population towards an "optimal" solution for the detection of antipatterns. To evaluate the solutions, the fitness functions, as explained previously, are used. The best solutions (detection rules) will cover the maximum of anti-pattern examples and a minimum of good design examples of Web services.

In the initialization of the MOGP algorithm, our base of examples is split into ten subsets, each representing a different application domain, e.g., finance, travel, etc. One subset (WS) is the test dataset and the remaining subsets (B or GE) are the training datasets (the ground truth). Thus, MOGP is run to detect antipatterns in the selected subset(WS), which is not of course part of the training set.

The initial population for MOGP is a set of individuals (I) that stand for possible solutions representing detection rules (metrics combination). Then, the algorithm explores the search space

and constructs new individuals by combining metrics to generate rules. In each iteration of the training process, antipatterns are iteratively evaluated using the generated rules. As described earlier, the process is driven by two fitness functions that calculates the quality of each candidate solution (detection rule) by comparing the base of examples along with the percentage of covered well-designed examples. A new population of individuals is generated by iteratively selecting pairs of parent individuals from population Pop and applying genetic operators to them (crossover and mutation). We include both the parent and child variants in the new population. We then apply the mutation operator, with a probability score, for both parent and child to ensure solution diversity; this produces the population for the next generation. Developers can use the best rules (solution) to detect potential antipatterns on any new Web service.

2) Solution Approach

In the following, we describe the three main steps of adaptation of the MOGP algorithm to our problem.

Solution representation:

Candidate solutions to the problem are antipattern detection rules. A solution is represented as a set of IF-THEN rules, each with the following structure:

IF "*Combination of metrics with their thresholds*" **THEN** "*antipattern type*"

The antecedent of the IF statement combines some metrics and their threshold values using logic operators (AND, OR). If these conditions are satisfied by a Web service, then it is determined to be of the antipattern type featuring in the THEN clause of the rule. Figure 6 provides an example. More formally, each candidate solution S is a sequence of detection rules where each rule is represented by a binary tree such that:

A. Each leaf node (terminal) L represents a metric (our metric suite described earlier) and its corresponding threshold, generated randomly.

B. Each internal node (function) N represents a logic operator, either AND or OR.

We will have as many rules as types of antipatterns to be detected. In this work, we focus on the detection of eight common types as defined in Section II-A.

Evaluation functions:

The solution is evaluated based on the two objective functions defined in the previous section. Since we are considering a bi-objective formulation, we use the concept of Pareto optimality to find a set of compromise (Pareto-optimal) solutions. The fitness of a particular solution in MOGP corresponds to a couple (Pareto Rank, Crowding distance). In fact, MOGP classifies the population individuals (of parents and children) into different layers, called nondominated fronts. The output of MOGP is the last obtained parent population containing the best of the non-dominated solutions found. When plotted in the objective space, they form the Pareto front from which the user will select his/her preferred antipatterns detection rules solution.

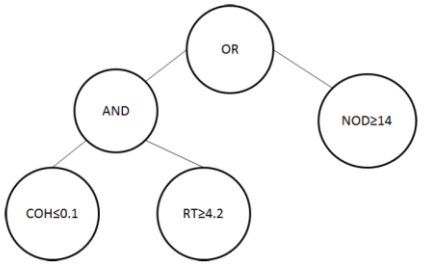


Figure 6 Solution representation example.

3.1.4 **Validation**

1) Experimental Setup

We designed our experiments to answer the following research questions:

- RQ1: How does our multi-objective approach, MOGP, compare to random search and an existing mono-objective technique [3]?
- RQ2: To what extent can the proposed approach efficiently detect Web service antipatterns?

- RQ3: What types of Web service antipatterns does it detect correctly?
- RQ4: How does MOGP perform compared to existing Web service antipattern detection approach not based on heuristic search [11]?

To evaluate our approach, we collected a set of Web services using different Web service search engines including eil.cs.txstate.edu/ServiceXplorer, programmableweb.com, biocatalogue.org, webservices.seekda.com, taverna.org.uk, and myexperiment.org. Table 3 summarizes the collected services. Furthermore, our collected Web services are drawn from ten different application domains: financial, science, search, shipping, travel, weather, media, education, messaging and location. All services were manually inspected and validated to identify antipatterns based on guidelines from the literature [4], [10]. Furthermore, our dataset is available online [72] to encourage future research in the area of automated detection of Web service antipatterns. We considered antipattern types range from eight common antipatterns, namely god object web service (GOWS), fine-grained Web service (FGWS), chatty Web service (CWS), data Web service (DWS), ambiguous Web service (AWS), redundant port types (RPT), CRUDy interface (CI), and maybe it is not RPC (MNR). In our study, we employed a 10-fold cross validation procedure. We split our data into training data and evaluation data. For each fold, one category of services is evaluated by using the remaining nine categories as a base of examples (ground-truth). For instance, weather services are analyzed using antipattern instances from travel, shipping, search, science financial, media, education, messaging, and location services. We use precision and recall [73] to evaluate the accuracy of our approach. Precision denotes the ratio of true antipatterns detected to the total number of detected antipatterns, while recall indicates the ratio of true antipatterns detected to the total number of existing antipatterns. To answer RQ1, we investigate and report on the effectiveness of MOGP, since one of our primary novelties lies in the

adoption of the multi-objective formulation. To this end, we implemented random search (RS) with the same fitness functions as MOGP. Indeed, it is important to compare our search technique to random search, since if an intelligent search method fails to outperform random search, then the proposed formulation is not adequate. In addition, we compared our multi-objective algorithm to an existing mono-objective approach where only examples of antipatterns were considered [3] without the use of positive examples of well-designed Web services. To answer RQ2, we use both recall and precision to evaluate the efficiency of our approach in identifying antipatterns. To answer RQ3, we investigated the antipattern types that were detected to find out whether there is a bias towards the detection of specific antipattern types. To answer RQ4, we compared our approach with the SODA-W approach of Palma et al. [11]. SODA-W manually translates antipattern symptoms into detection rules and algorithms based on a literature review of Web service design. All three approaches are tested on the same benchmark described in Table 3.

Table 3 Web services used in the empirical study.

Category	# services	# antipatterns	average NOD	average NOM	average NCT
Financial	94	67	29.52	57.31	19.01
Science	34	3	8.47	17.14	96.73
Search	37	13	8.35	18.94	26.13
Shipping	38	10	13.36	27.76	20.21
Travel	65	28	16.09	33.13	121.13
Weather	42	15	8.54	17.16	9.14
Media	19	14	10.9	16.4	28.6
Education	26	15	11.3	15.36	32.46
Messaging	29	20	7.6	11.21	18.25
Location	31	22	5.8	28.32	11.15
All	425	136	17.08	34.2	48.6

2) Experiment Results

Results for RQ1. The goal of RQ1 is to investigate how well MOGP performs against random search and an existing single-objective approach where only antipattern examples are used. Table 4 and Figure 7 report the comparative results. Over 31 runs, RS did not perform well

when compared to MOGP in terms of precision and recall achieving average values of only 29% and 31% respectively on the different Web services. The main reason could be related to the large search-space of possible combinations of metrics and threshold values to explore.

Table 4 MOGP results on the different Web service.

Category	Precision (%)	Recall (%)
Financial	94	92
Science	96	98
Search	94	95
Shipping	96	90
Travel	94	96
Weather	91	96
Media	96	91
Education	97	93
Messaging	92	98
Location	94	91
Average	**89**	**93**

The results achieved by MOGP are also better than the mono-objective approach in terms of precision and recall. In fact, the single-objective GP technique has an average of 86% and 87% of precision and recall however MOGP has better scores with 94% of precision and 92% of recall on the different Web services. These results confirm that an intelligent search is required to explore the search space and that the use of well-designed We service examples improved the obtained detection results.

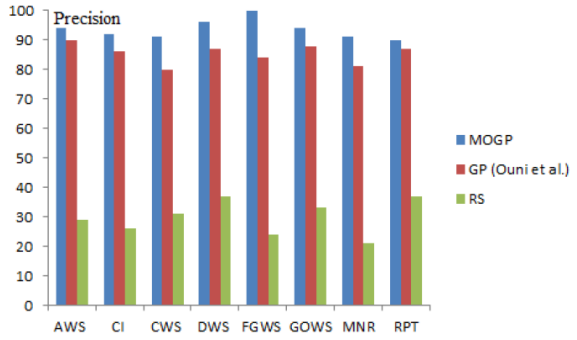
Results for RQ2. The results for RQ2 are presented in Table 4 MOGP results on the different Web service. The obtained results show that we were able to detect most of the expected antipatterns in the different categories with a median precision higher than 94%. The higher precision value for travel and Education (97%) can be explained by the fact that these Web services are large than the others and contain a high number of operations and complex types that match the GOWS antipattern. For the We service weather, the precision is the lowest one (91%), but is still a very acceptable score. This is due to the nature of the antipatterns involved which are typically data or chatty Web services. Indeed, some false positives are recorded for the DWS and

CWS antipatterns. These antipatterns are likely to be difficult to detect using metrics alone. Similar interpretations can be made for recall. The obtained results indicate that our approach is able to achieve a recall of 92%. The highest values were recorded for travel services with 96% where most of the detected services are GOWS and AWS. The lowest recall score was recorded for the location service (91%) which is attributable mostly to FGWS. Indeed, location Web services typically provide one or two operations which falsely matches the symptoms of FGWS.

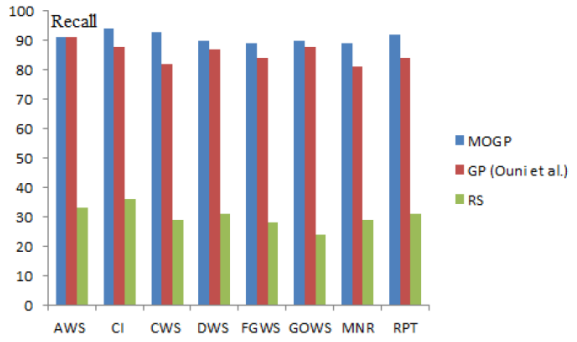
Results for RQ3. Based on the results of Figure 7, we observe that MOGP does not have a bias towards the detection of any specific antipattern type. As described the figure, we had an almost equal distribution of each antipattern type. On some Web services such as weather, the distribution is not as balanced. This is principally due to the number of actual antipattern types detected. Overall, all the 8 antipattern types are detected with good precision and recall scores (more than 88%). Most existing guidelines/definitions [10], [11] rely heavily on the notion of size to detect antipatterns. This is reasonable for antipatterns like GOWS and FGWS that are associated with a notion of size, but for antipatterns like AWS, however, the notion of size is less important, and this makes this type of anomaly hard to detect using structural information. This difficulty limits the performance of GP in detecting this type of antipattern. Thus, we can conclude that our MOGP approach detects well all the types of considered antipatterns (RQ3).

Results for RQ4. Figure 8 reports the comparison result of MOGP, Ouni et al. [3], [12], and SODA-W. While SODA-W shows promising results with an average precision of 71% and recall of 83%, it is still less than MOGP in all the eight considered antipattern types. We conjecture that a key problem with SODA-W is that it simplifies the different notions/symptoms that are useful for the detection of certain antipatterns. Indeed, SODA-W is limited to a set of WSDL interface metrics, but ignores the source code of the Web service artifacts. In fact, service design

may look promising at the interface level, but can prove to be an antipattern if the source code is not implemented well. In contrast, our approach is based on both interface and code metrics. Another limitation of SODA-W is that in an exhaustive scenario, the number of possible antipatterns to manually characterize with rules can be very large, and rules that are expressed in terms of metric combinations need substantial calibration efforts to find the suitable threshold value for each metric. By contrast, our approach needs only some examples of antipatterns to generate detection rules. Figure 8 also shows that the mono-objective GP [12] provides lower detection results for the eight studied antipatterns with an average of 72% for both precision and recall. The lower performance can be explained by the fact that of the mono-objective formulation is based only on interface metrics that may not be able to capture all possible antipattern symptoms.

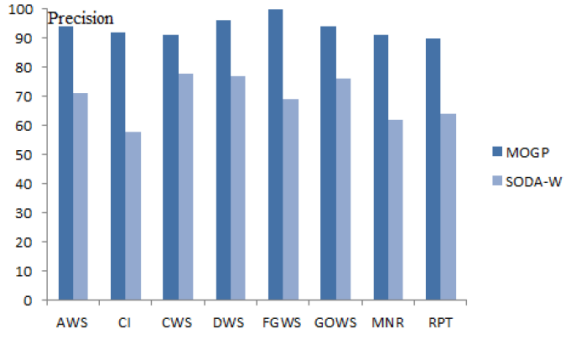


(a) Precision



(b) Recall

Figure 7 Detection results for each antipattern type



(a) Precision

(b) Recall

Figure 8 Comparative results of MOGP, Mono-objective GP and SODA-W

In this work, we introduced a new multi-objective approach for the detection of Web service antipatterns. In our multi-objective adaptation, two fitness functions are used to maximize the coverage of antipattern examples and minimize the coverage of well-designed Web service examples. The proposed approach is evaluated on a benchmark of 415 Web services and eight common Web service antipattern types. Statistical analysis of the obtained results provides compelling evidence that the proposed multi-objective algorithm outperforms mono-objective approaches, random search, and a recent state-of-the art approach with a median precision of more than 94% and a median recall of more than 92%. As future work, we plan to extend the approach to detect business process antipatterns in SBS in addition to individual Web service antipatterns and automate the correction, through refactoring, of the detected antipatterns.

3.1.5 **Conclusion**

In this contribution, we introduced a new multi-objective approach for the detection of Web Service antipatterns. In our multi-objective adaptation, two fitness functions are used to maximize the coverage of antipattern examples and minimize the coverage of well-designed Web service examples. The proposed approach is evaluated on a benchmark of 415 Web services and eight common Web service antipattern types. Statistical analysis of the obtained results provides compelling evidence that the proposed multi-objective algorithm outperforms mono-objective approaches, random search, and a recent state-of-the art approach with a median precision of more than 94% and a median recall of more than 92%.

In the next section, we extend the approach to Bi-level which is able to generate artificial design defects to improve the generation process of antipatterns detection rules.

3.2 **Bi-level Identification of Web Service Defects**

3.2.1 **Introduction**

In the majority of existing works, detection rules are manually defined to identify the key symptoms that characterize an interface design defect using combinations of mainly quantitative metrics. For each possible interface design defect, rules that are expressed in terms of metric combinations need high calibration efforts to find the right threshold value for each metric. Another important issue is that translating symptoms into rules is not obvious because there is no consensual symptom-based definition of design defects [74]. These difficulties explain a large portion of the high false-positive rates reported in existing research [11]. Recently, a heuristic-based approach based on genetic programming [3] is used to generate design defects detection. However, such approaches require a high number of interface design defect examples (data) to provide efficient detection rules solutions. In fact, design defects are not usually documented by developers. In addition, it is challenging to ensure the diversity of the examples to cover most of the possible bad-practices.

In this work, we start from the hypothesis that the generation of efficient Web service defects detection rules heavily depends on the coverage and the diversity of the used defect examples. In fact, both mechanisms for the generation of detection rules and the generation of defect examples are dependent. Thus, the intuition behind this work is to generate examples of defects that cannot be detected by some possible detection solutions then adapting these rules-based solutions to be able to detect the generated defect examples. These two steps are repeated until reaching a termination criterion (e.g. number of iterations). To this end, we propose, for the first time, to consider the Web services defects detection problem as a bi-level one [75], [76]. Bi-Level Optimization Problems (BLOPs) are a class of challenging optimization problems, which

contain two levels of optimization tasks. The optimal solutions to the lower level problem become possible feasible candidates to the upper level problem.

In our adaptation, the upper level generates a set of detection rules, combination of quality metrics, which maximizes the coverage of the base of defect examples; and artificial defects are generated by the lower level. The lower level maximizes the number of generated "artificial" interface defects that cannot be detected by the rules produced by the upper level. The overall problem appears as a BLOP task, where for each generated detection rule, the upper level observes how the lower-level acts by generating artificial Web service interface defects that cannot be detected by the upper level rule, and then chooses the best detection rule which suits it the most, taking the actions of the defects generation process (lower level or follower) into account. The main advantage of our bi-level formulation is that the generation of detection rules is not limited to some interface defect examples identified manually that are difficult to collect but it allows the prediction of new interface defect behaviours that are different from those in the base of examples.

The primary contributions of this work can be summarized as follows:

A. The work introduces a novel formulation of the Web services design defects detection as a *bi-level* problem.
B. The work reports the results of an empirical study with an implementation of our bi-level approach. The statistical analysis of our experiments over 30 runs on a benchmark of 415 Web services shows that 8 types of interface design defects were detected with an average of more than 93% of precision and 98% recall. The results confirm the outperformance of our bi-level proposal compared to state-of-art Web service design defects detection techniques [11], [12] and the survey performed by potential users and programmers also shows the relevance of detected defects.

3.2.2 **Bi-Level Optimization**

Most studied real-world and academic optimization problems involve a single level of optimization. However, in practice, several problems are naturally described in two levels. These latter are called BLOPs [75], [76]. In such problems, we find a nested optimization problem within the constraints of the outer optimization one. The outer optimization task is usually referred as the *upper level problem* or the *leader problem*. The nested inner optimization task is referred as the *lower level problem* or the *follower problem*, thereby referring the bi-level problem as a leader-follower problem or as a Stackelberg game. The follower problem appears as a constraint to the upper level, such that only an optimal solution to the follower optimization problem is a possible feasible candidate to the leader one.

BLOPs are intrinsically more difficult to solve than single-level problems, it is not surprising that most of existing studies to date has tackled the simplest cases of BLOPs, i.e., problems having nice properties such as linear, quadratic or convex objective and/or constraint functions. In particular, the most studied instance of BLOPs has been for a long time is the linear case in which all objective functions and constraints are linear with respect to the decision variables.

3.2.3 **Bi-level Approach Overview**

As described in Figure 9, our bi-level formulation includes two levels as described in the previous section. At the upper level, the detection rules generation process has a main objective which is the generation of detection rules that can cover as much as possible the Web service defects in the base of examples. The defects generation process has one objective that is maximizing the number of generated artificial defects that cannot be detected by the detection rules. The generated defects are dissimilar from the base of well-designed Web services design

based on a defined distance using the different metrics. There is a hierarchy in the problem, which arises from the manner in which the two entities operate. The detection rules generation process has higher control of the situation and decides which detection rules for the defects generation process to operate in. It should be noted that in spite of different objectives appearing in the problem, it is not possible to handle such a problem as a simple multi-objective optimization task. The reason for this is that the leader cannot evaluate any of its own strategies without knowing the strategy of the follower, which it obtains only by solving a nested optimization problem.

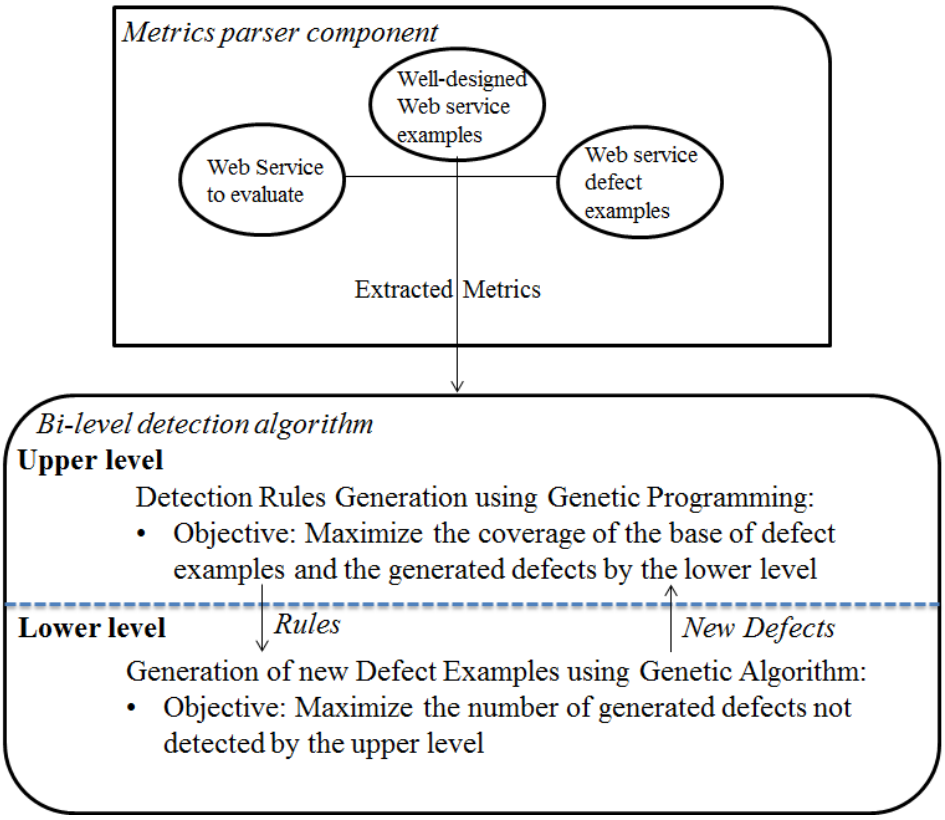


Figure 9 Bi-level Web service defects detection overview

The leader (upper level) takes as inputs a base (i.e. a set) of Web service defect examples, and takes, as controlling parameters, a set of metrics as described in Table 1 and generates as output a set of detection rules. The rule generation process selects randomly, from the list of possible metrics, a combination of quality metrics (and their threshold values) to detect a specific

defect types. Consequently, the ideal solution is a set of rules that best detect the defects of the base of examples and those generated by the lower level. For example, the following rule of Figure 10 states that a Web service *s* satisfying the following combination of metrics and thresholds is considered as a *GOWS* defect:

R1: **IF** (*NOD(s)≥17* AND *COH(s)≤0.43)* OR *NCT≥32,* **THEN** s = *GOWS.*

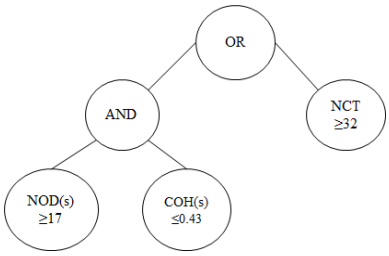


Figure 10 Solution representation at the upper level

An upper-level detection rules solution is evaluated based on the coverage of the base of defect examples (input) and also the coverage of generated "artificial" Web service design defects by the lower-level problem. These two measures are used to be maximized by the population of detection rules solutions. The follower (lower level) uses a set of well-designed Web service examples to generate "artificial" defects based on the notion of deviation from a reference (well-designed) set of Web services. The generation process of artificial defect examples is performed using a heuristic search that maximizes on one hand, the distance between generated defects examples and reference code examples using the list of considered metrics and, on the other hand, maximizes the number of generated examples that are not detected by the leader (detection rules). As described in Figure 11, the generated structure of defects is represented as a vector where each element is a (metric, threshold) pair that characterises the generated Web service.

NOD= 34	NCT= 64	NPT=104

Figure 11 Solution representation at the lower level.

There is no parallelism in our bi-level formulation. The upper level is executed for number iterations then the lower level for another number of iterations. After that the best solution found in the lower level will be used by the upper level to evaluate the associated solution (detection rules), and then this process in repeated several times until reaching a termination criterion (e.g. number of iterations). Thus, there is no parallelism since both levels are dependent.

Next, we describe our adaptation of bi-level optimization to the Web service defects detection problem in more details.

3.2.4 **Bi-Level Optimization Adaptation**

At the upper level, the objective function is formulated to maximize the coverage of Web services defect examples (input) and also maximize the coverage of the generated artificial Web service defects at the lower level (best solution found in the lower level). Thus, the objective function at the upper level is defined as follows:

$$Maximize\ f_{upper} = \frac{\frac{\Pr ecision(SR, WSDefectExamples) + \mathrm{Re}\, call(SR, WSDefectExamples)}{2} + \frac{\#\det ectedArtificialWSDefects}{\#artificialWSDefects}}{2}$$

It is clear that the evaluation of solutions (detection rules) at the upper level depends on the best solutions generated by the lower level (artificial Web service defects). Thus, the fitness function of solutions at the upper level is calculated after the execution of the optimization algorithm in the lower level at each iteration.

At the lower level, for each solution (detection rule) of the upper level an optimization algorithm is executed to generate the best set of artificial Web service defects that cannot be detected by the detection rules at the upper level. An objective function is formulated at the lower level to maximize the number of un-detected artificial defects that are generated and also maximize the distance with well-designed Web services. Formally,

$$Maximize \quad f_{lower} = u + Min\left(\sum_{j=1}^{ms}\left|M_j(ArtificialDefect) - M_j(\text{Re}\, ferenceExamples)\right|\right)$$

where *ms* is the number of structural metrics used to compare between artificial defects and the well-designed web services, *M* is a structural metric (such as the number of operations, etc.) and *u* is the number of artificial defects uncovered by the detection rule solution defined at the upper level.

For the GP algorithm (upper-level), the mutation operator can be applied to a function node (metric), or to a terminal node (logical operator) in our tree representation. It starts by randomly selecting a node in the tree. Then, if the selected node is a terminal (metric), it is replaced by another terminal (metric or another threshold value); if it is a function (AND-OR), it is replaced by a new function; and if tree mutation is to be carried out, the node and its sub-tree are replaced by a new randomly generated sub-tree. For the GA (lower-level), the mutation operator consists of randomly changing a metric in one of the vector dimension.

Regarding the crossover, two parent individuals are selected at the upper level, and a sub-tree is picked on each one. Then crossover swaps the nodes and their relative sub-trees from one parent to the other. This operator must ensure the respect of the depth limits. The crossover operator can be applied with only parents having the same rule category (defect type to detect). Each child, thus combines information from both parents. For the GA (lower-level), the crossover operator allows to create two offspring o_1 and o_2 from the two selected parents p_1 and p_2, where the first k elements of p_1 become the first k elements of o_1. Similarly, the first k elements of p_2 become the first k elements of o_2.

3.2.5 **Validation**

1) Experimental Setup

In order to evaluate the feasibility and the performance of our bi-level (BLOP) approach comparing to existing Web service defects detection approaches, we addressed the following research questions:

- RQ1: How does BLOP perform to detect different types of Web service defects? The goal of this research question is to quantitatively assess the completeness and correctness of our approach.
- RQ2: How do BLOP perform compared to existing mono-level Web service defects detection algorithms? The goal is to evaluate the benefits of the use of a bi-level approach in the context of Web service defects detection.
- RQ3: How does BLOP perform compared to the existing Web service defects detection approaches not based on the use of metaheuristic search?
- RQ4: Can our approach be useful for developers during the development of software systems?

To evaluate the performance of our approach, we used an existing benchmark [11], [12] that includes a set of Web services from different categories as described in Table 5.

Table 5 Web services used in the empirical study

Category	#services	#defects
Financial	94	67
Science	34	3
Search	37	13
Shipping	38	10
Travel	65	28
Weather	42	15
Media	19	14
Education	26	20
Messaging	29	22
Location	31	136

We considered the different antipattern types described in chapter 2. We used a 10-fold cross validation procedure. We split our data into training data and evaluation data. For each fold, one category of services is evaluated by using the remaining nine categories as training examples. We use the two measures of precision and recall evaluating the accuracy of our approach and to compare it with existing techniques [11], [12]. Precision denotes the ratio of true antipatterns detected to the total number of detected antipatterns, while recall indicates the ratio of true antipatterns detected to the total number of existing antipatterns.

To answer RQ1, we use both recall and precision to evaluate the efficiency of our approach in identifying antipatterns. We also investigated the Web service defect types that were detected to find out whether there is a bias towards the detection of specific Web service defect types.

To answer RQ2, we investigate and report on the effectiveness of BLOP comparing to existing approaches. We implemented random search (RS) with the same used fitness functions used at the two levels. If an intelligent search method fails to outperform random search, then the proposed formulation is not adequate. In addition, we compared our bi-level algorithm to an existing mono-level and mono-objective approach where only examples of defects were considered [11] without the use of the lower level.

To answer RQ3, we compared our approach with the SODA-W approach of Palma et al. [11]. SODA-W manually translates Web services defect symptoms into detection rules based on a literature review of Web service design. All three approaches are tested on the same benchmark described in Table 7.

To answer RQ4, we used a post-study questionnaire that collects the opinions of developers on our detection tool and Web service defects. To this end, we asked 31 software developers, including 17 professional developers working on the development of services-based application

and 14 graduate students from the University of Michigan. The experience of these subjects on web development and Web services ranged from 2 to 16 years. All the graduate students have an industrial experience of at least 2 years with large-scale systems especially in automotive industry.

2) Parameters Tuning

We performed a set of experiments using several population sizes: 30, 40 and 50. The stopping criterion was set to 500,000 fitness evaluations. We used a high number of evaluations as a stopping criterion since our bi-level approach requires involves two levels of optimization. Each algorithm was executed 30 times with each configuration and then comparison between the configurations was performed based on precision and recall using the Wilcoxon test with a 95% confidence level ($\alpha = 5\%$). The other parameters setting were fixed by trial and error and are as follows: (1) crossover probability = 0.6; mutation probability = 0.4 where the probability of gene modification is 0.2. Both lower-level and upper-level are run each with a population of 40 individuals and 50 generations.

3) Experiment Results

The results for the first research question RQ1 are presented in Table 6. The obtained results show that we were able to detect most of the expected antipatterns in the different categories with a median precision higher than 96%. The highest precision value for *Science* (100%) can be explained by the fact that these Web services contain the lowest number of Web service defects. For the Web service *Location*, the precision is the lowest one (89%), but is still an acceptable score. It could be explained by the nature of the antipatterns involved which are typically data or chatty Web services. These antipatterns are likely to be difficult to detect using metrics alone. Similar observations are valid for the recall. The obtained results indicate that our approach is able to achieve an average recall of more 93%. The highest values (after the Science category) were

recorded for *Location* services with 98% where most of the expected defects are detected but with the lowest precision. The lowest recall score was achieved the *Financial* services (92%). Indeed, these Web services contain the highest number of expected defects to be detected. Figure 12 and

Figure 13 confirm that our detection rules can detect different types of Web service defects with almost similar scores of precision and recall. Thus, the quality of the detection rules are good for almost all the defect types considered in our experiements. Overall, all the 8 antipattern types are detected with good precision and recall scores (more than 89%). This could be explained by the diverse set of generated defects by the lower level leading to a better coverage of possible defects to detect. This ability to identify different types of Web service defects underlines a key strength to our approach. Most other existing detection techniques rely heavily on the notion of size to detect defects. This is reasonable considering that some Web service defects like the *GOWS* are associated with the notion of size. For defects like *AWS*, however, the notion of size is less important, and this makes this type of defect hard to detect using structural information. Thus, we can conclude that our BLOP approach detects well all the types of considered antipatterns (RQ1).

The goal of research questions RQ2 and RQ3 is to investigate how well BLOP performs against random search (RS), an existing mono-level and single-objective approach (GP) where only defect examples are used (without the consideration of the lower-level algorithm), and an existing detection tool (SODA-W) not based on computational search. Figure 12 and

Figure 13 report the average comparative results. Over 30 runs, RS did not perform well when compared to BLOP both in terms of precision and recall achieving average around 30% on the different Web services. The main reason could be related to the large search-space of possible combinations of metrics and threshold values to explore, and the diverse set of Web service defects to detect. Furthermore, the results achieved by BLOP are also better than the mono-objective

approach in terms of precision and recall. In fact, the single-objective GP technique has an average of 86% and 87% of precision and recall however BLOP has better scores with an average of more than 93% of precision and recall on most of the different Web services. These results confirm that an intelligent search is required to explore the search space and that the use of the two levels improved the obtained detection results.

Table 6 Median precision and recall results based on 30 runs

Category	Precision	Recall
Financial	96	92
Science	100	100
Search	97	94
Shipping	98	96
Travel	94	96
Weather	93	97
Media	98	94
Education	96	96
Messaging	94	97
Location	89	98

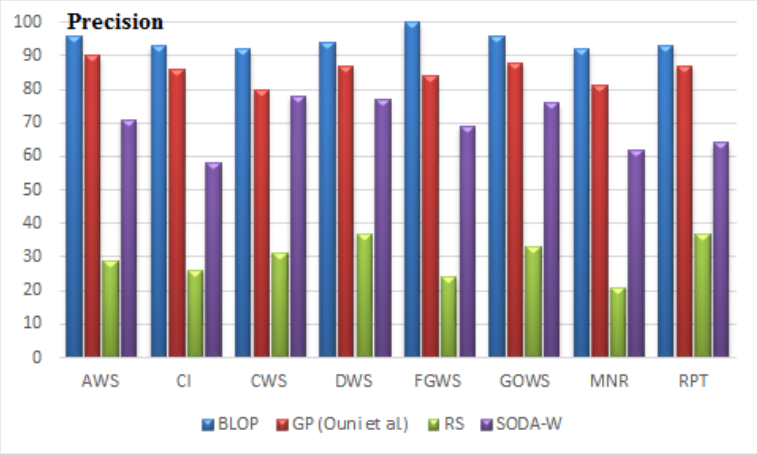


Figure 12 Median precision value over 30 runs on all the 10 Web service categories using the different detection techniques with a 95% confidence level ($\alpha < 5\%$)

While SODA-W shows promising results with an average precision of 71% and recall of 83% (Figure 12 and

Figure 13), it is still less than BLOP in all the eight considered defect types. We conjecture that a key problem with SODA-W is that it simplifies the different notions/symptoms that are useful for the detection of certain antipatterns. Indeed, SODA-W is limited to a smaller set of WSDL interface metrics comparing to our approach. In an exhaustive scenario, the number of possible antipatterns to manually characterize with rules can be large, and rules that are expressed in terms of metric combinations need substantial calibration efforts to find the suitable threshold value for each metric. However, our approach needs only some examples of defects to generate detection rules.

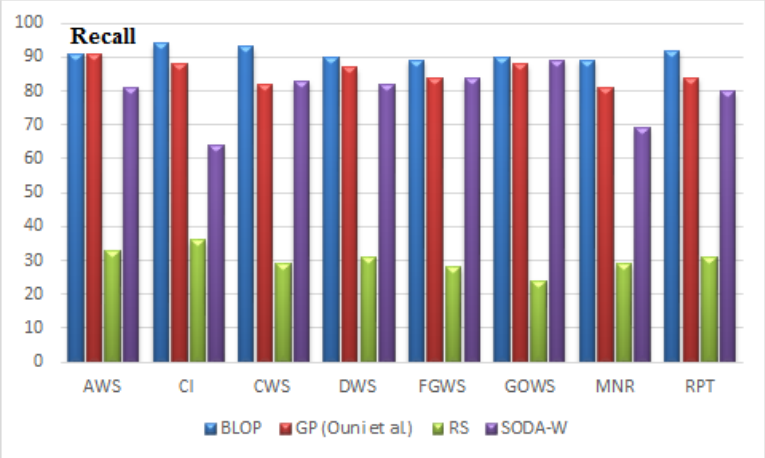


Figure 13 Median recall value over 30 runs on all the 10 Web service categories using the different detection techniques with a 95% confidence level ($\alpha < 5\%$)

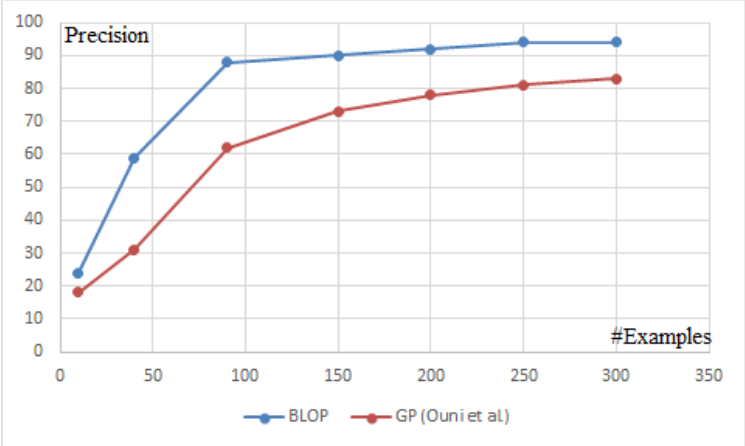


Figure 14 The impact of the number of Web service defect examples on the quality of the results (Precision on the *Financial* Web services).

One of the advantages of using our BLOP adaptation is that the developers do not need to provide a large set of examples to generate the detection rules. In fact, the lower-level optimization can generate examples of Web service defects that are used to evaluate the detection rules at the upper level. Figure 14 shows that BLOP requires a low number of manually identified defects to provide good detection rules with reasonable precision scores. The existing mono-level work of Ouni et al. [3] (GP) require a higher number of defect examples than BLOP to generate good quality of detection rules. We can conclude, based on the obtained results that our BLOP approach outperforms, in average, an existing mono-level search technique [3] and an approach not based on heuristic search [26] (response to RQ2 and RQ3).

To answer RQ4, subjects were first asked to fill out a pre-study questionnaire containing five questions. The questionnaire helped to collect background information such as their role within the company, their programming experience, their familiarity with Web services and web-based applications. The first part of the questionnaire includes questions to evaluate the relevance of some detected Web service defects using the following scale: 1. Not at all relevant; 2. Slightly relevant; 3. Moderately relevant; and 4. Extremely relevant. If a detected Web service defect is

considered relevant then this is mean that the developer considers that it is important to fix it. The second part of the questionnaire includes questions for those defects that are considered at least "moderately relevant", we asked the subjects to specify their usefulness based on the following list: 1. Refactoring guidance; 2. Quality assurance; 3. Bug prediction; 4. Web service stability; and 4. Web service selection. During the entire process, subjects were encouraged to think aloud and to share their opinions, issues, detailed explanations and ideas with the organizers of the study and not only answering the questions.

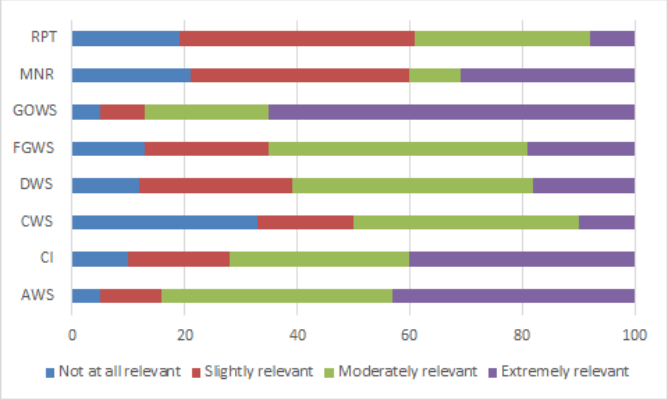


Figure 15 The relevance of detected Web service defects evaluated by the subjects

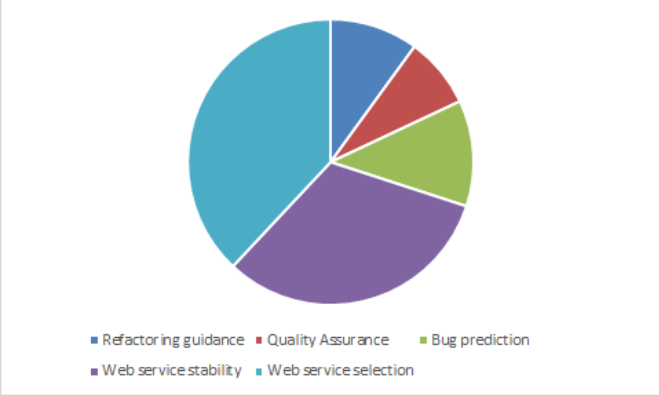


Figure 16 The usefulness of detected Web service defects evaluated by the subjects

Figure 15 illustrates that only less than 16% of detected Web service defects are considered not at all relevant by the developers. Around 67% of the defects are considered as moderately or

extremely relevant by the developers. This confirms the importance of the detected Web service defects for developers that they need to fix them for a better quality of their systems. It is also important to evaluate the usefulness of the detected Web service defects for the users. Figure 16 shows that the main usefulness is related to the Web services selection. In fact, most of the developers of service-based systems that we interviewed found that the detected defects give relevant advices about which service to select when several options are available. The users prefer, in general, to select services that are stable and have lower risk to include quality issues or bugs. However, we believe that we cannot generalize the results of our survey due to the limited number of participants.

3.2.6 **Conclusion**

In this work, we have proposed a bi-level evolutionary optimization approach for the problem of Web service defects detection. The upper-level optimization produces a set of detection rules, which are combinations of quality metrics, with the goal to maximize the coverage of not only a defect examples base but also a lower-level population of artificial defects. The lower-level optimization tries to generate artificial Web service defects that cannot be detected by the upper-level detection rules, thereby emphasizing the generation of broad-based and fitter rules. The statistical analysis of the obtained results over an existing benchmark have shown the competitiveness and the outperformance of our proposal in terms of precision and recall over a single-level genetic programming [3] and a non-search-based approach [11].

The next chapter will extend our detection methods by considering dynamic QoS metrics of Web services.

3.3 **On the Use of Quality of Service for Detecting Bad Design Practices**

3.3.1 **Introduction**

Quality of service (QoS) is a combination of several service qualities or properties, such as response time and availability. QoS has long been a major concern in areas, such as real-time application [77], middle-ware [78], [79], and networking [80], [81]. Organizations in modern markets, such as e-commerce activities, require QoS management [82]. With the right control of QoS, the quality service product can fulfill client expectations and achieve customer satisfactions. In this work, nine common QoS metrics are used to identify refactoring opportunities.

The common idea of previously mentioned or developed techniques is to generate detection rules, mainly based on the interface or code-level metrics of the bad-designed Web services. These metrics are extracted from the interface or the code skeleton of the Web service. Though these techniques, developers can evaluate the design quality based on the very limited static information exposed by the Web service provider.

Beside design quality of Web services, another important concept is called quality of service(QoS). It refers to the non-functional aspects of Web service. The non-functional attributes of QoS, e.g., response time, availability and reliability, have become the major concerns in the management of Web services [83], [84]. The service clients also compare QoS measurements to select from the Web services with similar functionalities. An acceptable QoS of the service is considered same importance as desired functional results [85].

In the real world, developers usually seek for the Web services that have not only a well-designed structure, but also an outstanding overall QoS performance. Such services can achieve non-functional requirements with less effort to implement or maintain. However, to the best of our knowledge, all the existing antipattern detection techniques of the Web service do not take into

consideration the QoS metrics. Without the real-time quality measurements of Web service, the detection of best refactoring opportunities could be hard to find. Furthermore, every service has its own business concept and purpose, so their exposed interface could be very different. The refactoring opportunity detection that only validates the static information could be not accurate. This issue can be addressed by introducing the dynamic QoS metrics.

In this work, we propose an antipattern detection approach based on the QoS measurement and the structural information of Web service. This work is an extension of our previous publication in the Proceedings of the 23rd International Conference on Web Services as follows.

We propose a novel approach for detecting Web service refactoring opportunities based on the dynamic QoS and the static interface/code metrics. In our approach, multi-objective algorithm NSGA-II [40] is implemented to generated the best detection rule sets that maximize the detection of Web service antipattern examples and minimize the detection of well-designed Web service design examples.

We extend our initial approach which is based on the static input of the web service examples by introducing dynamic QoS metrics. We introduce 9 QoS metrics namely, response time, availability, throughput, successability, reliability, compliance, best practices, latency, documentation. Manual inspection and survey of our previous work show, some detected antipatterns from the services with a high QoS performance do not cause difficulties and don't consider as antipattern to the users. These Web services are still preferred to use by the developers. On the other hand, some refactoring opportunities that can't be generalized perfectly by the current metric types since there are no considerations of the non-functional behaviors.

We perform an empirical study of our approach on 500 Web services from a QoS benchmark. We evaluated how well our approach can detect refactoring opportunities with the

state-of-the-art techniques [12], [26], [35], [37] that uses static structural metrics. Statistical analysis demonstrates the accuracy of our approach in service refactoring opportunities detection, with a precision score of 91% and a recall score of 85%.

3.3.2 **Motivating Example**

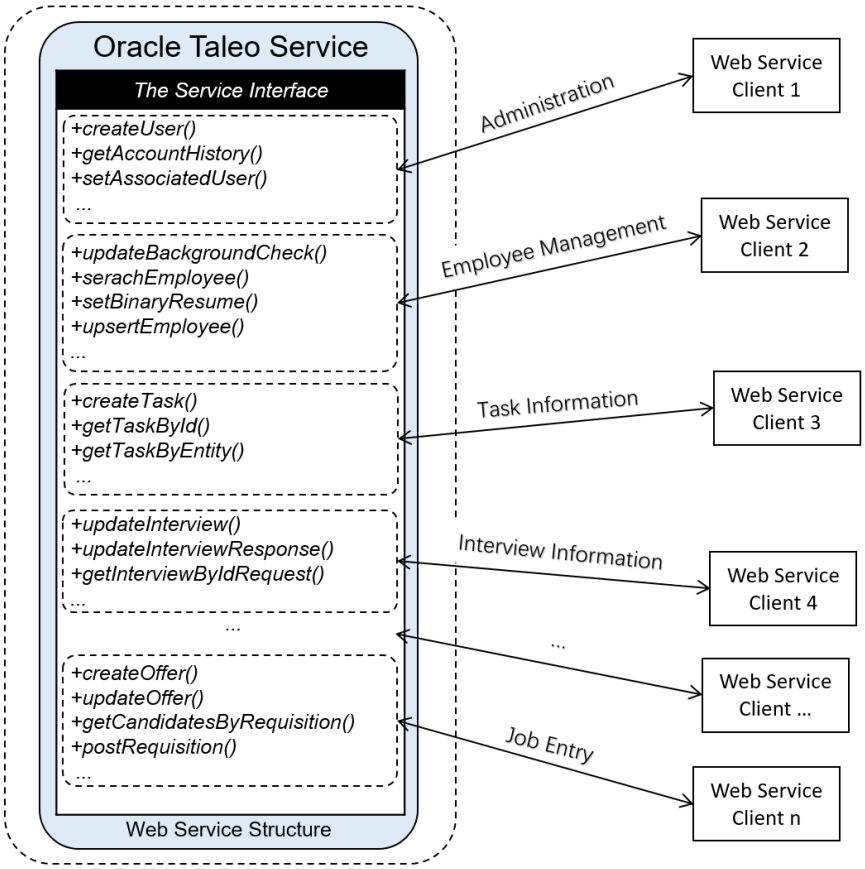


Figure 17 An example of god object Web service provided by Oracle Taleo.

In this section, we illustrate a real-world example of god object Web service(GOWS) antipattern provided by Oracle Taleo[1]. Oracle Taleo is a famous talent acquisition service which enables companies to easily source, recruit or manage talents. The antipattern in this example,

[1] Oracle Taleo: https://tbe.taleo.net/

GOWS, has a key characteristic that it implements uncohesive operations of many core business or/and technical abstractions. Figure 17 shows the interface[2] of Oracle Taleo which contains a large amount of operations for different business abstractions. In this example, total of 127 operations are implemented within a single port type. These operations represent different functionality aspect of the service, such as administration, employee management, task information, interview management, job entry, job allocation, and so on. For instance, *createUser*() is an administrative operation that creates new authorized user of the service, *updateBackgroundCheck*() attempts to update the background check data for employee management, while *getCandidatesByRequisition*() lists the potential candidates who fit for a specific requisition. These operations are desired and used by different service users, e.g., companies who want to hire and companies who want to provide talents.

From the QoS perspective, Taleo service also behaves as GOWS. According to the QWS dataset[3], there is only a little documentation (i.e. description tags) to help developers understand all the operations of Taleo service. Furthermore, the Taleo service takes long ($latency = 232.46ms$) to process the request, and the availability of the services is relatively low (40% of the error messages to total messages). Overall, these behaviors of GOWS, could cause difficulties of reuse or reduce the practical value of the service.

For the service clients, it is hard to know what is the appropriate design or performance of the Web service. On the other hand, the antipatterns are hard to avoid by the service providers, due to the changing requirements or human factors/resource of the development. In fact, the services

[2] Oracle Taleo Interface: https://tbe.taleo.net/wsdl/WebAPI.wsdl

[3] The QWS Dataset:http://www.uoguelph.ca/ qmahmoud/qws/

that contain more useful functionalities, could have a higher probability to certain antipatterns (e.g. GOWS in our example). Therefore, it is important to find the solution for service clients to detect antipatterns efficiently based on the programming and QoS metrics.

3.3.3 **Collection of Metric Suite**

Quality metrics can be used to extract the semantic and structural attributes of the Web services. These quality indicators can then be used to quantitatively track and evaluate the design patterns of Web Services architecture. The antipatterns detection process usually involves finding the fragments of the design which violate these metrics. In our previous work [35], we used a set of static Web service metrics from interface and code level, and one dynamic metrics, namely response time [11]. The static metrics aim at measuring the structural information of Web services in different levels. Table 7 describes all of the metrics that are being used in this work.

For the dynamic level, the response time was valuable to prove the concept of using QoS metric for us. However, it is not enough to measure full dynamic behaviors of the services. For instance, in the motivating example of Figure 18, the Oracle Taleo service has an acceptable response time of $442.53\,ms$. The response time includes network traffic time and latency time which is the time taken for service to process the request. However, Taleo suffers from a high latency time of $232.46\,ms$ compares to the others. This situation means that even though the service has good facilities and network configurations to provide a reasonable response time, it still suffers from high process time due to the GOWS antipattern, and by only using response time metric, this information can not be reflected in the metric suite. To better detect the antipattern of web services from dynamic Internet environment, we extend the dynamic metric suite to 9 parameters in this work. The dynamic and static metrics we used in this work are described as follow:

1) Web service QoS metrics

To detect antipattern from the QoS aspect, we introduced 9 popular metrics from the literature [85]–[87]. Documentation, compliance, and best practices are static metrics extracted based on interface level to extend our static metrics, they measure the usability of the web service interface from QoS aspect. Response, availability, throughput, successability, reliability and latency are dynamic metrics which measure the web service overall performance and experience. In this work, we use the QWS dataset, a widely used QoS benchmark in field of Web service and service composition [87]–[90]. Commercial benchmark tools are used to extract the parameters and each service was tested over a ten-minute period for three consecutive days.

Table 7 The collection of metrics used for service defect detection.

Metric Name	Definition	Metric Level
Response	Time taken to send a request and receive a response (ms)	QoS
Availability	Number of successful invocations/total invocations (%)	QoS
Throughput	Number of invocations for a given time (invokes/sec)	QoS
Successability	Number of response / number of request messages (%)	QoS
Reliability	Ratio of number of error messages to total messages (%)	QoS
Compliance	The extent to which a WSDL follows specification (%)	QoS
Best Practices	The extent to which a service follows WS-I Basic (%)	QoS
Latency	Time taken for the server to process a given request (ms)	QoS
Documentation	Measure of documentation (e.g. description tags) (%)	QoS
ALPS	Average length of port types signature	Interface
COH	Cohesion	Interface
COUP	Coupling	Interface
NAOD	Number of accessor operations declared	Interface
NCO	Number of CRUD operations	Interface
NOD	Number of operations declared	Interface
NOPT	Average number of operations in port types	Interface
NPT	Number of port types	Interface
RAOD	Ratio of accessor operations declared	Interface
ALOS	Average length of operations signature	Interface
AMTO	Average number of meaningful terms in operation names	Interface
ANIPO	Average number of input parameters in operations	Interface
ANOPO	Average number of output parameters in operations	Interface
NPO	Average number of parameters in operations	Interface
ALMS	Average length of message signature	Interface
AMTM	Average number of meaningful terms in message names	Interface

NOM	Number of messages	Interface
NPM	Average number of parts per message	Interface
AMTP	Average number of meaningful terms in port type names	Interface
NCT	Number of complex types	Interface
NCTP	Number of complex type parameters	Interface
NST	Number of primitive types	Interface
RPT	Ratio of primitive types over all defined types	Interface
Ca	Afferent couplings	Code
CAM	Cohesion Among Methods of Class	Code
CBO	Coupling between object classes	Code
Ce	Efferent couplings	Code
DAM	Data Access Metric	Code
DIT	Depth of Inheritance Tree	Code
LCOM	Lack of cohesion in methods	Code
LCOM3	Lack of cohesion in methods	Code
LOC	Lines of Code	Code
MFA	Measure of Functional Abstraction	Code
MOA	Measure of Aggregation	Code
NOC	Number of Children	Code
NPM	Number of Public Methods	Code
RFC	Response for a Class	Code
WMC	Weighted methods per class	Code
AMC	Average Method Complexity	Code
CC	The McCabe's cyclomatic complexity	Code

2) Web service interface-level (WSDL) metrics

There are fifteen metrics used in this work from the interface level. These metrics defined in the literature [6], [11], [12], [91] measure design concepts from interface type, message, operation and Port type levels. Most metrics are calculated directly based on the information of Web service interface description file. For *AMTO*, *AMTM*, and *AMTP*, they are implemented by comparing the tokenized identifiers of ever operation, port type and message with lexical database, WordNet[4].

3) Web service code-level metrics

[4] WordNet: http://wordnet.princeton.edu/

Web service only exposes the interface for clients to use while the source code is not available to access. In this work, code-level metrics [92], [93] are extracted from the service code skeletons which are generated by JAX-WS[5] (a Java API) for XML Web services. The code-level metrics used in this work are defined by Chidamber and Kemerer [49]. The ckjm tool[6] is used to extract these metric to reflect design quality from a deeper level of Web service.

The detection rules generated by our approach are composited by the metrics mentioned above. The dimensions of the solution space are set by the metrics associated with greater/less than, their threshold values, and logical operations between them, e.g., union ($metric1 > a \ OR \ metric2 < b$) and intersection ($metric1 > a \ AND \ metric2 < b$). A solution is a composited logical expression by multiple metrics, e.g., $metric1 > a \ AND \ (metric2 < b \ OR \ metric2 < b)$. By nature, this is a combinatorial optimization problem with a large search base(number of possible solutions is huge). A heuristic search algorithm is desired in this problem. Furthermore, since we also try to generate detection rules that can satisfy different detection strictness, multi-objective evolutionary algorithm - NSGA-II [40].

3.3.4 **Solution Approach**

1) Problem Statement

Efficient identification of refactoring opportunities is beneficial to the service clients and providers. However, there is no general consensus on how to decide if a specific design is a violation of the quality principles. Design patterns that contain antipattern symptoms, may not

[5] JAX-WS: http://docs.oracle.com/javase/6/docs/technotes/tools/share/wsimport.html

[6] ckjm tool: http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/

necessarily be an actual antipattern to the clients [12]. The reason is that every service user/provider has different understandings, requirements or contexts. For instance, some Web services focus to provide single business service to convenient their client (e.g., translation, weather, and calculation service). These services might be designed properly, but they still have a high chance to be classified as fine-grained Web service (FGWS) due to the size of service.

One major challenge of antipattern detection is to find the suitable metric combination to define the antipattern. Single metric can't represent the full characteristic of any antipattern. Also, manual work to find the metric composition is time-consuming and could have bias. Based on the antipattern symptoms in the literature [10], [11], [94], many work [11], [12], [35], [36], [95] have been proposed to find detection rules based on the interface-level/code-level metrics. These static metrics provide quantitative information of the design. However, transferring natural description to detection rules is difficult and could be subjective. By using only the static metrics, the detection rules could have less practical value. In fact, most client users treat QoS performance and functionalities of Web service as major concerns [83], [96].

Another challenge is to find the right threshold value for each metric of detection rule. A threshold value is needed for quantitative metrics to transform the description to detection rules. Similar to the previous challenge, there is no general opinion on the degrees of antipattern. To make it worse, after introducing more metrics of QoS perspective, this NP-hard problem by nature is getting more complex.

The final challenge raises when the detection problem gets subjective and complicated. The detection rules generated by most existing approaches are fixed results, in another word, there is only one detection method for each antipattern. Since there is no "best" result due to the various requirements or background of the users, it's common that different users may feel the result is too

strict or too flexible. For instance, a general detection rule of GOWS is not suitable for the users who usually deal with complex Web services. On the other hand, a general FGWS detection rule is not preferred by the developers who look for light-weight services for composition. However, the users can't have an alliterative choice based on their preference or need.

To address or circumvent the above-mentioned challenges, we introduce new QoS metrics to this problem, and propose a multi-objective heuristic-based approach which automatically detects Web service antipatterns and generates solutions for different preferences as detailed in the next sub-section.

2) Solution Approach Overview

Given a set of service metrics and their value ranges, there are many ways in which the rule with metric combination can be combined and leading to different detection results. This problem is an NP-hard problem by nature, therefore it should be suited to a meta-heuristic search-based approach [15], [65], [97]. Also, our problem requires a search for a solution which balances the objectives to generate rules suitable for different scenarios.

Figure 18 shows our approach to the QoS-aware Web service refactoring opportunities identification problem. It targets to explore the large search space and finds a set of optimal detection rules, by combining metrics and their threshold values. The output is a set of detection rules which are the optimal solutions to the two conflict objects of the search-based algorithm

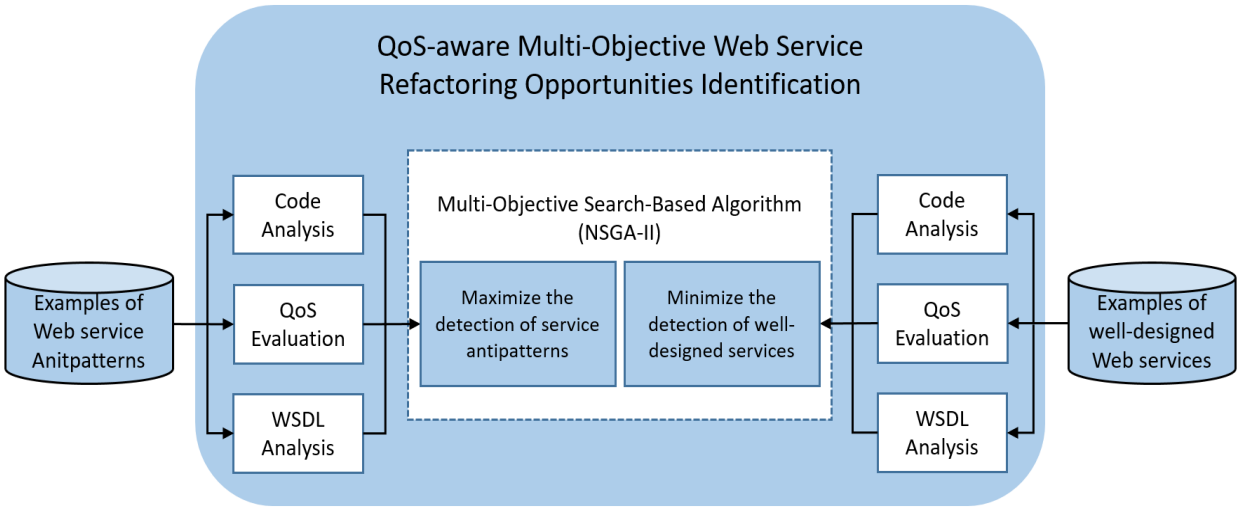


Figure 18 QoS-aware detection approach overview

The approach takes two sets of Web service examples: one set contains service antipattern examples and another has well-designed service examples. These example sets are selected from the QWS dataset[7], which includes a set of 2,507 Web services and their QWS measurements. We took a sample of 500 services from this dataset, and manually inspect and validate the antipatterns of these services based on the existing guidelines. The approach processes and calculates the metrics of each service in the sets through: (*i*) QoS Evaluation: measures real-time metrics from the services or documentation metrics from the interface file, (*ii*) Interface Analysis, parses the interface source though tree walking up the XML hierarchy to extract the Web service structure data (e.g., operation, message, and input/output), then calculates the interface level metrics, and (*iii*) Code Analysis, extracts the Web service code skeleton and uses typical object-oriented metrics to evaluate. The metric suite used in this work, which contains a total of 49 metrics, is described in the previous section.

[7] The QWS Dataset: http://www.uoguelph.ca/~qmahmoud/qws/

Then, the metric data of two service example sets are passed to multi-objective algorithm among with service metric types and threshold ranges. The search-based algorithm - NSGA-II [40], generates, evaluates and selects antipattern detection rules based on the following objectives: (*i*) Maximizing detection number of the service antipatterns, and (*ii*) Minimizing the detection number of well-designed services. The details of this step and algorithm are described next.

3) Problem Adaptation

To adapt a search-based algorithm to a specific problem, the following elements should be defined: (*i*) solution representation and the generation of initial population, (*ii*) fitness function to evaluate candidate solutions according to each objective, (*iii*) Evolutionary operators to generate new individuals using genetic operators (crossover and mutation). In the following, we describe how we formulate the problem and these elements.

1) Solution Representation:

The candidate solutions we seek in this problem are the antipattern detection rules. Each solution is a logical expression, and the antipattern is detected while the condition in the logical expression is satisfied.

The logical expression is encoded in a tree-based structure and connects every metric with its threshold value using a logic operator ("AND" or "OR"). If the expression is satisfied by a Web service, then it is determined to be of the antipattern type associated with this solution. A solution is randomly generated at the beginning, and can be "mutated" by itself or "crossovered" with another solution to generate new ones. Figure 19 provides an example of a candidate solution. Formally, each candidate solution is a sequence of detection rules where each rule is represented by a binary tree such that:

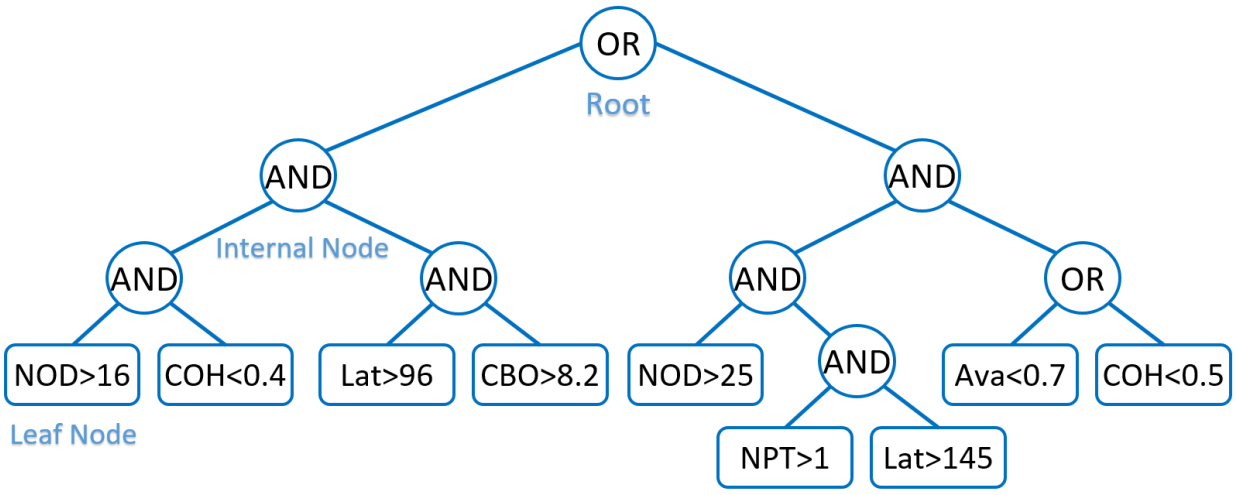


Figure 19 Example of NSGA-II solution representation

- The Root R and each internal node N represents a logic operator to connect other nodes or leaf, either AND or OR.
- Each leaf node L represents a metric (from the metrics described in section 3.3) and its corresponding threshold (generated randomly among the range of the metric).

Each solution represents a detection rule for one specific type of service defects, and each execution of the approach only generate the solution set which is used for one antipattern. In this work, we focus on detecting eight popular types as defined in Chapter 2.

2) Fitness Functions:

The quality of each solution is determined by the fitness functions in multi-objective problems. Each fitness function evaluates one objective by calculating a specific value that is desired to be either minimized or maximized for a solution. In this problem, we aim to optimize the following two finesse functions: (*i*) Maximizing the coverage of antipattern examples. (*ii*) Minimizing the detection of good design practice examples of Web services. The collected examples of well-designed Web services and antipatterns and the metrics of these services are taken as the input of NSGA-II. In algorithm iterations, each solution (detection rule) is applied to

both example sets, and evaluated by the fitness functions. Analytically speaking, the formulation of the multi-objective problem can be state as follows:

$$\begin{cases} \max f_1(x) = \dfrac{\dfrac{||\ DCS(x)\ ||\cap||\ ECS\ ||}{||\ ECS\ ||} + \dfrac{||\ DCS(x)\ ||\cap||\ ECS\ ||}{||\ DCS\ ||}}{2} \\ \min f_2(x) = \dfrac{\dfrac{||\ DCS(x)\ ||\cap||\ EGE\ ||}{||\ EGE\ ||} + \dfrac{||\ DCS(x)\ ||\cap||\ EGE\ ||}{||\ DCS(x)\ ||}}{2} \end{cases}$$

where is the cardinality of the set of detect antipatterns by the solution , is the cardinality of the set of antipattern examples, and is the cardinality of the set of well-designed service examples. These two fitness functions drive the algorithm to search for the optimal solutions by comparing the list of detected antipatterns with the expected ones from the base of examples along with the percentage of covered well-designed examples. Once the bi-objective trade-off Pareto front is generated, service developers/users can select preferred detection rule from the best solution rules to detect potential antipatterns on any new Web service.

3) Evolutionary Operators:

Evolutionary algorithms deploy change operators to generate new solutions in each iteration. Except the initial population is randomly generated, all other candidate solutions are generated by applying change operators to the existing ones. These generated solutions can explore the search space, potentially and eventually, increase the diversity and approach to better solutions. We use crossover and mutation operators in this work to produce offspring solutions.

✧ Mutation Operator:

The mutation operator is used to apply minor random change into one parent solution. This operator promotes the algorithm into the location of the search space that would not be reachable through recombination alone and avoids the convergence of the population towards a

few elite solutions. In our adaptation, the mutation operator first picks a random node of the parent solution. Then, if a non-leaf node is selected, the operator simply replace it by another possible function (e.g., replace “AND” by “OR”). If a leaf node is selected, the mutation operator assigns either a new threshold value for the existing metric, or a new metric with a new random threshold value to replace the leaf completely. Figure 20 shows an example of mutation, the highlighted leaf ($NOD < 6$) of $Parent_1$ is selected and replaced by a new metric ($CBO > 8.2$) to generate new solution $Child_1$.

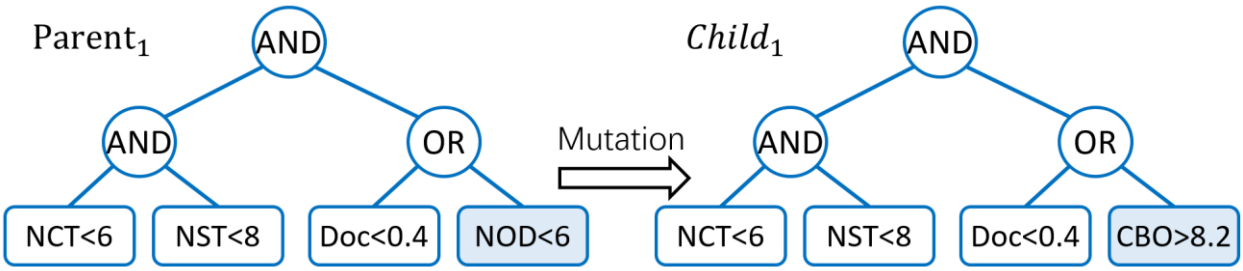


Figure 20 Example of mutation

✧ Crossover Operator

In population-based algorithms, the crossover operator is using more than one solution to create the new and different solutions, e.g., re-combining solutions into ones. In our approach, we use a single, random cut-point crossover to generate offspring solutions. Two cut-points are selected randomly in two parents, more specifically, two non-leaf nodes of the solutions. Then all the relative sub-trees from the cut-point nodes are swapped to generate new solutions to perform crossover action. Therefore two new solutions are created after the crossover. An example of crossover is described in Figure 21, two highlighted sub-trees are selected and swapped from $Parent_1$ and $Parent_2$, then two new solutions, $Child_1$ and $Child_2$ are generated.

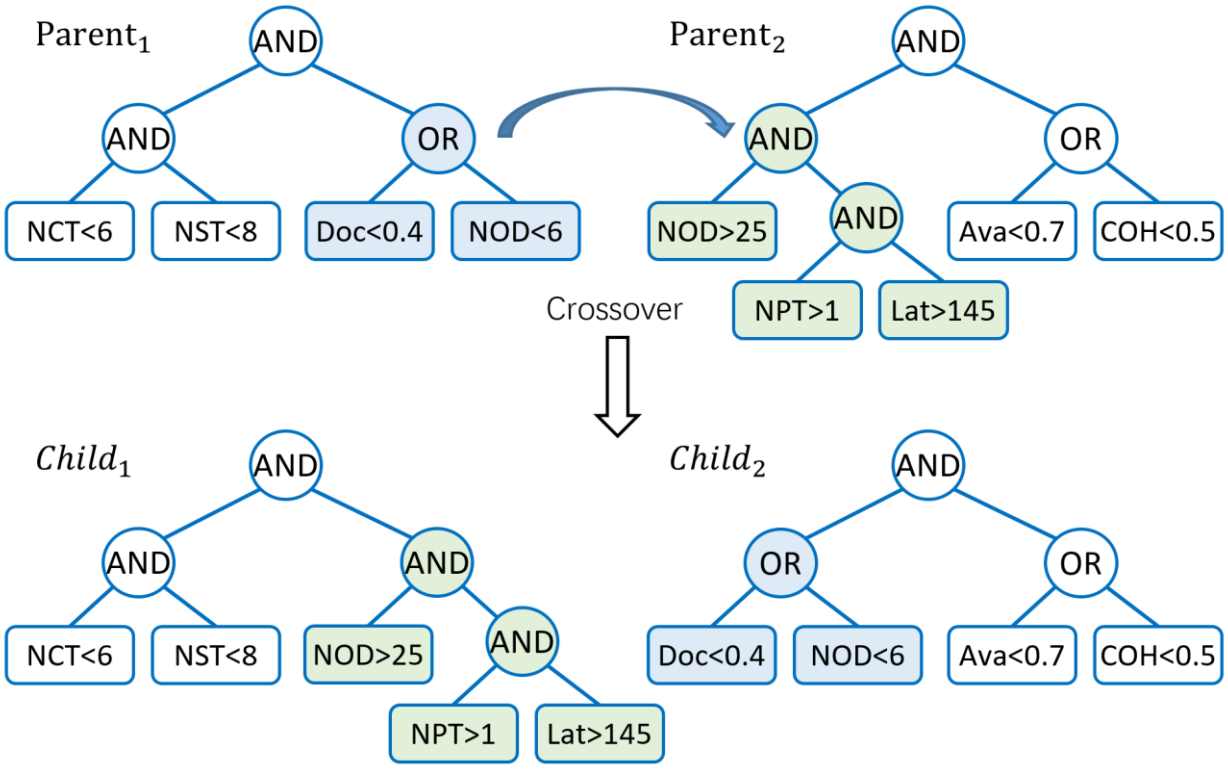


Figure 21 Example of crossover

3.3.5 **Validation**

1) Experimental Setup

To validate our approach, experiments are designed to answer the following research questions:

- RQ1. To what extent does QoS dynamic metric improve the antipattern detection of Web services?
- RQ2. How does our multi-objective approach compare to random search and mono-objective search-based algorithm?
- RQ3. How does approach perform compared to other existing Web service antipattern detection approaches [3], [26]?

To evaluate our approach, we select a popular QoS benchmark, the QWS dataset[8] which includes a set of 2,507 Web services and their QWS measurements. We randomly selected a sample of 500 available services from the dataset, extract their interface file and code skeleton, and manually inspect and validate the antipatterns of these services based on the existing guidelines [4], [10]. To avoid possible biases in the empirical study, we select services covered different sizes, QoS performance, and application categories (such as financial, science, travel, weather, and so on). Table 8 summarizes the selected Web services.

Table 8 Overview of 500 Web services used in the empirical study

	NOD	**NOM**	**NCT**	**Response**	**Latency**	**Availability**	**Throughput**	**Reliability**
Max	231	462	287	3768.33 *ms*	1991 *ms*	100%	41.2/sec	89%
Min	1	1	0	46.05 *ms*	0.33 *ms*	9%	0.2/sec	33%
Average	14.43	30.44	21.40	343.10 *ms*	52.84 *ms*	84.81%	8.01/sec	68.21%

In this work, we validate our approach on eight common antipattern types, namely, god object web service (GOWS), fine-grained Web service (FGWS), chatty Web service (CWS), data Web service (DWS), ambiguous Web service (AWS), redundant port types (RPT), CRUDy interface (CI), and maybe it is not RPC (MNR) as described. We use 10-fold cross-validation method to evaluate our approach. For the services in each fold, are used to test the detection rule set generated by examples in other nine folds. Therefore, in the experiments, 450 services are selected as training examples(ground-truth) to execute the algorithm, and rest 50 services are used as the test set. Precision and recall [65] are used to evaluate the accuracy and effectiveness of our approach. Precision represents the ratio of true antipatterns detected to the total number of detected antipatterns, and recall denotes the ratio of true detected antipatterns to the total number of real antipatterns in the test set. Since our approach generates a solution set, we evaluate on all generated

[8] The QWS Dataset: http://www.uoguelph.ca/~qmahmoud/qws/

detection rules and record the one with highest average precision and recall to compare. During the piratical implementation of our approach, the users can select based on their preference to better detect the defects.

While comparing different search-based evolutionary algorithms, changing parameters could lead to completely different results (e.g., low number of iteration). To ensure fair comparisons, we used same parameters for the all the evolutionary algorithm experiments as following: $PopulationSize = 300$, $MaxIteration = 1000$, $MaxDepth_{Solution} = 10$, $R_{crossover} = 0.8$, $P_{mutation} = 0.1$. We used a high number of population size due to the size of search space and solution combination, a small population size leads to low diversity in our approach. Also for the mutation probability is relatively high (compare to applications of NSGA-II in other fields), to ensure the good convergence.

To answer RQ1, we investigate the effectiveness of using QoS metrics on different types of service antipattern, since this is one of the main novelties in our work. To this end, we compared the results of our approach, with the results of the approach using the same algorithm, settings, and training/testing sets, but only use interface-level and code-level metrics as described.

To answer RQ2, we investigate the efficiency of using NSGA-II and our problem formulation. We compared our approach to random search and mono-objective genetic algorithm. Since another novelty of our approach is using multi-objective optimization search-based techniques, it's important to compare with the random search (RS) to prove the adaptation is adequate [98]. Therefore, we implemented a random search using same training and test sets. Further more, an genetic algorithm is implemented with an aggregated fitness function (average of our two fitness functions) to validate if our multi-objective approach is able to improve the detection process.

To answer RQ3, we compared our approach with state-of-the-art detection approaches: a search-based approach from [3] and SODA-W of [26]. All three approaches were tested on the same services examples described in Table 8.

2) Experiment Results

The goal of RQ1 is to investigate the importance of QoS metric in detecting service antipattern. We executed both approaches (with and without QoS metrics) 5 times for the each fold of the validation (total of 50 runs). Figure 22 report the comparative results. With same experiment settings, our multi-objective approach with QoS metrics performs slightly better than the one with only static metrics. The average precision is improved from 91.4% to 93.2%, while recall is improved is 89.4% to 91.6%. For antipattern types that have negative impacts on the service performance such as GOWS and MNR, the QoS metrics improved the accuracy service. However, for few antipattern types such as AWS, DWS, and RPT, the precision and recall remain the same because these antipatterns don't affect the QoS parameters of the service. Thus, the QoS-aware approach shows effectiveness in detecting antipatterns that are related to the service dynamics. Furthermore, by observing result of our approach on different types of antipatterns. We had high precision and recall score very equally for all eight types of antipatterns. By using QoS metrics related to real-time performance like latency and availability, we manage to have promising detection results in antipatterns like FGWS, MNR and GOWS, especially for FGWS we reached 100% precision for all the tests. The reason is that these antipatterns have substantial impacts to service performance and signatures on the structural documentation as well. While checking lower scores for few types like AWS which is more related to human understandability issue, we managed to identify most of them by using parameters like AMTO, AMTM, and AMTP. However, it's in general hard to detect only by the design or QoS metrics.

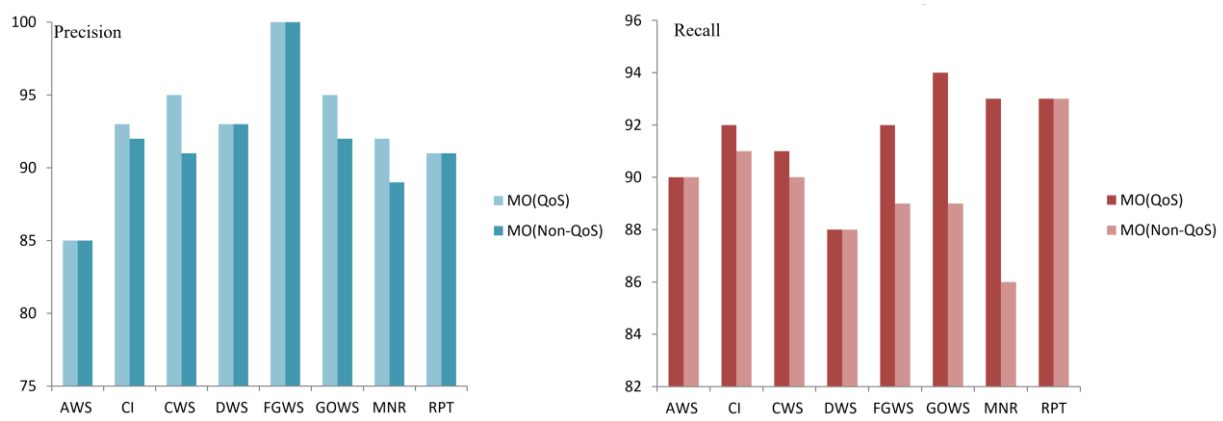


Figure 22 Comparative results of multi-objective approaches with and without QoS metrics

The result of RQ2 is presented in Figure 23. Over 50 runs for 8 types of antipatterns, the random search (RS) didn't perform well, the average precision is 26.2%, and average recall is 27.3%. The main reason to this is obvious, which is the very large search space of this problem due to the high possible combinations of metrics with possible thresholds. While comparing to result of our approach in the figure, this shows we made a success evolutionary adaptation to antipattern detection problem. Furthermore, comparing our approach with the mono-objective genetic algorithm (GA), the results were close due to the same evolutionary steps, problem adaptation, and solution representation in the algorithm. The average precision and recall for GA were 84.8% and 84.1%, and the experiments also show that there is no obvious bias to any specific antipattern type. However, QoSMO outperforms GA significantly in average, while QoSMO can generate at least one better solution than the solution of GA in each single test. The reason is mainly due to the limitation of GA is only able to handle this problem as mono-objective and output one solution which is limited by the single fitness function. As the conclusion to this research question, our approach outperforms GA and RS and proves to be a success use of multi-objective evolutionary algorithms.

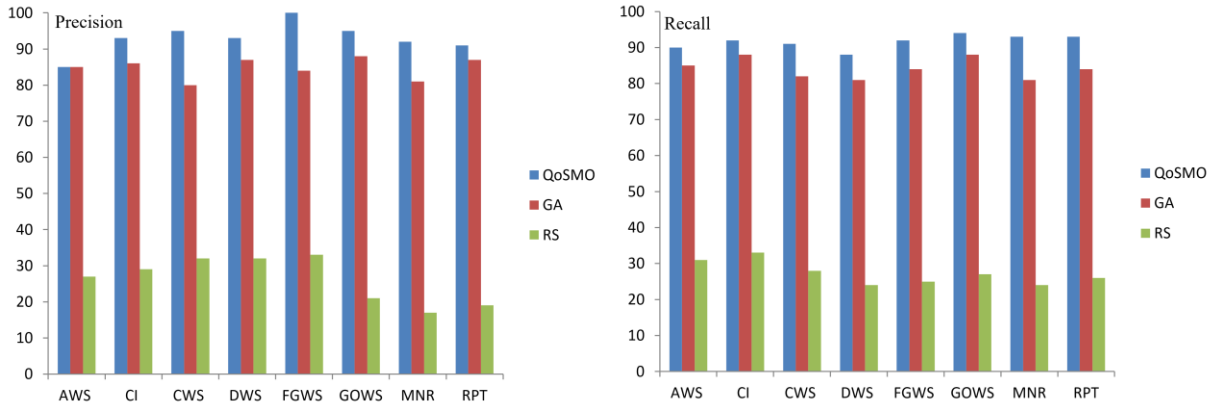


Figure 23 Comparative results of QoSMO, GA and RS

Results for RQ3. Figure 24 reports the comparison study of our approach (QoSMO), PE-A [3], SODA-W [26]. PE-A performs well an average precision of 88.8% and a recall of 90.0%, it's one of the best state-of-the-art algorithms we tested. However, it's still less than QoSMO in detecting 7 types of antipatterns and performs same as QoSMO in detecting AWS. PE-A is using a cooperative parallel model to combine GA and GP, it's limited by using non-QoS metrics and mono-objective search algorithms. Serval antipatterns are easier to detect using QoS metric as we describe in RQ1, and mono-objective approach may produce a solution that has a certain bias in optimizing an aggregated fitness function. SODA-W also has good results with an average precision of 69.8% and recall of 77.9%. The limitations of this work include the ones we mentioned earlier for PE-A, and not considering the source code of the service artifacts. These two levels of metrics reflect the design of different layers, and both layers are necessary to evaluate the static assessment of service design. Also, PE-A uses an exhaustive approach, which could limit the result by the large search base. Different from PE-A and SODA-W, our approach using not only both static metrics, but also QoS metrics to evaluate the real-time performance of the services and multi-objective approach to better manage the challenge for the larger search space.

Figure 24 Comparative results of QoSMO, PE-A and SODA-W

External threats may exist because in this work, we did not evaluate our approach on all possible antipattern types. However, the eight types of Web service antipatterns we employed constitute a broad representative set of standard and frequent defects. In addition, we did not yet generalize our approach for other service types such as RESTful services. It is also possible to extend the work for other domains such as mobile apps to validate the generality of our approach.

Construct threats: This type of threats is caused by the relationship between theory and what is observed. A possible threat is related to the antipattern examples that are being used to train/validate the approach, as the users may not agree with classified antipatterns. As we mentioned early, there is no general consensus on how a specific design violates the quality principles. This is indeed one key motivation to use multi-objective search-based approach to generate a set of solutions for users to choose from based on their preferences. In our experiments, the standard metrics such as precision and recall are used to validate the proposed approach, these metrics are widely used in validating code smell detection tools. As part of our future work, we may need to conduct a survey with developers to study the relevance of detected antipatterns.

3.3.6 **Conclusion**

We introduced a search-based multi-objective approach to generated detection rule solutions as a composition of QoS, Interface, and code metrics. In our multi-objective adaptation, two fitness functions are used to maximize the coverage of antipattern examples and minimize the coverage of well-designed Web service examples. The proposed approach is evaluated on 500 Web services of a QoS benchmark and eight common Web service antipattern types. The empirical study shows that proposed QoS-aware antipattern detection outperforms our previous multi-objective approach and other state-of-the-art approaches with an average precision score of 94% and a recall of 93%.

Chapter 4 **Detection of Changes among Service Releases**

4.1 **Introduction**

Systems implemented based on Web services depend on the interface that only shows the list of available operations/features to the subscribers. The users of the services would be discouraged to integrate a new release of an existing Web service if major changes are introduced. In fact, they have to introduce changes to their implementation of the system to be coherent with the new service. Thus, it is critical to provide support for developers to better understand the introduced changes to the services. Some recent studies were proposed to understand the evolution of Web services especially at the interface level [2], [51], [57], [59].

The majority of the changes in a web service interface typically affect the systems of its subscribers. Thus, it is important for subscribers to estimate the risk of using a specific service and compare its evolution to other services offering the same features in order to reduce the effort of adapting their applications in the next releases. Subscribers prefer to use, in general, Web services that are stable with a low risk to include bugs and introduce major revisions in the future. Thus, a support to compare between multiple releases of a service may help the developers to select the best service in terms of stability between multiple competing ones.

In this work, we propose a genetic algorithm approach [99] to detect composite changes between multiple Web service releases. Our approach takes as input an exhaustive list of possible change types, the initial release and the revised one, and generates as output a list of detected changes in terms of refactorings (composite changes). A solution is defined as the combination of

refactoring operations that should maximize the structural and textual similarity between the expected new Web service interface release and the generated one after applying the refactoring sequence on the initial release. Due to the large number of possible solutions, a search-based method, based on Genetic Algorithms (GA) is used instead of an enumerative one to explore the space of possible solutions.

We evaluated our approach on a set of 6 popular Web services including more than 110 releases. We report the results on the efficiency and effectiveness of our approach to detect changes of the evolution of Web services interfaces. The results indicate that the detection results of several Web service metrics, on the different releases of the 6 Web services, were correct with an average precision and recall respectively higher than 86% and 89%.

4.2 **Approach**

4.2.1 **Overview**

As described in Figure 25, the proposed approach takes as input two or more Web service releases, and as controlling parameters, an exhaustive list of Web service refactoring operations. The approach generates a set of refactoring applications that represents the evolution from the initial release to the target one. The process of detecting Web service changes between two releases can be viewed as the search mechanism that finds the best way to combine refactoring operations of the exhaustive list of possible refactorings, in such a way to maximize the structural and semantic/textual similarity between the initial and target releases when applying the detected refactorings on the initial one.

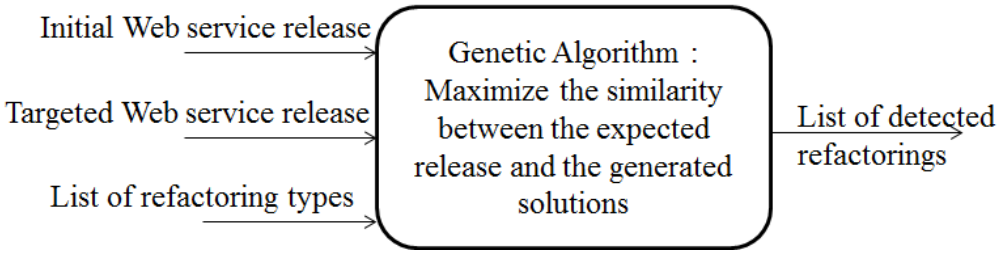


Figure 25 Genetic algorithms for the detection of changes among multiple releases

Due to the large number of possible refactoring solutions between the Web service releases, we consider changes detection as an optimization problem. The algorithm explores a large search space. In fact, the search space is determined not only by the number of possible refactoring combinations, but also by the order in which they are applied. To explore this search space, we use a mono-objective Genetic Algorithm. This algorithm and its adaptation to our problem are described in the next section.

4.2.2 **Adaptation**

Genetic Algorithm (GA) is a population-based metaheuristic inspired by Darwinian Theory. The basic idea is to explore the search space by evolving a population of solutions for a pre-specified number of generations. The algorithm next gives the pseudo-code of a canonic elitist GA. The Elitism concept consists that a sub-optimal solution could not be favored for survival over a better one. Its basic iteration is as follows. A parent population *P* is generated (by environmental selection except for the initialization step where it is produced randomly) and each of its individuals is evaluated. Once the evaluation step is performed, we fulfill the mating pool by selecting parents from *P*. These selected parents are then subject to genetic operators (crossover and mutation) in order to generate an offspring population *Q*. Once offspring individuals are evaluated, *P* and *Q* are merged to form the population *U*. We perform now environmental selection on *U* by selecting fittest individuals and thereby we generate the parent population for the next

generation. Elitism is then ensured by saving best individuals coming from parent population *P* and offspring population *Q* in each generation of the algorithm. Once the termination criterion is met, the GA returns the best (fittest) individual from *P*.

GA pseudo-code
1: $P_0 \leftarrow$ *random_initialization()*
2: $P_0 \leftarrow$ *evaluate*(P_0)
3: $t \leftarrow 0$;
4: ***while*** *(NOT termination_condition)* ***do***
5: $Parents \leftarrow$ *parent_selection*(P_t)
6: $Q_t \leftarrow$*genetic_operators(Parents)*
7: $U_t \leftarrow$ *merge* (P_t, Q_t)
8: $P_{t+1} \leftarrow$*environmental_selection*(U_t)
9: $t \leftarrow t+1$;
10: ***end***
11: $s \leftarrow$ *fittest*(P_t)
12: ***return*** *s*

4.2.3 **Solution Representation**

One key issue when applying a search-based technique is to find a suitable mapping between the problem to solve and the techniques to use, i.e., in our case, detecting Web service changes. The i^{th} individual (solution) represents a combination of refactoring operations to apply. The order of applying refactorings corresponds to their position in the vector (referred to as dimension number in the following). In addition, the execution of the refactorings is respecting pre- and postconditions to avoid conflicts and semantic inconsistencies. Furthermore, it has to be noted that the same type of refactoring operation could be applied several times in the same solution (but to different interface elements). The list of considered refactoring types in our experiments and their controlling parameters are the following:

1. *ExtractOperation (name of source operation, name of the new operation, list of operation parameters to move);*
2. *MoveOperation (name of source port type, name of the target port type, name of the operation);*
3. *MergeOperations(name of the first operation, name of the second operation, etc.);*
4. *AddOperation(operation name, operation parameters);*
5. *RemoveOperation(operation name, operation parameters);*
6. *RenameOperation(operation name, new name);*
7. *AddParameter(operation name, parameter name);*
8. *RemoveParameter(operation name, parameter name);*
9. *RenameParameter(operation name, parameter name, new name);*
10. *AddPortType(port type name);*
11. *RemovePortType(port type name);*
12. *RenamePortType(port type name, new name).*

Initial Population Generation. To generate an initial population, we start by defining the maximum vector length including the number of refactorings. The vector length is proportional with the number of refactorings to use for detecting Web service changes. Sometimes, a high vector length does not mean that the results are more precise, but that only a few refactorings are sufficient to detect changes. These parameters can be specified either by the user or chosen randomly. Thus, the individuals have different vector length (structure). Then, for each individual we randomly assign one refactoring, with its parameters, to each dimension. The generated solutions (refactoring combination) are applied to the initial release of the Web service to generate a new one. Then, the new generated Web service will be evaluated by the fitness function to check its similarity with the expected release.

Table 9 List of considered structural metrics

Metric Name	Definition
NPT	Number of port types
NOD	Number of operations declared
NAOD	Number of accessor operations declared
NOPT	Average number of operations in port types
ANIPO	Average number of input parameters in operations
ANOPO	Average number of output parameters in operations
NOM	Number of messages
NBE	number of elements of the schemas
NCT	Number of complex types
NST	Number of primitive types
NPM	Number of parts per message
COH	Cohesion
COU	Coupling
AMTO	Average meaningful terms in operation names
AMTM	Average meaningful terms in message names
AMTMP	Average meaningful terms in message parts
AMTP	Average meaningful terms in port-type names
ALOS	Average length of operations signature
ALPS	Average length of port-types signature
ALMS	Average length of message signature

4.2.4 **Fitness Functions**

The fitness function quantifies the quality of the proposed detection solutions. The goal is to define an efficient and simple fitness function in order to reduce the computational complexity. The proposed function is based on two main components:

- Maximizing structural similarities between the initial and target Web service interface elements
- Maximizing the syntactic/textual similarities between the initial and target Web service interface elements

In this context, we define the fitness function to maximize as:

$$f = \frac{\sum_{i=1}^{n} StructureSim(e_i) + SyntacticSim(e_i)}{2 * n}$$

where n is the number of Web service interface elements. We treat both components of the fitness function with an equal importance and we normalized it in the range of [0, 1].

The structural similarity is calculated using the metrics defined in Table I, whereas the goal is to minimize the difference between the metric values of matched source and target elements. Thus, this similarity is defined as follows:

$$StructureSim(e_i) = 1 - \frac{\sum_{j=1}^{t} \left| sqm_{i,j} - tqm_{i,j} \right|}{\sum_{j=1}^{t} Max(sqm_{i,j}, tqm_{i,j})}$$

where t is the number of target Web service interface elements matched to source Web service interface elements e_i; *sqm* (for e_i) and *tqm* (for the matched element(s)) are the average metrics values used to characterize the structure as described in Table 9.

The syntactic similarity $SyntacticSim(e_i)$ of a source Web service interface element e_i corresponds to the weighted sum of each vocabulary used to calculate the similarity between e_i and the target Web service interface elements matched to e_i. Hence, the syntactic similarity of a solution corresponds to the average of syntactic coherence for each source Web service interface element:

$$SyntacticSim(e_i) = \frac{\sum_{k=1}^{t} SynHomog(e_i, e_k)}{t}$$

where $SynHomog(e_i, e_k)$ is the average of syntactic measures applied between the source Web service interface element e_i and the matched Web service interface element e_k of the new release.

We start from the assumption that the vocabulary that is used for naming the interface elements is borrowed from the respective domain terminology and then we determine which part of the domain properties is encoded by an element. Thus, two interface elements could be syntactically similar if they use a similar/common vocabulary. The vocabulary of an element includes names of port types, operations, parameters, etc. This similarity could be interesting to consider when searching for correspondences between the two initial and expected interfaces. We are using two measures to approximate the syntactic homogeneity between metamodel elements: (1) cosine similarity [100] and (2) the Normalized Levenshtein Edit Distance [101].

4.2.5 **Change Operators**

Several change operators are used as part of the adapted GA.

1) **Selection**.

To select the individuals that will undergo the crossover and mutation operators, we used the stochastic universal sampling (SUS), in which the probability of selection of an individual is directly proportional to its relative fitness in the population. SUS is a random selection algorithm which gives higher probability to be selected to the fittest solutions while still giving a chance to every solution. For each iteration, we use SUS to select individuals (population_size/2) from population P_n for the next population P_{n+1}. These selected individuals (upper half of the ranking) will "give birth" to new individuals (substituting the lower half of the ranking) using the crossover operator.

2) **Crossover**.

When two parent individuals are selected, a random cut point is determined to split them into two sub-vectors. The crossover operator selects a random cut-point in the interval [0, min_{length}] where min_{length} is the minimum length between the two parent chromosomes. Then, crossover swaps the sub-vectors from one parent to the other. Thus, each child combines information from both parents. This operator must enforce the length limit constraint by eliminating randomly some refactoring operations.

For each crossover, two individuals are selected by applying the SUS selection. Even though individuals are selected, the crossover happens only with a certain probability. The crossover operator allows creating two offspring P_1' and P_2' from the two selected parents P_1 and P_2. It is defined as follows. A random position *k* is selected. The first k refactorings of P_1 become the first k elements of P_1'. Similarly, the first *k* refactorings of P_2 become the first *k* refactorings of P_2'.

3) **Mutation**.

The mutation operator consists of randomly changing one or more dimensions (refactoring) in the solution (vector). Given a selected individual, the mutation operator first randomly selects some positions in the vector representation of the individual. Then the selected dimensions are replaced by other refactoring. Furthermore, the mutation can only modify the controlling elements of some positions without replacing the refactoring by a new one.

When applying the mutation and crossover, we used also a repair operator to delete duplicated refactorings after applying the crossover and mutation operators.

4.3 **Validation**

1) Experimental Setup

In order to evaluate the ability of our changes detection framework to efficiently detect the refactorings applied between multiple Web service releases, we conducted a set of experiments based on six widely used Web services. In this section, we first present our research questions, the experiments setup and then describe and discuss the obtained results.

We defined the following two research questions that address the applicability and performance of our Web services changes detection approach. The two research questions are as follows:

- RQ1: To what extent can our approach detect correctly the composite changes applied between multiple releases of Web services?
- RQ2: How does our approach perform comparing to techniques just based on either structural or textual similarities?

To answer these two research questions, the quality of the results was measured by two methods: automatic correctness (AC) and manual correctness (MC). Automatic correctness consist of comparing the detected changes to the reference ones, operation by operation using precision (AC-P) and recall (AC-R). AC method has the advantage of being automatic and objective. However, since different refactoring combinations exist that describe the same evolution (different changes but same target Web services interface), AC could reject a good solution because it yields different refactoring operations from reference ones. To account for those situations, we also use MC which manually evaluates the detected changes, here again operation by operation. The precision corresponds to the number of correct refactorings divided by the total number of

generated refactorings. The recall is the number of correct refactorings divided by the number of expected ones.

To answer RQ2, we compared our approach that combines both structural and textual measures into one fitness function to two different techniques. The first technique, ST, is based also on a GA algorithm but using only the structural measures. The second technique, TE, used a GA algorithm but only based on cosine similarity and edit distance measures.

We selected these 6 Web services for our validation because different releases of their WSDL interface are publicly available and belong to different categories. Table 10 provides some descriptive statistics about these six Web services:

- Amazon EC2: Amazon Elastic Compute Cloud is a web service that offers resizable compute capacity in the cloud. In this study we have considered a total of 44 releases from 2006 until 2014.
- Amazon Simple Queue Service (Amazon SQS) offers reliable hosted queues for storing messages exchanged between computers. We considered in our study a total of 6 releases.
- Fedex Track service offers accurate update of the status of shipments. We used 10 releases from this Web service.
- FedEx Ship Service: Ship Service provides functionalities for managing package shipments and their options. A total of 17 releases are considered in our experiments from this Web service.
- FedEx Rate Service: the Rate Service provides the shipping rate quote for a specific service combination depending on the origin and destination information supplied in the request. We used 18 releases for our prediction algorithm.

- Amazon Mechanical Turk Requester: it is a web service that provides an on-demand, scalable, human workforce to complete jobs that humans can do better than computers such as recognizing objects in photos. We used 15 releases developed between 2005 until 2012.

Table 10 Web service statistics

Web Service Name	# Releases	Average number of changes per release
Amazon EC2	44	13
Amazon Mechanical Turk	15	11
Amazon Simple Queue	6	9
FedEx Rate Service	18	16
FedEx Ship Service	17	21
FedEx Track Service	10	14

Since metaheuristic algorithms are stochastic optimizers, they can provide different results for the same problem instance from one run to another. For this reason, our experimental study is performed based on 30 independent simulation runs for each problem instance and the obtained results are statistically analyzed by using the Friedman test with a 99% confidence level ($\alpha = 1\%$). In fact, we computed the p-value of the ST and TE results with GA ones. In this way, we could decide whether the superior performance of GA to one of each of the others (or the opposite) is statistically significant or just a random result.

2) Experiment Results

Results for RQ1. Figure 26 summarizes the outcome for the first research question. Most of the Web service changes were detected accurately on the different Web services with an average precision higher than 86% as described in Figure 26. For Fedex Track service and Fedex Rate service, the precision is the highest with more than 88%. This could be related to the lower number of changes to detect comparing to other services. For Amazon EC2, the precision is also high with more than 86% even that this service has a higher number of changes to detect comparing to several

other Web services. This confirms that our detection results are independent from the number of changes to detect.

The same observations are valid for the recall. Fedex Track Service and Amazon Mechanical Turk have the highest recall with more than 90% but still they have a good precision higher than 83%. This may confirm that the precision and recall were both acceptable for the different services. Overall, the recall results were better than the precision. This could be explained by the fact that our optimization algorithm is based on the use of heuristics (fitness functions) to estimate the similarity between the generate interface and the expected one of the new release. Thus, some flexibility is introduced based on this estimation of structural and textual similarities which may explain the lower precision comparing to the recall on the different Web services.

Figure 26 also confirms that the manual correctness MC of the detected changes is the highest on all the Web services comparing to the automatic correctness based on both precision and recall. This could be explained by the fact that there are multiple ways to describe the changes between the different Web service releases.

To answer the first research question, our approach is able to detect the changes during the evolution of Web service with a high accuracy.

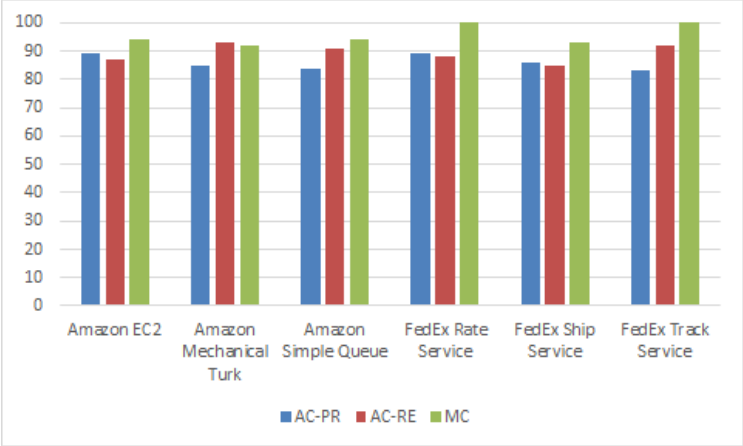


Figure 26 Median precision, recall and manual correctness of detected refactorings by our GA approach based on 30 independent runs.

Results for RQ2. Based the AC and MC measures, Figure 27 show that the solutions provided by GA have the highest manual and automatic correctness values on all the Web services comparing to the two other approaches that just use either structural (ST) and textual (TE) similarities. In fact, the average AC value for GA is 88% and it is lower than 80% for all the remaining algorithms on all the Web services. The same observation is valid for MC, GA has the highest MC average value with 90% while the remaining algorithms their MC average is lower than 82%. Based on these results, it is also interesting to note that there is no correlation between the number of refactorings to detect and the correctness values. More precisely, we sort AC and MC of the different approaches based on the number of refactorings for each Web service. From this data, we conclude that AC and MC are not necessarily affected negatively by a larger number of refactorings to detect. Thus, we can conclude that our proposal shows a good scalability and is not affected negatively by the number of refactorings.

Overall the obtained results of the TE approach is better than the ST ones based on the different measures. This may confirm the importance of considering semantics similarity when

detecting changes between Web services. The results confirm that both textual and structural similarities are complementary based on the outperformance of our tehcniques.

In conclusion, we answer RQ2, the results support the claim that our GA formulation provides the best compromise between the structural and textual similarities when detecting changes between multiple Web service releases.

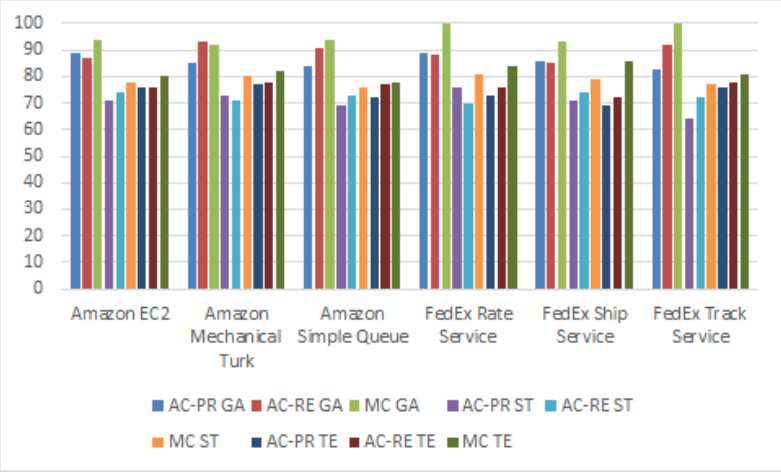


Figure 27 Comparison between the median precision, recall and manual correctness of detected refactorings by the different approaches based on 30 independent runs.

3) Threats to Validity

Conclusion validity is concerned with the statistical relationship between the treatment and the outcome. We used the Wilcoxon rank sum test on 30 runs with a 99% ($\alpha < 0.05$) confidence level to test if significant differences existed between the measurements for different treatments. This test makes no assumption that the data is normally distributed and is suitable for ordinal data, so we can be confident that the statistical relationships we observed are significant.

Internal validity is concerned with the causal relationship between the treatment and the outcome. The parameter tuning of our genetic algorithm is important. In fact, different results can be obtained with different parameter settings such number of iterations, stopping criteria, etc. We

need to evaluate in our future work the impact of different parameters on the quality of the results (parameters sensitivity) in terms of precision, recall and manual correctness.

Construct validity is concerned with the relationship between theory and what is observed. Most of what we measure in our experiments are standard metrics such as precision, recall, etc. that are widely accepted as good proxies for quality evaluation. The metrics used for structural and syntactic similarities can be extended using additional ones such as number of detected refactorings. Additional experiments are required in future work to evaluate the impact of number of used metrics on the quality of the results. Another limitation is related to the use of Wordnet to find synonyms of the name of service elements such as operations which is not usually feasible due to the limited words/vocabulary considered in Wordnet. We will investigate in our future work the adaptation of other dictionaries on the quality of our results.

External validity refers to the generalizability of our findings. In this study, we performed our experiments on different widely used Web services belonging to different domains and with different sizes in terms of expected changes and number of operations. However, we cannot assert that our results can be generalized to other services. Future replications of this study are necessary to confirm our findings.

4.4 **Conclusion**

In this work, we proposed an approach to detect changes during the evolution of Web services. Our approach, based on genetic algorithms, takes as input an exhaustive list of possible change types, the initial release and the revised one, and generates as output a list of detected changes in terms of composite changes, and not atomic ones. A solution is defined as the combination of refactoring operations that should maximize the structural and textual similarity between the expected new Web service interface release and the generated one after applying the refactoring

sequence on the initial release. We evaluated our approach on a set of 6 popular Web services including more than 110 releases. We reported the results on the efficiency and effectiveness of our approach to detect changes of the evolution of Web services interfaces in terms of precision and recall.

Chapter 5 **Prediction of Software and Service Defects**

5.1 **On the Use of Time Series for Software Refactoring Recommendation**

5.1.1 **Introduction**

During software maintenance and evolution, software systems undergo continuous changes, through which new features are added, bugs are fixed, and business processes are adapted constantly. However, this may in turn introduce poor design effects and make systems more complex. This complexity leads to significantly reduced productivity, decreased system's performance, increased fault-proneness, more costly software and even canceled projects. Many studies reported that software engineers spend around 60% of their time in understanding the code and that software maintenance activities consume up to 90% of the total cost of a typical software project.

This high cost could potentially be greatly reduced by providing automatic or semi-automatic solutions to increase their understandability, adaptability, and extensibility, to avoid and fix bad-design practices. A widely used technique to improve the overall quality of systems is refactoring which improves design structure while preserving the overall functionalities and behavior .

A variety of refactoring work has been proposed in the literature [27], [62], [64], [65], [67], [102], [103]. In general, refactoring is performed through two main steps: 1) detection of code fragments that need to be improved (e.g., code-smells) and 2) identification of refactoring solutions to achieve this goal. The first step is well covered in the literature, and there exists a growing

number of techniques to identify code-smells [27]. Once detected, not all code-smells have equal effects and importance. In general, developers need to start by fixing the higher risk code-smells. However, in the literature, the majority of existing contributions proposes manual or semi-automated refactoring solutions that can be applied to fix particular types of code-smells (e.g. blobs, spaghetti code, etc.) or to improve some quality metrics (e.g., cohesion, coupling, etc.) without taking into consideration the importance/risk of the code-smells to fix. Furthermore, some code smells could become more and more risky if they are not fixed as early as possible. For example, a blob defect (a large class with high number of responsibilities) is difficult to fix if it was not detected early. However, existing refactoring studies did not consider the impact of refactoring solutions on future releases of the system when not all the detected code smells were fixed.

In this work, we introduce a novel approach to support automated refactoring suggestions for correcting not only existing code smells but also the code fragments that may contain quality issues in the next releases. Hence, we formulated the refactoring suggestion problem as a combinatorial optimization problem to find the near-optimal sequence of refactorings from a large number of possible refactorings.To this end, we propose to combine the use of search-based software engineering with time series [104] to recommend good refactoring strategies in order to manage technical debt. We used a multi-objective algorithm, based on NSGA-II [40], to generate refactoring solutions that maximize the correction of important and riskiest quality issues, and minimize the effort. For these two fitness functions, we adapted time series forecasting to estimate the impact of the generated refactorings solution on future releases of the system by predicting the evolution of the remaining code smells in the system, after refactoring, using different quality metrics. We evaluated our approach on one industrial project and a benchmark of 4 open source

systems. The results confirm the efficiency of our technique to provide better refactoring management compared to several existing refactoring techniques [54], [65], [103], [105].

5.1.2 **Time Series Forecasting**

A time series [104] is a sequence of data points that are typically measured at successive equally-spaced time instants. Examples of time series are global temperature, ocean tides, daily closing value, etc. A time series is either used for analysis or forecasting. Analysis means extracting meaningful statistics and characteristics of the data; while forecasting means building a model to predict future values based on previously observed values. In this work, we are interested in forecasting. A time series model is a stochastic process that can have different models. Some of the most used ones are the AR (Auto-Regressive) model, the MA (Moving-Average) model, and the ARMA (Auto-Regressive Moving-Average) one. The hybridization of these three models has yielded the ARIMA (Auto-Regressive Integrated Moving Average) model, which is one of the most used models in the literature [106]. Motivated by the interesting results of ARIMA in previous Software engineering applications [107]–[110], we chose to use it in this work. [107]–[110], we choose to use it in this work.

By definition, the ARIMA (p, d, q) model consists of a combination of AR(p), MA(q), and ARMA(p, q) where p is the order of the autoregressive component, d is the order of the differenced component, and q is the order of the moving average component. An autoregressive model of order p views the present value of the series as the linear regression of the previous p values, whereas a moving average model of order q is conceptually a line regression of the current value of the series against previous white noise error terms. The ARMA (p, q) model is obtained by combining AR and MA. If a time series is not stationary, this time series need to be differenced before applying ARMA (p, q).

The ARIMA modelling strategy usually follows four steps as described by Figure 28. These steps are the following:

1) Identification

This step consists in plotting the time series and some related measures such as the mean, the range, the ACF (Auto-Correlation Factor), and the PACF (Partial Auto-Correlation Factor). The different plots allow understanding the nature of the series changes over time. The ACF represents the correlation, at specific lags, between the residuals of the data. If the lag terms persist in the ACF plots, it indicates inertia in the series. Such series can be differenced to remove this effect. While identifying non-stationary series, the ACF and PACF of the difference are studied, to ensure that the persistence of ACF due to non-stationary nature does not lead to incorrect identification. A non-stationary series can be converted to a stationary series by differencing (taking the difference ($Y_t - Y_{t\text{-}1}$)).

2) Estimation

The ARIMA model is mathematically stated by the following equation:

$$\Phi_p(B)\Delta^d x_t = \Theta_q(B)\,\varepsilon_t$$

where B is the backward shift operator, $Bx_y = x_{y-1}$, $\Delta = 1 - B$ is the backward difference, ε_t is a white noise, and Φ_p and Θ_q are polynomials of order p and q respectively. We can observe that the model is composed with different parts: (1) an auto-regressive ($AR(p)$) part $\Phi_p = 1 - \phi_1 B^1 - \ldots - \phi_p B^p$, an integrating part $I(d) = \Delta^{-d}$, and a moving average ($MA(q)$) part $\Theta_q = 1 - \theta_1 B^1 - \ldots - \theta_q B^q$. The estimation step consists in estimating the parameters of equation (1) (i.e, Φ_p and Θ_q). The most used estimation methods are the least square method and the maximum likelihood one.

3) Diagnostic checking

This step consists in assessing the goodness of the model in fitting the data. If the model well fits the data, the residuals of the model behave as an independent identically distributed sequence with a mean of zero and a variance of one. That is, the residual sequence should correspond to a white noise; otherwise, the model needs to be improved. The χ^2 testing on the residual sequence is commonly used in this step.

4) Application

Once the ARIMA (p, d, q) model is built, we can use this model to make a prediction of a future quantity or to explain actual data trends. In this work, we are interested in prediction (i.e., forecasting). Generally, there are two ways to make forecasting: (1) static forecasting and (2) dynamic forecasting. Static forecasting computes a sequence of one-step forecasting values using the actual values rather than the forecasted ones for lagged dependent variables. Dynamic forecasting calculates multi-step forecast values at once. Previously forecasted values for the lagged dependent variable are used in forming forecasts of the current value. In this work, we used the static forecasting. For our problem, the Time Series prediction process consists of three main steps: the metrics extraction, the model definition and prediction of the future value of the metrics. In our case, the features represent the quality metrics values of the system after refactoring and the output of the model is the new values of the metrics in the new release if the given refactoring solution was applied. Of course, the quality of our prediction depends mainly on the size of the training set which is, in our case, the previous releases of a system. To this end, we selected in our experiments systems containing an extensive set of previous releases.

In the next sub-section, we describe the adaption of NSGA-II to our problem and how the Time Series algorithm was combined with that multi-objective algorithm.

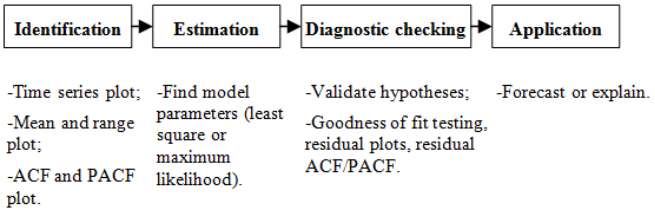


Figure 28 ARIMA steps: Box-Jenkins methodology

5.1.3 **Approach Overview**

The goal of our approach is to generate the best refactoring sequence that improves the quality of the design and minimize the effort. Therefore, we use a multi-objective optimization algorithm to compute an optimal sequence of refactorings in terms of finding trade-offs between these two objectives. The first objective represents the main novelty of this work since we consider the prediction of code smells evolution when evaluating refactoring solutions. Thus, the main goal is to reduce not only the number of existing code smells but also to fix risky code fragments that can become severe code smells to fix in future releases. The second objective minimizes the number of code smells to fix before the next release to reduce the effort. The general structure of our approach is sketched in Figure 29.

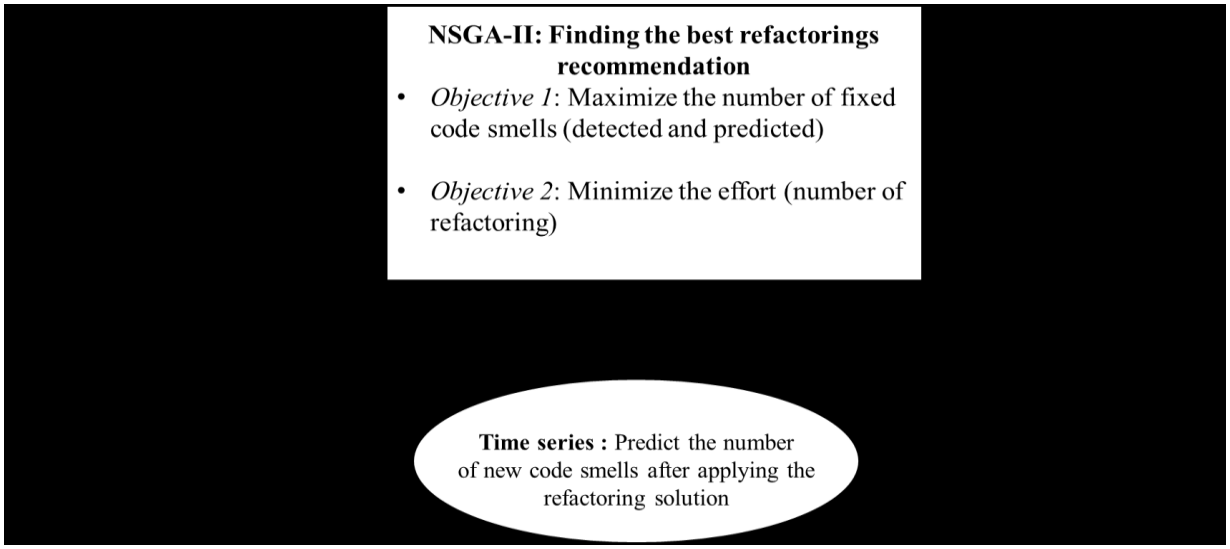


Figure 29 Multi-objective model refactoring: overview

The search-based process takes as inputs the list of 23 possible types of refactoring, the code smells detection rules from the previous work of Kessentini et al [22], a list of metrics to predict the evolution of code smells after refactoring, and the system to refactor. The process of generating a solution can be viewed as the mechanism that finds the best refactorings sequence among all possible solutions that optimize the above two conflicting objectives. The size of the search space is determined not only by the number of refactoring but also by the order in which they are applied. Due to the large number of possible refactoring combinations and the two conflicting objectives to optimize, we considered the refactoring problem as a multi-objective problem. In order to predict the evolution of the quality metrics after refactoring, we used a machine learning algorithm based on Time Series, taking as input the code smells detection rules, the system after applying the refactoring solution and a set of training data composed of the metrics values of the previous releases. It generates as output the predicted metrics value for the next release after refactorings along with the number of code smells that will be created in the future releases based on the prediction.

5.1.4 **NSGA-II Adaptation**

1) Solution representation

To represent a candidate solution (individual), we used a vector representation. Each vector's dimension represents a refactoring operation. Thus, a solution is defined as a long sequence of refactorings applied to different parts of the system. When created, the order of applying these refactorings corresponds to their positions in the vector. In addition, for each refactoring, a set of controlling parameters are stored in the vector, e.g., actors and roles are randomly picked from the class diagram to be refactored and stored in the same vector. An example of a solution is given in Figure 30.

move method (*Person, Employee, calculateExperience(int)*)
move field (*Employee, Company, salary*)
extract class (*FinanceDept, Company, id, numberEmployee (int,)*)

Figure 30 Representation of an NSGA-II individual

After the generation of the refactoring solutions, it is important to guarantee that they are feasible and that they can be legally applied. For example, to apply the refactoring operation *move method*, a number of necessary preconditions should be satisfied, e.g., *Person* and *Employee* should exist and should be classes; *calculateExperience(int)* should exist and should be a method; the classes *Person* and *Employee* should not be in the same inheritance hierarchy; the method *calculateExperience(int)* should be implemented in *Person*; the method signature of *calculateExperience(int)* should not be present in class *Employee*. As postconditions, *Person, Employee* and *calculateExperience(int)* should exist; *calculateExperience(int)* declaration should be in the class *Employee*; and *calculateExperience(int)* declaration should not exists in the class *Person*.

2) Fitness functions

After creating a solution, it should be evaluated using fitness functions. Since we have two objectives to optimize, we are using two different fitness functions to include in our NSGA-II adaptation. We used the following fitness functions: *Number of detected and predicted code smells after applying the refactoring solution*. More formally, this fitness function is composed as following: $NCS = dCS + pCS$, where *dCS* represents the number of detected code smells after applying the refactoring solution on the system. *pCS* is the number of predicted code smells (on the system after refactoring) in the next release using Time Series. The code smells are detected using the rules defined by Kessentini et al. [22]: *Effort E (number of changes to introduce in the system): Number of refactorings composing the solution (size).*

3) Genetic operators

To better explore the search space, the crossover and mutation operators are defined. For crossover, we use a single, random, cut-point crossover. It starts by selecting and splitting at random two parent solutions. Then crossover creates two child solutions by putting, for the first child, the first part of the first parent with the second part of the second parent, and, for the second child, the first part of the second parent with the second part of the first parent. This operator must ensure that the length limits are respected by eliminating randomly some refactoring operations. Each child combines some of the refactoring operations of the first parent with some ones of the second parent. In any given generation, each solution will be the parent in at most one crossover operation.

The mutation operator picks randomly one or more operations from a sequence and replaces them by other ones from the initial list of possible refactorings. After applying genetic operators (mutation and crossover), we verify the feasibility of the generated sequence of refactoring by checking the pre and post conditions. Each refactoring operation that is not feasible due to

unsatisfied preconditions will be removed from the generated refactoring sequence. The new sequence is considered valid in our NSGA-II adaptation if the number of rejected refactorings is less than 5% of the total sequence size.

5.1.5 **Validation**

1) Experimental Setup

Our study aims at addressing the two research questions outlined below.

- RQ1: (*Usefulness*) To what extent can the proposed approach improve the quality?
- RQ2: (*Comparison to state-of-the-art*) To what extent can the proposed approach improves the results of refactoring suggestion using the prediction of code smells component compared to existing work that do not use it [22], [65], [68]?

To answer RQ1, we validate the proposed refactoring solutions to improve the quality of the system by evaluating their ability to improve the different maintainability objectives defined by the QMOOD model [111] related to reusability, flexibility, understandability, functionality, extendibility, and effectiveness. The improvement in quality can be assessed by comparing the quality before and after refactoring independently to the number of fixed design defects.

We have also asked a group of five software engineers (graduate students in Software Engineering) to manually evaluate the best refactoring solutions. To this end we define the following precision metric *MP* (Manual precision) defined as the number of good refactorings divided by the total number of recommended refactoring:

$$MP = \frac{\#\,good\ \text{refactorings}}{\#\,\text{proposed refactorings}}$$

To answer RQ2, we compared our refactoring results with two existing search-based refactoring techniques and one tool not based on Heuristic search by the mean of JDeodorant [22],

[65], [68]. All these techniques did not consider the prediction of quality issues when recommending refactorings.

Our study considers model fragments extracted from four open source projects and one industrial project: FindBugs, JFreeChart, Hibernate, Pixelitor and JDI-Ford. Table 11 summarizes the statics related to these systems. JDI-Ford is a system provided by our industrial partner the IT department at Ford Motor Company. It is Java-based software system that helps Ford Motor Company analyze useful information from the past sales of dealerships data and suggests which vehicles to order for their dealer inventories in the future. JDI is a highly structured and several versions were proposed by software engineers at Ford during the past 10 years. We selected these systems for our validation because they range from medium to large-sized open source projects that have been actively developed over the past 10 years, and include a large number of design defects and previous releases. In addition, these open source systems were analyzed by previous work [22], [54], [102], [105].

Parameter setting has a significant influence on the performance of a search algorithm on a particular problem instance. For this reason, for each algorithm and for each system, we perform a set of experiments using several population sizes: 50, 100, 200, 300 and 500. The stopping criterion was set to 100000 evaluations for all algorithms in order to ensure fairness of comparison. The other parameters' values were fixed by trial and error and are as follows: (1) crossover probability = 0.8; mutation probability = 0.5 where the probability of gene modification is 0.3; stopping criterion = 100000 evaluations.

Since metaheuristic algorithms are stochastic optimizers, they can provide different results for the same problem instance from one run to another. For this reason, our experimental study is performed based on 31 independent simulation runs for each problem instance and the obtained

results are statistically analyzed by using the Wilcoxon rank sum test [112] with a 99% confidence level ($\alpha = 1\%$).

Table 11 Systems studied

Systems	Releases	#Classes	#Smells (last release)
JFreeChart	From v0.9.6 to v1.0.13	960	84
FindBugs	From v1.2.1 to v 3.0.1	1907	118
Hibernate	From v4.0.0 to v4.2.18	1,004	124
Pixelitor	From v0.1 to v1.1.2	564	73
JDI-Ford	From v2.1 to v5.8	638	88

We note that the mono-objective algorithm provides only one refactorings solution, while NSGA-II generates a set of non-dominated solutions. In order to make meaningful comparisons, we select the best solution for NSGA-II using a knee point strategy. The knee point corresponds to the solution with the maximal trade-off between the two objectives. We use the trade-off "worth" metric proposed by Rachmawati and Srinivasan to find the knee point. This metric estimates the worthiness of each non-dominated merging solution in terms of trade-off between our conflicting objectives. After that, the knee point corresponds to the solution having the maximal trade-off "worthiness" value.

2) Experiment Results

Results for RQ1. As described in Figure 31, after applying the proposed refactoring operations by our approach (NSGA-II), we found that, in average, all the quality attributes of QMOOD were improved on all the five studied systems. It is clear that the understandability and extendibility attributes were improving better than all other quality attributes. This can be explained by the nature of code smells that were fixed. Moreover, to ensure the efficiency and usefulness of our approach, we verified manually the feasibility of the different proposed refactoring sequences for each system. We applied the proposed refactorings using Eclipse. Some

semantic errors (programs behavior) were found. When a semantic error is found manually, we consider the operations related to this change as a bad recommendation. We calculate a correctness precision score *MP* (ratio of possible refactoring operations over the number of proposed refactoring) as one of the performance indicators of our algorithm. Figure 32 shows also that an average of more than 80% of refactorings are feasible confirming the correctness of the recommended refactorings.

We have evaluated the ability of our approach to reduce the number of refactorings to apply while maximising the quality. Figure 33 describes that NSGA-II proposed a reasonable number of refactings to apply, lower than 200, on the different systems. Another interesting observation is that the number of suggested refactorings was correlated with the number of code smells to fix on the different systems.

To sum up, we can conclude based on the results of Figure 31 that our approach succeeded in improving the design quality not only by fixing the majority of detected code smells in the system (as a fitness function to maximize) but also by improving the user understandability, the reusability, the flexibility, as well as the effectiveness of the refactored design.

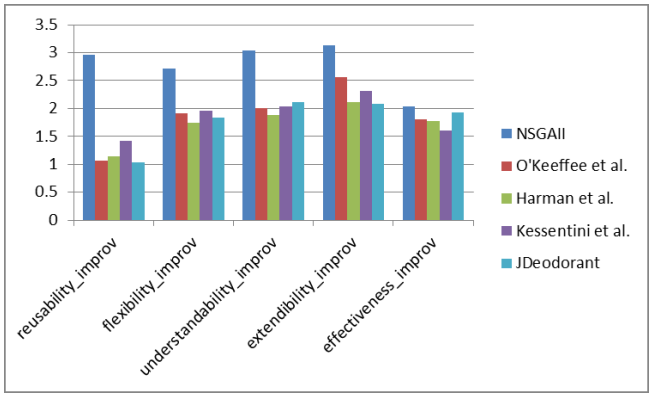


Figure 31 QMOOD quality attributes median values

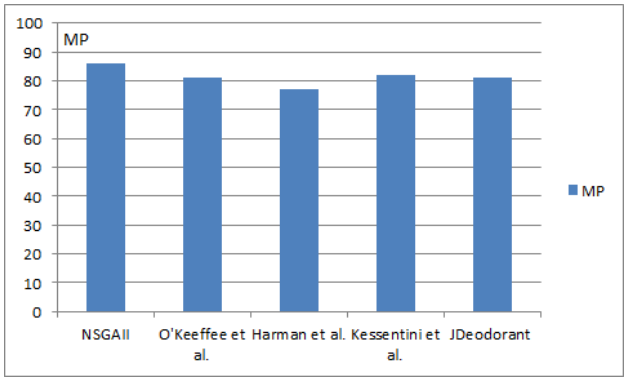


Figure 32 The manual refactoring precision (RP) median values

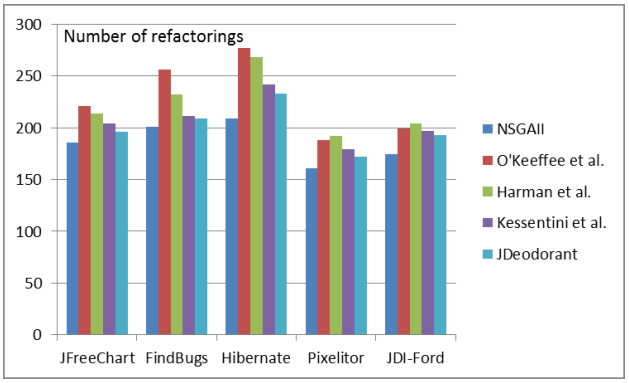


Figure 33 The number of refactorings median values

Results for RQ2. As described in Figure 31, Figure 32, and Figure 33, it is clear that our proposal outperforms the different existing techiques. Figure 31 shows that our approach improves the quality of the design with a better quality attributes value comparing to all existing approaches considered in our experiements. In terms of behavior preservation it is clear that our approach provides much more feasible refactorings than existing approachges for all the systems as described in Figure 32. Figure 33 shows that our proposal requires less effort to apply the best solutions on all the systems than existing approaches. In fact, all the other existing techniques are not considering the minimization of the size of a refactoring solution. We found that an explanation of the outperformance of our technique is that the recommended refactorings by NSGA-II fixed

not only the majority of existing code smells but also several code fragments that contain some early symptomps of code smells that were predicted using the Time Series algorithm.

Usually in the optimization research field, the most time consuming operation is the evaluation step. In fact, all the algorithms under comparison were executed on machines with Intel Xeon 3 GHz processors and 8 GB RAM. Figure 8 illustrates the obtained average CPU times of all algorithms on the systems. We note that the results presented in this figure were analyzed by using the same previously described statistical analysis methodology. In fact, based on the obtained p-values regarding CPU times, the NSGA-II provides a comparable execution time to the remaining techniques as highlighted through Figure 34. This observation could be explained by the fact that a multi-objective algorithm requires, in general, a higher execution time than a mono-objective one. In addition, the use of Time Series to evaluate the refactoring solutions is another reason of the higher execution time. However, we can consider that an average of less than 25 minutes of difference between the execution time of our algorithm and existing work is not an issue especially that refactoring is not a real-time problem. For example, developers can execute our tool overnight.

Figure 34 Average execution time on all the systems

5.1.6 **Conclusion**

This work presented a novel multi-objective refactoring approach taking into consideration multiple criteria to suggest good and feasible refactoring solutions to improve the design quality while reducing the effort. The suggested refactorings preserve the behavior of the design to restructure and consider the impact of fixing code smells in the system using several quality metrics. Our search-based approach succeeded to find the best trade-off between these criteria. Thus, our proposal produces more meaningful refactorings in comparison to some of those discussed in the literature. Moreover, the proposed approach was empirically evaluated on several open-source systems and one industrial project, and compared successfully to an existing approach not based on heuristic search.

5.2 **Prediction of Web Services Defects and Evolution**

5.2.1 **Introduction**

Service-based systems heavily depend on the interface of selected services used to implement specific features. However, service providers do not know, in general, the impact of their changes, during the evolution Web services, on the applications of subscribers. The subscribers are reluctant, in general, to use Web services that are risky and not stable [2]. Thus, analyzing and predicting Web service changes is critical but also challenging because of the distributed and dynamic nature of services. As a consequence, recent studies were proposed to understand the evolution of Web services especially at the interface level [2], [51], [57].

We use, in this contribution, the changes collected from previous Web service releases to address the following problems. Most of the changes in a web service interface typically affect the systems of its subscribers. Thus, it is important for subscribers to estimate the risk of using a specific service and compare its evolution to other services offering the same features in order to

reduce the effort of adapting their applications in the next releases. Subscribers prefer to use, in general, Web services that are stable with a low risk to include bugs and introduce major revisions in the future. In addition, the prediction of interface changes may help web service providers to better manage available resources (e.g. programmers' availability) and efficiently schedule required maintenance activities to improve the quality of developed services. In fact, the prediction of Web service changes can be used to identify potential quality issues that may occur in the future releases. Thus, it is easier to fix these quality issues as early as possible before that they become more complex.

In this work, we propose a machine learning approach based on Artificial Neural Networks (ANN) [43] to predict the evolution of Web services interface from the history of previous releases' metrics. The predicted interface metrics value is used to predict and estimate the risk and the quality of the studied Web services. We evaluated our approach on a set of 6 popular Web services including more than 90 releases. We report the results on the efficiency and effectiveness of our approach to predict the evolution of Web services interfaces and provide useful recommendations for both service providers and subscribers. The results indicate that the prediction results of several Web service metrics, on the different releases of the 6 Web services, were similar to the expected ones with very low deviation rate. Furthermore, most of the quality issues of Web service interfaces were accurately predicted, for the next releases, with an average precision and recall higher than 82%. The survey conducted with a set of developers also shows the relevance of prediction technique for both service providers and subscribers.

5.2.2 **Approach Overview**

As described in Figure 35, our technique takes as input the previous releases of the Web service interfaces to predict its evolution, an exhaustive list of metrics to predict, and a list of

detection rules to detect potential future quality issues, called Web service antipatterns, based on the predicted metrics. Our approach generates as output the set of predicted evolution metrics values and possible future quality issues for the next release.

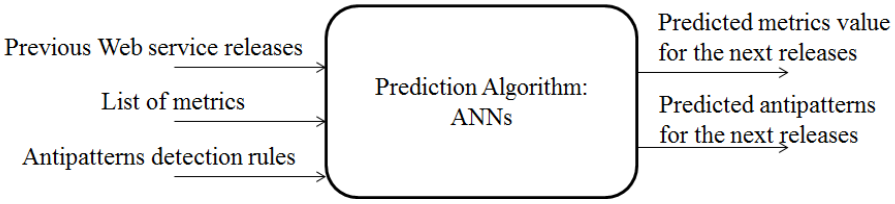


Figure 35 Prediction approach: overview

Our prediction model is based on machine learning algorithm using Aritificial Neural Network (ANN) model. In the following we describe the ANN adaptation to our Web services evolution prediction problem.

5.2.3 **Artificial Neural Network Model**

Artificial Neural Network (*ANN*): ANN models are mathematical models inspired by the functioning of nervous systems [113]–[116], which are composed by a number of interconnected entities, the artificial neurons. ANNs are based on learning which is a characteristic of adaptive systems which are capable of improving their performance on a problem as a function of previous experience [116]. An ANN builds a map between a set of inputs and the corresponding outputs. This model can deal with non-linear regression analysis with noisy signals and incomplete data. In this work, we used a Multi-Layer Perception ANN (MLP-ANN) [113]. It is well-known that MLP-ANNs are universal approximators, which makes them attractive for modeling black-box functions for which little information about their form is known. The output of each neuron is expressed as follows:

$$y = \phi\left(\sum_{i=1}^{n} w_i a_i + b\right)$$

where w denotes the weight vector, a is the input vector, b is the bias, ϕ is the activation function, and n is the number of neurons in the hidden layer. A hidden neuron influences the network outputs only for those inputs that are near to its center, therefore requiring an exponential number of hidden neurons to cover entirely the input space. For this reason, it is suggested that MLP-ANN are suitable for problems with a small number of inputs like our prediction of Web services evolution problem.

5.2.4 **Artificial Neural Network Adaptation**

We applied the ANN as being among the most reliable predictive models, especially, in the case of noisy and incomplete data. Its architecture is chosen to be a multilayered architecture in which all neurons are fully connected; weights of connections have been, randomly, set at the beginning of the training. Regarding the activation function, the sigmoid function is applied [43] as being adequate in the case of continuous data. The network is composed of three layers: the first layer is composed of p input neurons. Each neuron is assigned the value x_{ki}. The hidden layer is composed of a set of hidden neurons. The learning algorithm is an iterative algorithm that allows the training of the network. Its performance is controlled by two parameters. The first parameter is the momentum factor that tries to avoid local minima by stabilizing weights. The second factor is the learning rate which is responsible of the rapidity of the adjustment of weights.

Learning process. Before the learning process, the data used in the training set should be normalized. In our case, we choose to apply the *min-max* technique since it is among the most accurate techniques according to [117]. In our adaptation, we used the following list of metrics from the literature [12] to predict for the next Web service releases, as described in Table 12.

Table 12 Web service interface metrics used

Metric Name	Definition
NPT	Number of port types
NOD	Number of operations declared
NAOD	Number of accessor operations declared
NOPT	Average number of operations in port types
ANIPO	Average number of input parameters in operations
ANOPO	Average number of output parameters in operations
NOM	Number of messages
NBE	number of elements of the schemas
NCT	Number of complex types
NST	Number of primitive types
NBB	Number of bindings
NBS	Number of services
NPM	Number of parts per message
NIPT	Number of identical port types
NIOP	Number of identical operations
COH	Cohesion
COU	Coupling
AMTO	Average meaningful terms in operation names
AMTM	Average meaningful terms in message names
AMTMP	Average meaningful terms in message parts
AMTP	Average meaningful terms in port-type names
ALOS	Average length of operations signature
ALPS	Average length of port-types signature
ALMS	Average length of message signature

During the learning process, our ANN solutions are represented as follows: let us denote by O the matrix that includes numerical values related to the set of metrics to predict. O is composed of n lines and p columns where n is equal to the number of metrics to predict and p is equal to the number of steps (releases).

$$O = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1p} \\ x_{21} & x_{22} & \dots & x_{2p} \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ x_{n1} & x_{n2} & \dots & x_{np} \end{bmatrix}$$

Learning technique. There are several learning algorithms, depending on whether the ANN model is linear or non-linear. Our MLP model utilizes a supervised learning technique called back-propagation (BP) for training the network. MLP is a modification of the standard linear perceptron and can distinguish data that are not linearly separable. BP is one of the most popular and common training procedures used, that is described in depth in the literature [43]. Our BP neural network has been trained with moderate values for the learning rate (α) and momentum (μ). The weights are recalculated every time a training vector is presented to the network. The exit strategy or the termination condition for the network is based on the sum square error until it reaches a certain threshold assigned prior to running the network. Our implementation is based on the Weka[9] framework with it default configuration.

5.2.5 **Validation**

1) Experimental Setup

In order to evaluate the ability of our prediction framework to efficiently predict the evolution trends of Web services, we conducted a set of experiments based on six widely used Web services. In this section, we first present our research questions, the experiments setup and then describe and discuss the obtained results. Finally, we discuss some threats related to our experiments.

We defined the following three research questions that address the applicability, performance, and the usefulness of our Web services prediction approach. The three research questions are as follows:

- RQ1: To what extent can our approach predict correctly the evolution of Web services?
- RQ2: To what extent can our approach predict Web service quality issues?

[9] http://www.cs.waikato.ac.nz/ml/weka

- RQ3: Can our prediction results be useful for developers?

To answer RQ1, we calculated the deviation between the actual expected metrics value and the predicted ones using our ANNs algorithm on different Web service releases. To this end, we considered the list of metrics described in the previous section. The error rate is defined as follows:

$$e\_rate(M_i, S) = \left| PM_i - EM_i \right|,$$

where *PM* is the predicted metric value using ANNs and *EM* is the expected value. We calculated the error rate for one and many steps (releases) over time for every of the considered Web services. To answer RQ2, we calculated precision and recall scores to compare between the predicted Web services antipatterns and the expected ones:

$$\mathrm{RC}_{recall} = \frac{\text{predicted antipatterns} \cap \text{expected antipatterns}}{\text{expected antipatterns}} \in [0,1]$$

$$\mathrm{PR}_{\text{precision}} = \frac{\text{predicted antipatterns} \cap \text{expected antipatterns}}{\text{predicted antipatterns}} \in [0,1]$$

We considered five types of antipatterns from the literature [11]: Multi-service (MS: a service implementing many operations), Nano-service (NS: too-fine grained service), Chatty-service (CS: a service including many fine-grained operations), Data-service (DS: a service including only data access operations) and Ambiguous service (AS: a service including ambiguous names of operations). More details about existing Web service antipatterns can be found in the following references [11]. We used the manually defined rules in [12] to detect the predicted and actual Web service antipatterns.

To answer RQ3, we used a post-study questionnaire that collects the opinions of developers on our prediction results. We also wished to assess how these results may help developers working on services-based applications. To this end, we asked 24 software developers, including 11 developers working in a Web development startup and providing some Web services for customers

from the automotive industry sector. The remaining participants are 13 graduate students (8 MSc and 5 PhD students) in Software Engineering at the University of Michigan-Dearborn. 9 out the 13 students are working either full-time or part-time programmers in Software industry. All the participants are volunteers and have a minimum of 2 years' experience as a developer. The participants were first asked to fill out a pre-study questionnaire containing five questions. The questionnaire helped to collect background information such as their role within the company, their programming experience, their familiarity with Web services and service-based applications. In addition, all the participants attended one lecture about Web service antipatterns and passed five tests to evaluate their performance to evaluate the design of Web services using quality metrics.

We selected these 6 Web services for our validation because different releases of their WSDL interface are publicly available and belong to different categories. Table 13 Web service statistics provides some descriptive statistics about these six Web services:

- Amazon EC2: Amazon Elastic Compute Cloud is a web service that offers resizable compute capacity in the cloud. In this study we have considered a total of 44 releases from 2006 until 2014.
- Amazon Simple Queue Service (Amazon SQS) offers reliable hosted queues for storing messages exchanged between computers. We considered in our study a total of 6 releases.
- Fedex Track service offers accurate update of the status of shipments. We used 10 releases from this Web service.
- FedEx Ship Service: The Ship Service provides functionalities for managing package shipments and their options. A total of 17 releases are considered in our experiments from this Web service.

- FedEx Rate Service: The Rate Service provides the shipping rate quote for a specific service combination depending on the origin and destination information supplied in the request. We used 18 releases for our prediction algorithm.
- Amazon Mechanical Turk Requester: it is a web service that provides an on-demand, scalable, human workforce to complete jobs that humans can do better than computers such as recognizing objects in photos. We used 15 releases developed between 2005 until 2012.

Table 13 Web service statistics

Web Service Name	# Releases	Average number of Antipatterns
Amazon EC2	44	134
Amazon Mechanical Turk	15	61
Amazon Simple Queue	6	21
FedEx Rate Service	18	17
FedEx Ship Service	17	82
FedEx Track Service	10	44

2) Experiment Results

Results for RQ1. Figure 36,

Figure 37 and Figure 38 summarize the outcome for the first research question. Most of the Web service metrics were predicted accurately on the different Web services with an average error rate lower than 2.8 as described in

Figure 37. For FedEx Track service and FedEx Rate service, the average error rate is the highest. This could be related to the lower training set comparing to the other services. For Amazon EC2, the metrics were predicted with a minimum deviation score of 2.1 due to the large training set available for this service. However, Amazon Simple Queue has one of the lowest deviation score of 1.8. This confirms that our prediction results are independent from the size of the Web services to evaluate and the training data.

Figure 37 shows more detailed results of the average error rate by metric. The results clearly support the claim that our results are independent from the type of metric to predict. However, the error rate depends on the range of every metric. For example, it is expected that the number of operations per service may get the highest error rate since the variation of this metric is high and its range is larger than the other metrics.

Figure 38 describes the ability of our algorithm to predict the metrics value not only for the next release but for up-to the next 5 releases. In fact, the obtained results on the different Web services (except Amazon Simple Queue, not considered due to the limited number of releases) clearly show that the error rate for the 5th upcoming release is minimal with a score less than 4.5. To answer the first research question, our approach is able to predict the evolution of Web service metrics with a high accuracy.

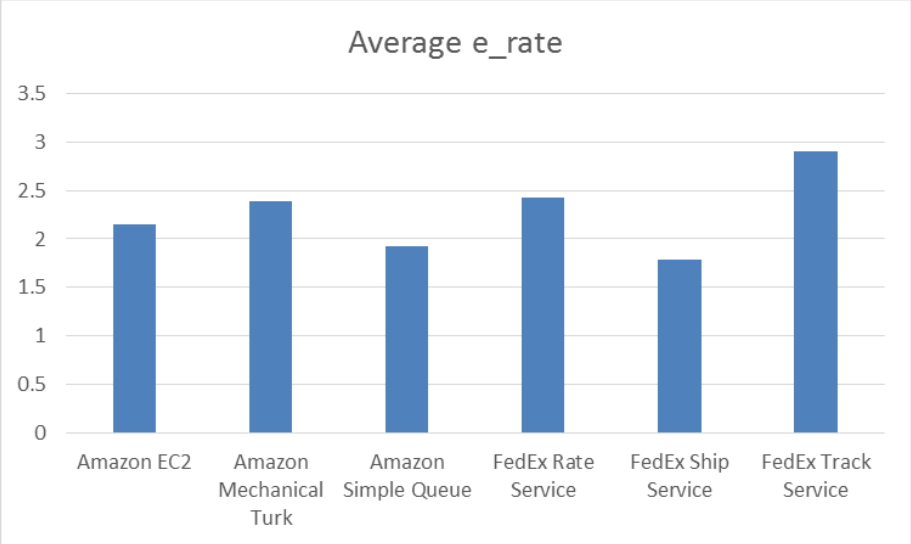


Figure 36 Average error rate (e_rate) on the different Web services

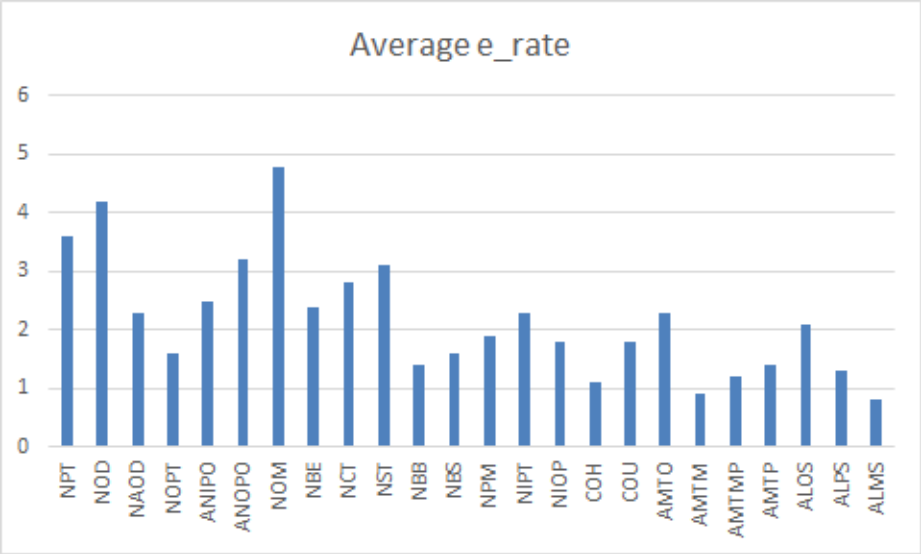


Figure 37 Average error rate (e_rate) per metric on the different Web services

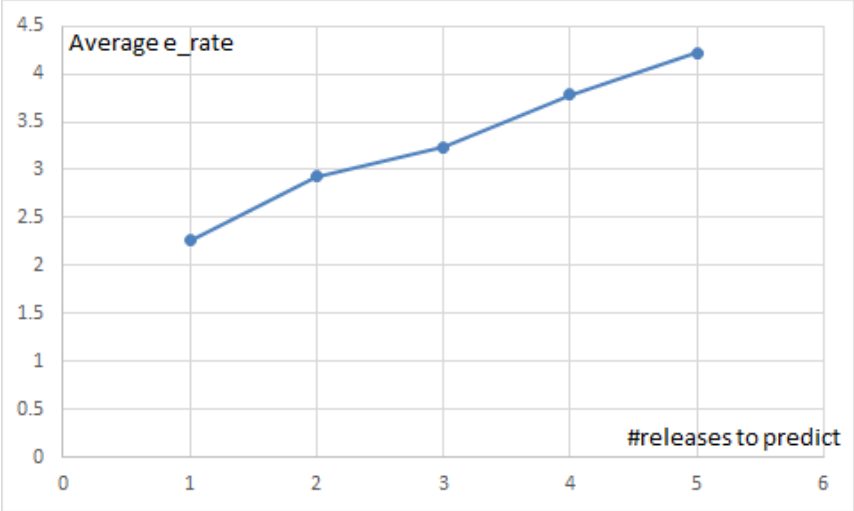


Figure 38 Average error rate (e_rate) of the different metrics on the Web services

(except Amazon Simple Queue) per prediction step

Results for RQ2. Figure 39, Figure 40 and Figure 41 summarize our findings. Overall, most of the expected quality issues (Web service antipatterns) for the next release were identified as described in Figure 39. Our prediction algorithm was able to detect Web service antipatterns on the different services with an average precision and recall respectively higher than 84% and 86%. For FedEx Ship service and Amazon Mechanical Turk, the precision is higher than for the other

systems with more than 88%. This can be explained by the fact that these systems are smaller than others and contain a lower number of antipatterns to predict. For FedEx Rate Service, the precision is also high (around 82%), *i.e.*, most of the predicted antipatterns are correct. This confirms that our precision results are independent from the size of the Web services to evaluate. For Amazon EC2, the precision is one of the lowest (81%) but still acceptable. Amazon EC2 contains a high number of ambiguous services that are difficult to detect using metrics.

The same observations are valid for the recall. The average recall on the six Web services was higher than 86%. For Fedex Track service and Amazon EC2, the precision is higher than for the other systems with more than 90%. This can be explained by the fact that these systems are using more training data than others. For FedEx Ship Service, the precision is also high (around 81%), thus the impact of the size of the training data was not high on the quality of the prediction results. An interesting observation is that the obtained precision and recall scores are conflicting since the services with the highest precision scores received the lowest recall. However, both scores are acceptable for all the Web services.

One key strength of our technique is the ability to predict quality issues not only for the next release but for up-to the next 5 releases as described in Figure 41. In fact, the obtained results clearly show that both precision and recall are still high for all the Web services when predicting quality issues for the 5^{th} upcoming release with an average higher than 73%. We did not consider in our evaluation the Amazon Simple Queue due to the limited number of available releases.

To summarize, it is clear based on the obtained results that our approach predicts Web service quality issues with a high accuracy.

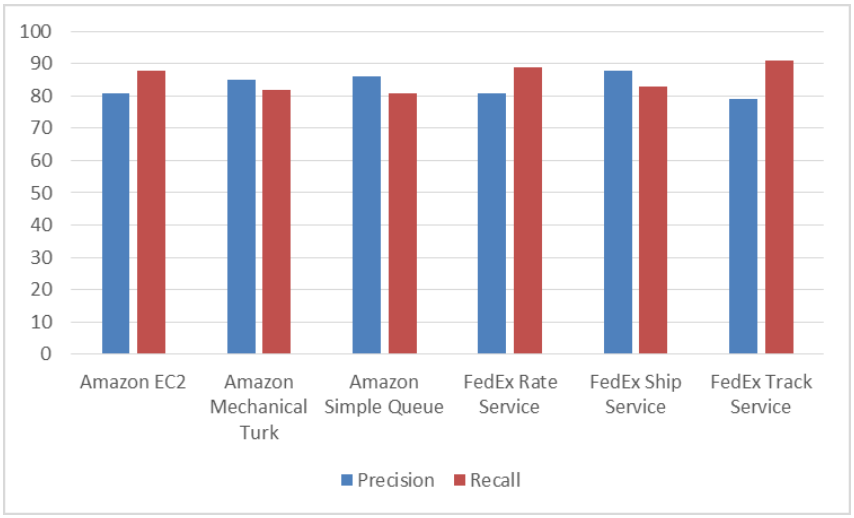


Figure 39 Average precision and recall of the predicted antipatterns on the different Web services

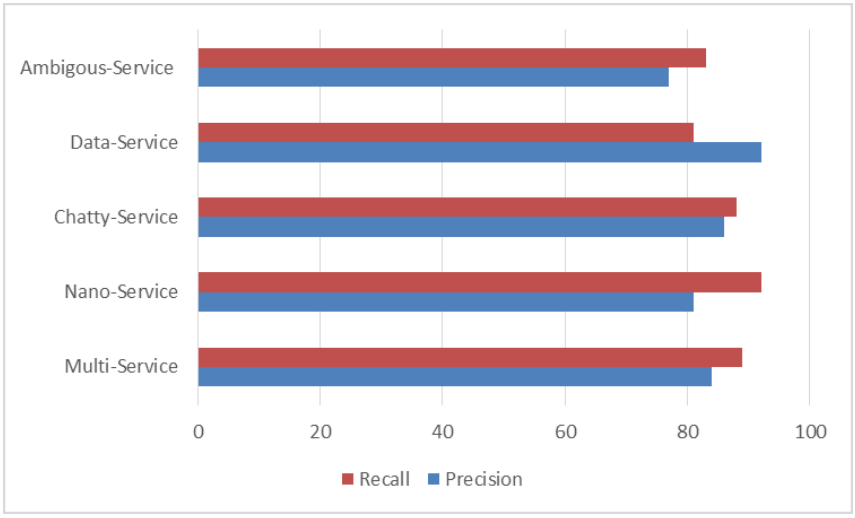


Figure 40 Average precision and recall per antipattern type on the different Web services

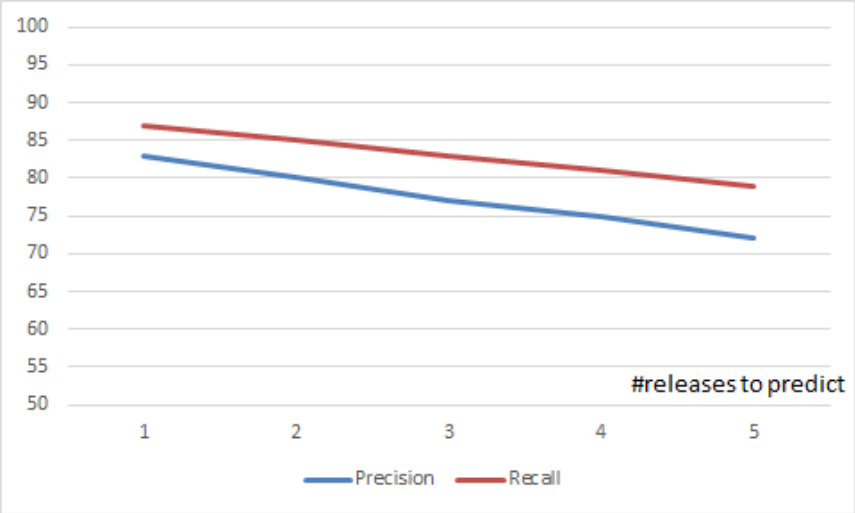


Figure 41 Average precision and recall on the Web services (except Amazon Simple Queue) per prediction step

Results for RQ3. To answer RQ3, we used a post-study questionnaire to the opinions of the participants about their experience in using our prediction tool and results. The questionnaire asked participants to rate their agreement on a Likert scale from 1 (complete disagreement) to 5 (complete agreement) with the following statements:

- The predicted metrics value is useful to estimate the risk and cost of using a specific Web service and may help the developer to select the best service based on his preferences.
- The predicted quality issues may help developers and managers to better schedule maintenance activities and reduce the cost of fixing these issues.

The agreement of the participants was 4.6 and 4.8 for the first and second statements respectively. This confirms the usefulness of our prediction results for the developers considered in our experiments.

The remaining questions of the post-study questionnaire were about the benefits and also limitations (possible improvements) of our prediction approach. We summarize in the following the feedback of the developers. Most of the participants mention that our results may help

developers of the service providers to decide when to refactor their Web service implementations. For example, they can consider to perform some refactorings when the prediction results show that the quality issue may become much more severe after few releases such as a multi-service antipattern. Thus, the developers liked the functionality of our tool that helps them to identify refactoring opportunities as early as possible.

The participants found our tool helpful for also the developers of Service-based applications. In fact, the majority of the participants mention that they consider the stability and quality of services as important criteria to select a Web service when several options are available. The non-stability of a service may negatively impact their systems in the future and it is maybe an indication that the used service includes many bugs explaining several new releases. Furthermore, the subject liked the prediction of antipatterns feature since it is easier for them to evaluate the quality of Web services in next releases based on the number of antipatterns rather than analyzing a set of metrics.

The participants also suggested some possible improvements to our prediction approach. Some participants believe that it will be very helpful to extend the tool by adding a new feature to automatically calculate the risk, cost and benefits of using different possible Web services. Another possibly suggested improvement is to use some visualization techniques to evaluate the evolution of the We services to easily estimate their stability.

3) Threats to Validity

Conclusion validity is concerned with the statistical relationship between the treatment and the outcome. The parameter tuning of the ANNs used in our experiments creates a threat that we need to evaluate in our future work. The parameters' values used in our experiments are found by trial-and-error. However, it would be an interesting perspective to design an adaptive parameter

tuning strategy for our approach so that parameters are updated during the execution in order to provide the best possible performance.

Internal validity is concerned with the causal relationship between the treatment and the outcome. We used a set of manually defined rules for the detection of possible future quality issues in the next releases [19]. However, the obtained results depend on the used rules and some of the predicted quality issues may not be important antipatterns to fix by the service provider's developers.

Construct validity is concerned with the relationship between theory and what is observed. To evaluate the relevance of our prediction results, we interviewed a group of developers. For the selection threat, the participant diversity in terms of experience could affect the results of our study. We addressed the selection threat by making sure that all the participants have almost the same experience in web development and familiarity with Web services. For the fatigue threat, we did not limit the time to fill the questionnaire and we also sent the questionnaires to the participants by email and gave them the required time to complete each of the required tasks.

External validity refers to the generalizability of our findings. In this study, we performed our experiments on six widely used Web services belonging to different domains and having different sizes. However, we cannot assert that our results can be generalized to other Web services, and to other practitioners. Future replications of this study are necessary to confirm our findings. In addition, our study was limited to the use of specific metrics. Future replications of this study are necessary to confirm our findings.

5.2.6 **Conclusion**

We proposed, in this contribution, an approach to predict the evolution of Web services. In fact, it is maybe important for subscribers to estimate the risk of using a selected service and

compare its evolution to other possible services offering the same features. Furthermore, the prediction of future changes may help web service providers to better manage available resources and efficiently schedule required maintenance activities to improve the quality. In this work, we propose to use machine learning, based on Artificial Neuronal Networks, for the prediction of the evolution of Web services interface design. To validate the proposed approach, we collected training data from quality metrics of previous releases from 6 Web services. The validation of our prediction techniques shows that the predicted metrics value, such as number of operations, on the different releases of the 6 Web services were similar to the expected ones with a very low deviation rate. In addition, most of the quality issues of the studied Web service interfaces were accurately predicted, for the next releases, with an average precision and recall higher than 82%. The survey conducted with developers also shows the relevance of prediction technique for both service providers and subscribers.

Chapter 6 **Recommendation of Web Service Refactoring**

6.1 **Context**

The Service-Oriented Architecture (SOA) [118], [119] is a modern development paradigm that changes the way of software design, implementation, and management process. It enables enterprises to adapt to new requirements efficiently and utilize the existing services provided by others conveniently. Most of the common products of SOAs are based on Web services. A Web service is a self-describing software application that can be invoked on the internet using a set of standards (SOAP, REST, etc.) [120]. Nowadays, there are many companies aim to extend their business and accessibilities by providing Web services to their clients, such as FedEx, PayPal, YouTube, and Twitter. They only need to expose the interface of the services and its description (such as WSDL document) to the clients, so clients' application can communicate and interact with the Web services.

Similar to traditional software, the interface of Web services carries the duty of interaction with its customers and maintain the binding between them. However, a Web service with one single interface may be composited by others [121] and have many different clients. Though the evolution of the Web services, context, environment and business process may lead them to violations of quality principles [4]. Such violations may present with bad design and programming practice, known as antipatterns [13], [94]. As a service provider, these antipatterns can cause problems such as fragile design, bug rate, and inflexible code. As clients of the service, they need to spend more time to understand the service, maintain activities and avoid creating antipatterns in their code.

Furthermore, the client applications and the Web services that they invoke with can be both changed over time due to new requirements or management. As a consequence, these changes can lead to a high cost of testing and reduce the maintainability of client applications [5]. Again, since the Web services interface and descriptions are provided to all clients at once as the same format, most of the clients do not need all the functionalities, but they still have to go through the same documentation and access through the same interfaces with others. This unnecessary work could lead to extra cost and development time.

To address this issue, we propose fived solutions in this chapter to recommend remodularization of Web service interface based on the client's preference and quality metrics. The recommendations will help clients by giving them different options to create their own subinterfaces, to increase the usability, understandability, and maintainability of their application. From the other hand, Web service provider can preserve the original service interface to other clients, and at the same time use this approach to customize their product to increase their most valuable customer satisfaction by creating additional interface though one or more new interfaces.

6.2 **Web service Interface Refactoring**

Modularity. The service interface *modularity* can be defined as the degree to which the operations of a service belong together and well partitioned into cohesive interfaces. A proper *modularization* of design leads to a service which is easier to use, design, develop, test, maintain, and evolve. The importance of design *modularity* was best articulated by David et al. [91]: "*Perhaps the most widely accepted quality objective for design is modularity*." Although modularity tends to be a subjective concept, measuring the degree of modularization of a software design can be achieved through two quality measures: cohesion and coupling [122].

Refactoring. Software refactoring is defined by Fowler [27] as "*the process of changing the internal structure of software to improve its quality without altering the external behavior*." Refactoring is recognized as an essential practice to improve software quality. Dudney et al. [10] have defined an initial catalog of refactoring operations for Web services including *Interface Partitioning, Interface Consolidation, Bridging Schemas or Transforms and Web Service Business Delegate*. Despite being widely used in the Object-Oriented Programming (OOP) paradigm and supported by OOP integrated development environments (IDEs), *refactoring* is still unexplored in the context of service-oriented computing (SOC). In fact, SOC refactoring is not a trivial case of recoding existing OOP refactoring techniques.

To the best of our knowledge, there is no tool currently supports developers in refactoring decision making with Web services interface based on their application releases. Our approachs proposed in this diserrtation, defines and supports three WSDL refactoring operations based on literature [4], [10]:

- *Interface Partitioning*: This refactoring decomposes a large, multi-abstraction interface into multiple interfaces that each represents a distinct abstraction.
- *Interface Consolidation*: This refactoring merges a set of interfaces that collectively implement a complete, single abstraction. Different service interfaces that operate against the same abstraction are merged into one interface that represents a single cohesive abstraction.
- *Move Operation:* This refactoring moves an operation from one interface to another one. It implies deciding what the core abstraction should be and moving the operations that do not fit that abstraction to some other interface(s).

Table 14 Adopted refactorings of Web service

Refactoring	ID	Parameters	Pre and Post-conditions
Interface Partitioning (si1,si2,ops[ ])	IP	si1: SourcePortTypye si2: TargetPortTypye ops[ ]: Operations to be extracted to si2	**Pre:** $\exists si1 \wedge isPortType(si1) \wedge \nexists si2 \wedge si1.size \geqslant 2 \wedge ops.lengh \geqslant 1$ **Post:** $\exists si2 \wedge isPortType(si2) \wedge si1.size \geqslant 1 \wedge si2.size \geqslant 1$
Interface Consolidation (si1,si2)	IC	si1: SourcePortTypye si2: TargetPortTypye	**Pre:** $\exists si1 \wedge isPortType(si1) \wedge \exists si2 \wedge isPortType(si2)$ **Post:** $\exists si1 \wedge isPortType(si1) \wedge \nexists si2$
Move Operation (op,si1,si2)	MO	op: SourceOperation si1: SourcePortTypye si2: TargetPortTypye	**Pre:** $\exists si1 \wedge isPortType(si1) \wedge \exists si2 \exists isPortType(si2) \wedge op.interface = si1 \wedge \exists op \wedge \neg declares(si2, op) \wedge si1.size \geqslant 2$ **Post:** $si1.size \geqslant 1 \wedge \neg declares(si1, op) \wedge declares(si2, op)$

Table 14 presents the set of parameters, pre and post-conditions required for each of our adopted refactorings [123].

To illustrate some of the issues related to service interface remodularization, let's consider a real-world web service example, Amazon Simple Storage Service (Amazon S3). AmazonS3 is object storage with a simple web service interface to store and retrieve data from the web, its main interface design extracted from its latest version[10] is described in Figure 42. This interface enables Amazon S3's main functionalities and communications between clients and Amazon S3 such as creating a new bucket for the storage - *CreateBucket()*, putting objects into storage – *PutObject()*, and setting object access control protocol - *SetObjectAccessControlPolicy()*.

A history of a client application releases is described in Figure 42 Motivating example (Amazon S3). Originally, it is an application created to analyze and track Amazon S3 access log and control policy and based on Amazon S3. In Version 4, *ListBuck()* and *DeleteBucket()* are added to the application, so that the users can delete useless bucket. In Version 11, the users can access data through *GetObject()* and *GetObjectExtended()* and delete useless object in the bucket.

Despite that Amazon S3 is a small and straightforward service, it is still obvious to notice that the interface of it exposes different functionalities that do not belong together including bucket management, object management, access control policy and access log operations. This design

[10] http://docs.aws.amazon.com/AmazonS3/latest/API/APISoap.html

makes the service hard to understand and reuse by the developers from industrials. Potential developers may need to understand the whole API documentation just to find few operations needed by their implementation. Furthermore, whenever there is an update of the service interface, developers have to read, re-implement, test and configure for all operations because they are all in one interface.

A better SOA design practice could consider partitioning the AmazonS3 interface into appropriately-sized, cohesive and loosely coupled interfaces that related to the management of queue attributes, such as four sub-interfaces related to “bucket management”, “object management”, “access control management” and “log management”. Furthermore, to make sure the new interface is suitable to the client and easy to use by the developers, the study of the client releases is needed to find the connection and favor between the operations. Learning from the first release of the client application, operations of “access control management” and “log management” are favored to use together; “Bucket Management” and “Object management” can be in separate interfaces since related operations are invocated in later releases separately. Figure 42 (on the right) shows one best-decomposed interfaces solution for the client user; it’s composed of three sub-interfaces related to “Access control and log management”, “Bucket Management” and “Object Management”. In addition, the new interface is built based on the evolution of client application; this can make client developers easier to test or update their application in the future. Thus the reusability, maintainability, and even performance of Web service can be improved through good decomposition design.

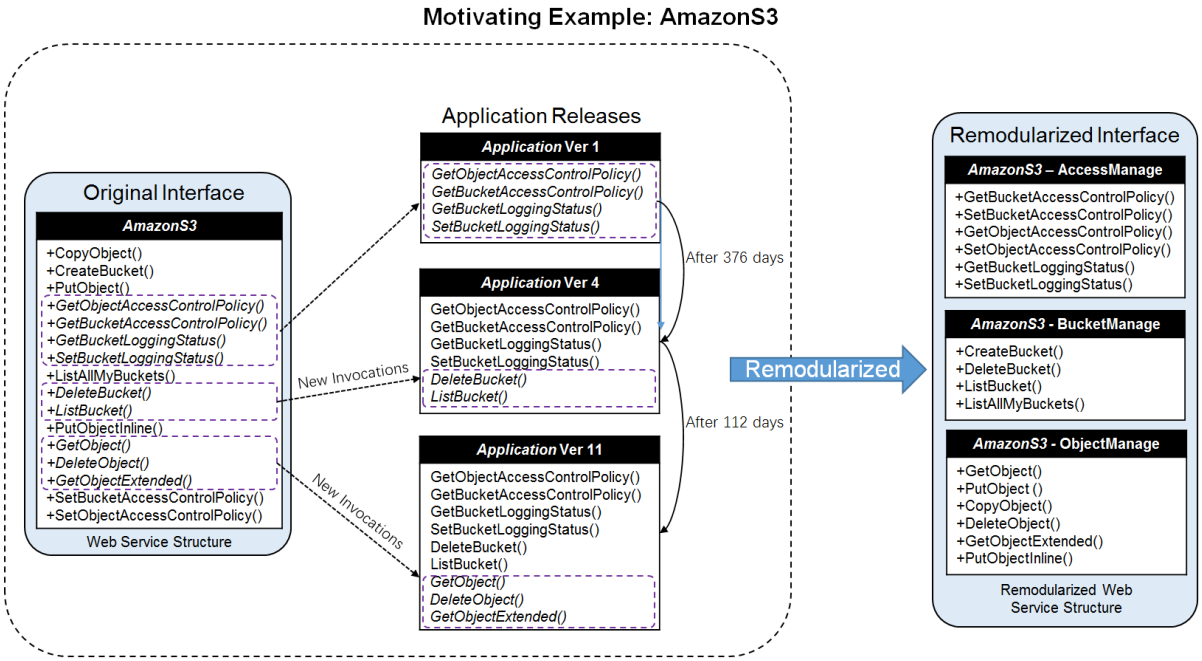


Figure 42 Motivating example (Amazon S3)

6.3 **Web Service Interface Remodularization Using Multi-Objective Optimization**

6.3.1 **Approach Overview**

Given a set of service operations there are many ways in which the module boundaries can be drawn leading to different possible modularizations of the service abstractions. The problem is a graph partitioning problem, which is known to be NP hard and therefore seems suited to a metaheuristic search-based approach [15], [124].

Figure 43 shows the overall architecture of the Multi-Ojbective approach(*WSIRem*) to the Web service interface remodularization problem. *WSIRem* aims at exploring a large search space to find a set of optimal remodularization solutions, by grouping together all collections of operations that have high cohesion into separate interfaces. *WSIRem* takes as input a Web service interface WSDL file/url to be improved. Then, *WSIRem* parses the WSDL sources by tree walking

up the XML hierarchy. It then analyses the parsed WSDL through 1) a structural analysis to extract both sequential and communicational operations similarity, and 2) a semantic analysis in order to extract semantic relationships between operations. The extracted information will be used in an optimization process based on the non-dominated sorting genetic algorithm (NSGA-II) [40]to generate remodularization solutions. An optimal modularization solution should find the best trade-off between the following objectives (*i*) maximizing cohesion, (*ii*) minimizing coupling, (*iii*) maximizing the number of interfaces, and (*iv*) minimizing the number of operations per interface. As output, the result of *WSIRem* should be a set of interfaces, one for each distinct abstraction, and each one containing the complete set of operations that operates on that abstraction.

To manipulate instances of this kind, *WSIRem* start by (*i*) creating a set of new empty interfaces, and (*ii*) assigning each operation to a unique interface. A modularization solution should assign each operation to exactly one interface, and has no empty interfaces. Then, *WSIRem* uses NSGA-II in order to find the best modularization solution that provide the best trade-off between our four objective functions.

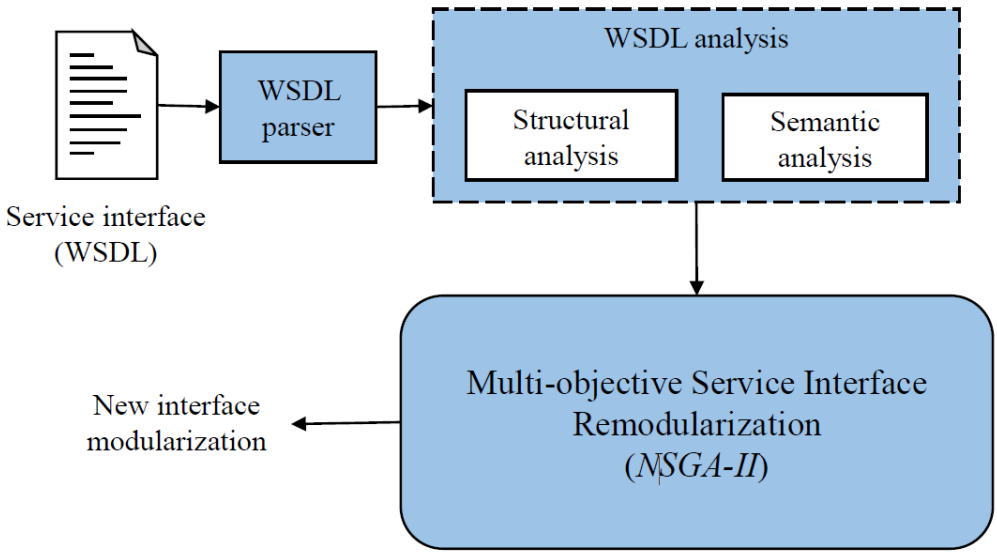


Figure 43 Overall WSIRem architecture

6.3.2 **Web Service Interface Modularization Metrics**

We define the Web service interface Modularization $\mathcal{M}$ as a decomposition of the set of operations *O* into a set of service interfaces *si*, where each interface represents a WSDL port type *PT*, i.e., a container of operations. We define the interface size, *size(si)*, by the number of its operations. Consider a Web service with set of operations $O = \{op_1, op_2, \ldots, op_n\}$ where n is the number of operations in the service. The set of possible modules, i.e., interfaces, is represented by $\mathcal{M} = \{si_1, si_2, \ldots, si_m$ where m is the number of service interfaces, and each interface has its unique number $1,2,\ldots,m$. A possible modularization solution for this problem is defined by the decision variables $X = \{x_1, x_2, \ldots, x_n\}$f, where $x_i = si$ indicates that op_i belongs to interface si. Figure 48 shows a service interface modularization example. A simple solution $X = \{1,2,3,1,3,2,2\}$, for example, denotes a modularization of seven operations into three service interfaces. Operations op_1 and op_4 are in service interface si_1, op_2, op_6 and op_7 in si_2, and finally op_3 and op_5 are in si_3. Moreover, different service operation dependencies exist in order to implement the required functionalities by the service. An appropriate modularization should maximize the cohesion within an interface while minimize coupling between interfaces. Interface Cohesion

Cohesion is a widely used metric in SOC to measure how strongly related are the operations of a service interface [5], [6], [125]. *WSIRem* employs three commonly used interface cohesion metrics that will drive the remodularization search process: sequential, communicational, and conceptual cohesion. Our cohesion metrics focus on interface-level relations, as service implementation is typically not provided by the service providers. Similarly, we do not consider information concerning the usage of operations by clients, as this information is mostly influenced by the specific scenario where the service is used.

A. Lack of Sequential Cohesion (LoC_{seq}):

The sequential similarity S_{seq} between two operations quantifies the sequential category of cohesion [6]. Two operations are deemed to be connected by a sequential control flow if the output from an operation is the input for the second operation, or vice versa. Formally, let $op_1, op_2 \in si$, two operations belonging to an interface si, then S_{seq} is defined as follows:

$$S_{seq}(op_1, op_2) = \frac{MS(I(op_1), O(op_2)) + MS(O(op_1), I(op_2))}{2}$$

where $I(op)$ and $O(op)$ refer to the input and output messages of the operation op, respectively; and $MS(I(op_1), O(op_2))$ is the function that returns the message similarity between two messages $I(op_1)$ and $O(op_2)$.

Message similarity (MS). Two messages are similar if they have common parameters, or similar types of parameters. To calculate MS of two messages m_1 and m_2, our approach is based on the average of:

- *The number of common subtrees:* it corresponds to the sum of the orders of common bottom-up subtrees of m_1 and m_2, divided by the order of the message that results from the union of m_1 and m_2, as defined in [125].
- *The number of common primitive types:* it corresponds to the Jaccard similarity between m_1 and m_2, i.e., the ratio of common primitive types in m_1 and m_2, divided by the union of primitive types of m_1 and m_2.

By combining these two measures, MS aims at capturing message similarity. The more two messages share common primitive types, the more they are likely to be related.

B. The Lack of sequential cohesion LoC_{seq} of an interface si is defined as the complement of the average S_{seq} of all pairs of operations belonging to the interface si [40]. Formally, LoC_{seq} is defined as follows:

$$LoC_{seq}(si) = 1 - \frac{\sum_{\forall(op_i, op_j) \in si, op_i \neq op_j} S_{seq}(op_i, op_j)}{\frac{|si| \times (|si| - 1)}{2}}$$

C. Lack of Communicational Cohesion (LoC_{com})

The Communicational Similarity S_{com} between two operations quantifies the communicational category of cohesion [6]. Two service operations are deemed to be connected by a communication similarity, if they share (or use) common parameter and return types, i.e., both operations are related by a reference to the same set of input and/or output data. Formally, let m_1 and m_2, two operations, then S_{com} is defined as follows:

$$S_{com}(op_1, op_2) = \frac{MS(I(op_1), I(op_2)) + MS(O(op_1), O(op_2)}{2}$$

where $I(op)$ and $O(op)$ refer to the input and output messages of the operation op, respectively; and $MS(I(op_1), I(op_2))$ is the function that returns the message similarity between two messages $I(op_1)$ and $I(op_2)$.

D. The Lack of communicational cohesion LoC_{com} of an interface si is defined as the complement of the average S_{com} of all pairs of operations belonging to the interface si [40]. Formally, LoC_{com} is defined as follows:

$$LoC_{com}(si) = 1 - \frac{\sum_{\forall(op_i, op_j) \in si, op_i \neq op_j} S_{com}(op_i, op_j)}{\frac{|si| \times (|si| - 1)}{2}}$$

E. Lack of Semantic Cohesion (LoC_{sem})

The Semantic Similarity S_{sem} between two operations quantifies the conceptual category of cohesion. We define a concrete refinement of the conceptual cohesion, as it is regarded as the strongest cohesion metric [9].

S_{sem} is based on the meaningful semantic relationships between two operations, in terms of some identifiable domain level *concept*. We expand the existing definition to get more meaningful sense of the semantic meanings embodied in the operation names. To this end, we perform a lexical analysis on operation signature. Our lexical analysis consists of the four following steps:

a) *Tokenization*. The operation names are tokenized using a camel case splitter where each name is broken down into tokens/terms based on commonly used coding standards.

b) *Filtering*. We use a stop word list to cut-off and filter out all common English words[11] from the extracted tokens. Typically, these words are irrelevant to the implemented concept. Such words carry a very low information value and can negatively affect the semantic similarity process as they have no direct relation to the business abstraction domain.

c) *Lemmatization*. This is a morphological process that transforms each word to its basic form (i.e., lemma). This process aims at reducing a word to its basic form in order to group together the different inflected forms of a basic word so they can be analyzed as a same word. Hence, different forms of words that may have similar meanings are grouped together and handled as identical word. For example, the verb 'to pay' may appear as 'pay', 'paid', 'paying', 'payment', 'payments'. The base form, 'pay' is then the lemma of all these

[11] http://www.textfixer.com/resources/common-english-words.txt

words. To do so, we use Stanford's CoreNLP[12] to find the base forms of all extracted words.

d) *Vocabulary expansion*. To enhance the effectiveness of the semantic similarity, we considered WordNet[13], a widely used lexical database that groups words into sets of cognitive synonyms, each representing a distinct concept. We use WordNet to enrich and add more informative sense to the extracted bag of words for each operation. For example, the word *customer* can be used with different synonyms (e.g., *client*, *purchaser*, etc.), but pertaining to a common domain concept.

To capture semantics or textual similarity between two bags of words A and B extracted from two operations op_1 and op_2 respectively, we use the cosine of the angle between both vectors representing A and B in a vector space using *tf-idf* (term frequency-inverse document frequency) model. We interpret term sets as vectors in the n-dimensional vector space, where each dimension corresponds to the weight of the term (tf-idf) and thus n is the overall number of terms. Formally, the S_{sem} between op_1 and op_2 corresponds to the cosine similarity of their two weighted vectors $\vec{A}$ and $\vec{B}$ and defined as follows:

$$S_{sem}(op_1, op_2) = cosine(\vec{A}, \vec{B}) = \frac{\vec{A} \cdot \vec{B}}{\| \vec{A} \| \times \| \vec{B} \|}$$

F. The Lack of semantic cohesion LoC_{sem} of an interface si is defined as the complement of the average S_{sem} of all pairs of operations belonging to the interface si. Formally, LoC_{sem} is defined as follows:

[12] nlp.stanford.edu/software/corenlp.shtml

[13] wordnet.princeton.edu

$$LoC_{sem}(si) = 1 - \frac{\sum_{\forall (op_i, op_j) \in si \; op_i \neq op_j} S_{sem}(op_i, op_j)}{\frac{|si| \times (|si| - 1)}{2}}$$

G. Lack of Cohesion (LoC)

The LoC metric covers all possible aspects of service interface cohesion as captured by the previously defined metrics LoC_{seq}, LoC_{com} and LoC_{sem}. Thus, it quantifies the total (overall) cohesion of a service interface. LoC of an interface si is defined as follows:

$$LoC(si) = w_{seq} * LoC_{seq}(si) + w_{com} * LoC_{com}(si) + w_{sem} * LoC_{sem}(si)$$

where $w_{seq} + w_{com} + w_{sem} = 1$ and their values denote the weight of each similarity measure.

Interface Coupling

Although best service design practice suggests that operations in service interface should be cohesive, e.g., operate on the same set of data, however, some interactions can arise between different service interfaces. This is because, typically, operations of a service may contribute to either single business abstractions or some other semantically meaningful concepts such as a data entity or another abstraction in the problem domain, and therefore coupling between service interfaces is sometimes unavoidable.

We define the *Coupling* metric between two service interfaces si_1 and si_2 as the average similarity between all possible pairs of operations from si_1 and si_2. Formally, the coupling, Cpl, is defined as follows:

$$Cpl(si_1, si_2) = \frac{\sum_{\forall op_i \in si_1, \forall op_j \in si_2} Sim(op_i, op_j)}{|si_1| \times |si_2|}$$

where $|si_1|$ denotes the number of operations in the interface si_1, and $Sim(op_i, op_j)$ is defined as the weighted sum of the different operations similarity measures defined in the previous section:

$$Sim(op_i, op_j) = w_{seq} * S_{seq}(op_i, op_j) + w_{com} * S_{com}(op_i, op_j) + w_{sem} * S_{sem}(op_i, op_j)$$

6.3.3 **NSGA-II Adaptation**

To adapt a search algorithm to a specific problem, the following elements should be defined: (*i*) solution representation and the generation of initial population, (*ii*) fitness function to evaluate candidate solutions according to each objective, (*iii*) change operators to generate new individuals using genetic operators (crossover and mutation). In the following we describe these element.

A. Solution representation

In our problem, a candidate solution is a service modularization, i.e., a set of interfaces, each exposes a set of cohesive operations. A valid solution should have each interface contains at least two operations, and each operation should exist in one interface. To this end, we adopt the *label-based integer* encoding [126] where a candidate solution is represented as an integer array of n positions, where n is the total number of operations available in a service. Each position corresponds to a specific operation. The integer values in the array represent the interface to which the operations belong. For instance, the modularization example in

Figure 44 is encoded as shown in Figure 45 where the operations op 1, 4 and 5 belong to the same service interface si labeled with 1; operations 2,6, 8 and 9 belong to the interface 2, and operations 3 and 7 belong to the interface 3.

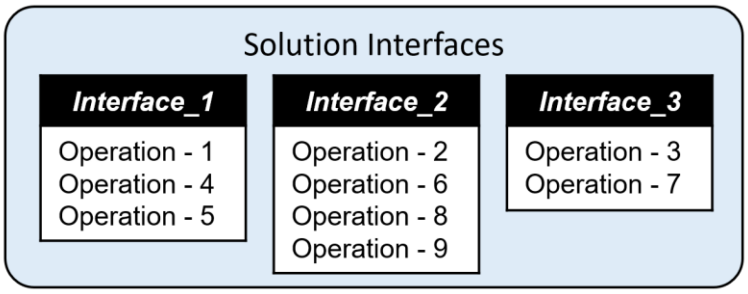


Figure 44 An example of Web service interfaces remodularization solution.

Operation ID	1	2	3	4	5	6	7	8	9
Solution Encoding	1	2	3	1	1	2	3	2	2

Figure 45 An example of a solution encoding

To create the initial population, we first define the parameter $minSize$ as minimum number of operations per interface. Then, create a number of interfaces completely random where a max number of interfaces ($\frac{n}{minSize}$) is fixed. Then, for each interface, ($minSize$) operations of the Web service are randomly assigned to it make sure the all the interfaces has at least ($minSize$) operations. Then, for the rest of the operations, we assign each one of them randomly to any of the interfaces. Furthermore, for this problem, we fixed $minSize$ at 2, as typically a core business abstraction requires at least two operations.

B. Objective Functions

The quality of each candidate modularization solution is defined by a fitness function that evaluates multiple objective and constraint dimensions. Each objective dimension refers to a specific value that should be either minimized or maximized for a solution to be considered "better" than another solution. In our approach, we optimize the following four objectives:

Cohesion: The cohesion objective function is a measure of the overall cohesion of a candidate interface modularization. This objective function corresponds to the average cohesion score of each interface in a Modularization $\mathcal{M}$ and is computed as follows:

$$Cohesion(\mathcal{M}) = 1 - \frac{\sum_{\forall si \in \mathcal{M}} LoC(si)}{|\mathcal{M}|}$$

where $LoC(si_i)$ denotes the total interface lack of cohesion given by equation 10, and $|\mathcal{M}|$ is the total number of interfaces in the modularization $\mathcal{M}$.

Coupling: The coupling objective function measures the overall coupling among interfaces in a modularization $\mathcal{M}$. This objective function corresponds to the average coupling score

between all possible pairs interfaces in a the modularization $\mathcal{M}$ in a service and is calculated as follows:

$$Coupling(\mathcal{M}) = \frac{\sum_{\forall(si_i, si_j) \in \mathcal{M}, si_i \neq si_j} Cpl(si_i, si_j)}{\frac{|\mathcal{M}| \times (|\mathcal{M}| - 1)}{2}}$$

where $Cpl(si_i, si_j)$ denotes the coupling between the interfaces si_i and si_j given by equation 9, and $|\mathcal{M}|$ is the total number of interfaces in the modularization $\mathcal{M}$.

Typically, coupling among service interfaces should be minimized as this indicates that each interface covers separate functionality aspects.

Number of interfaces (NI): This objective function refers to the total number of interfaces in the modularization $\mathcal{M}$.

$$NI(\mathcal{M}) = |\mathcal{M}|$$

The number of interfaces should be maximized in order to avoid having all operations in a single large interface.

Average number of operations per interface (AOI): The average number of operations per interface in a modularization $\mathcal{M}$ ought to be minimized to aim at appropriately, equal-sized interfaces.

$$AOI(\mathcal{M}) = \frac{\sum_{\forall si \in \mathcal{M}} Size(si)}{|\mathcal{M}|}$$

where $Size(si)$ returns the number of operations in the interface si.

One can notice that these objective functions are conflicting by nature making service interface remodularzation more challenging to find the best balance between coupling and cohesion. On the other hand, decreasing the average number of operations per interface (AOI) might result in a large number of interfaces (NI), leading to several scattered core abstractions. This makes service interface modularization a non-trivial decision-making task for developers.

Population-based search algorithms deploy crossover and mutation operators to improve the fitness of the solutions in the population in each iteration. Change operators such as crossover and mutation aim to drive the search towards near-optimal solutions, i.e., remodularizations.

The *crossover* operator is responsible for creating new solutions based on already existing ones, e.g., re-combining solutions into ones. In our adaptation, we use a single, random cut-point crossover to construct offspring solutions. It starts by selecting and splitting at random two-parent solutions. Then crossover creates two child solutions by putting, for the first child, the first part of the first parent with the second part of the second parent, and vice versa for the second child. An example of crossover is depicted in Figure 46.

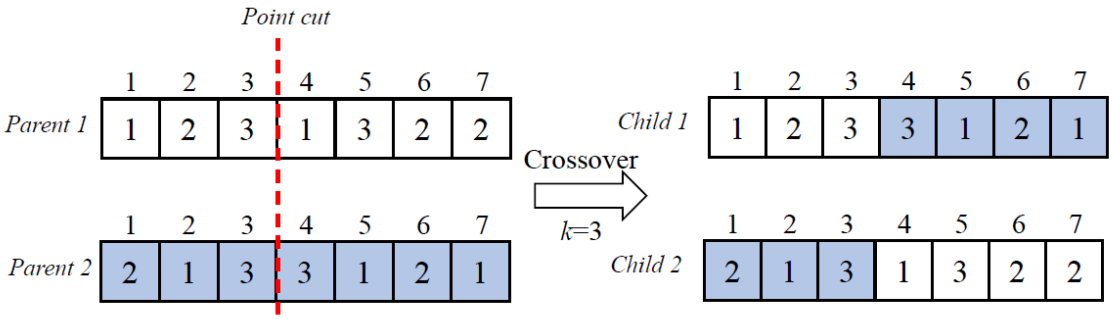


Figure 46 Crossover operator

The *mutation* operator is used to introduce slight random changes into candidate solutions. This operator guides the algorithm into areas of the search space that would not be reachable through recombination alone and avoids the convergence of the population towards a few elite solutions. With Web service interface remodularization, we use a mutation operator that picks at random one or more positions from their integer array and replaces them by other ones randomly as shown is Figure 47.

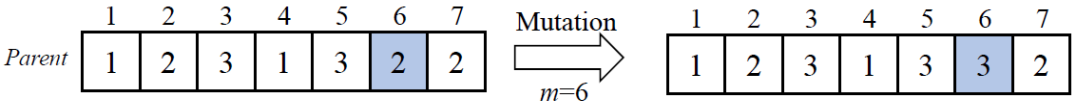


Figure 47 Mutation operator

Note that, to be valid, crossover and mutation operators should ensure that (*i*) each operation is assigned to a unique interface, and (*ii*) each interface should contain more than one operation ($minSize = 2$). In addition, when applying crossover and mutation operators we ensure the validity of the solution using a repair function that eliminates the redundancy when assigning operations to the interfaces. Thus, we ensure that an operation is not assigned to two interfaces at the same time after applying the change operators.

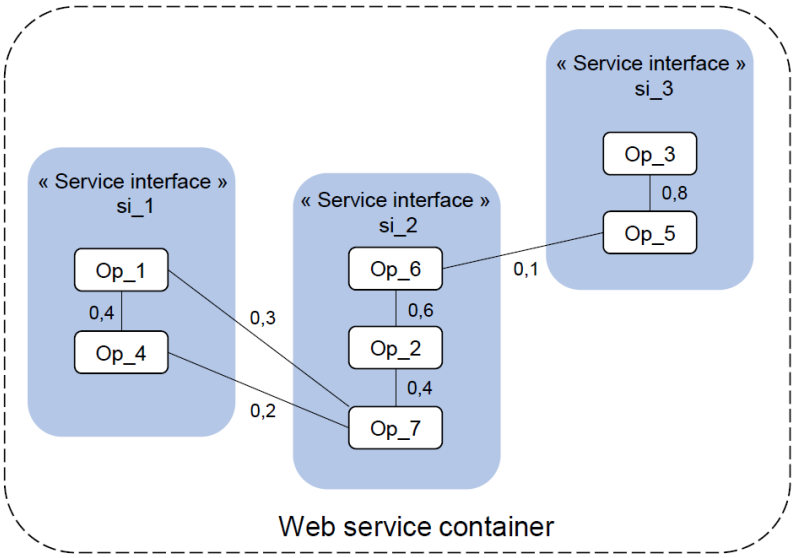


Figure 48 An example of Web service interface modularization.

Problem complexity. Finding the best partitioning of operations into cohesive service interfaces is not an obvious task for developers as the number of possible partitions can be very large causing a combinatorial explosion. The search space tends to be enormous as the number of possible partitions is given by:

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k$$

where B_n, counts the number of different possibilities of how a given set of n operations can be divided into interfaces. The order of the partitions, i.e., interfaces, as well as the order of the operations within an interface do not need to be considered. For instance, consider the AmazonEC2PortType Web service which exposes 87 operations in the version 2010. To find the right interface partitioning for AmazonEC2PortType, the number of combinations of its 87 operations, a developer need to explore $B_{87} \approx 6.39 \times 10^{98}$ possible ways to create interfaces. Due to this huge search space, an exhaustive search is unsuitable. Instead, a heuristic search maybe efficient for this kind of combinatorial problems [15], [62].

where $w_{seq} + w_{com} + w_{sem} = 1$ and their values denote the weight of each similarity measure.

6.3.4 **Validation**

1) Experimental Setup

The purpose of this study is to investigate how well our WSIRem approach provides modularization solutions and compare it with available state-of-the-art approach by Athanasopoulos et al. [5]. All the materials used in our study as well as the raw results are publicly available in a comprehensive replication package[14].

To the best of our knowlege, Athanasopoulos et al. [5] is the only existing technique that attempt to automate the service interface remodularization. In the rest of the work we refer by Greedy to denote the approach proposed in [5]. Greedy is a cohesion-based approach that

[14] *Data: http://sel.ist.osaka-u.ac.jp/~ali/WSIRem.*

iteratively split a service interface using a greedy algorithm without considering the coupling between the generated interfaces.

Our empirical study is performed through three types of evaluations:

- Evaluation with design metrics: we evaluate the impact of the suggested remodularizations by our approach on the interface design quality in terms of cohesion, coupling and modularity.
- Evaluation with interface partitioning correctness: We compare our remodularization results with those manually performed by developers in terms of precision and recall. The goal is to see if our technique can actually identify new abstractions which were improperly embedded in the original interface.
- Evaluation with developers: We asked independent developers to evaluate each of the modularizations provided by our approach, and give more qualitative feedback. For each evaluation, we present the research questions we set out to answer:

  RQ1: To what extent can WSIRem improve the service interface design quality?

  RQ2: Does WSIRem able to identify appropriate partitioning of distinct business abstractions?

  RQ3: Does WSIRem result in 'useful' interface remodularization solutions from a developer's point of view?

2) Experiment Results

To evaluate our approach, we conducted our experiment on a benchmark of 22 real-world services provided by Amazon[15] and Yahoo[16]. We selected services that are identified as god object Web service antipatterns [5], [11] with interfaces exposing at least 10 operations. We chose these

[15] http://aws.amazon.com/
[16] developer.searchmarketing.yahoo.com/docs/V6/reference/

Web services because their WSDL interfaces are publicly available, and they were previously studied in the literature [5], [57]. Table 15 presents our used benchmark. We chose these Web services because their WSDL interfaces are publicly available, and they were previously studied in the literature [5], [57].

Table 15 Experimental benchmark overview.

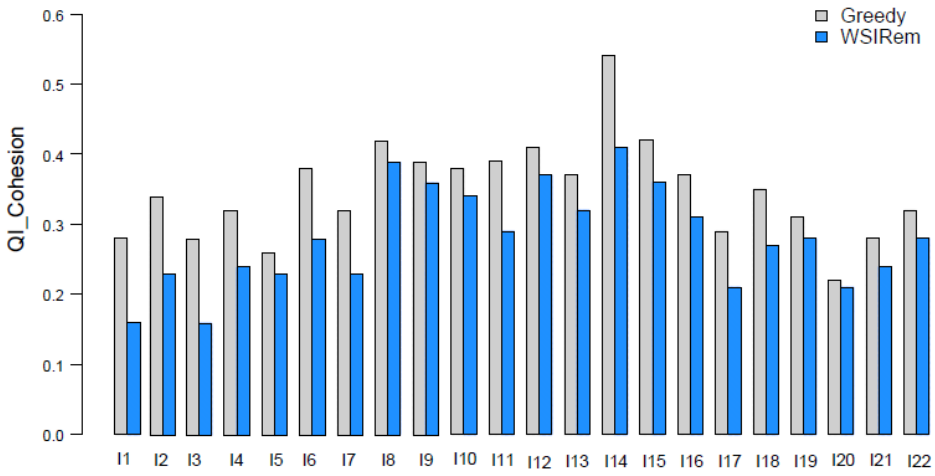
Service interface	Provider	ID	#operations	LoC_{seq}	LoC_{com}	LoC_{sem}
AutoScalingPortType	Amazon	I1	13	0,98	0,96	0,79
MechanicalTurkRequesterPortType	Amazon	I2	27	0,84	0,91	0,85
AmazonFPSPorttype	Amazon	I3	27	0,97	0,92	0,93
AmazonRDSv2PortType	Amazon	I4	23	0,96	0,91	0,58
AmazonVPCPortType	Amazon	I5	21	0,96	0,93	0,82
AmazonFWSInboundPortType	Amazon	I6	18	0,96	0,93	0,73
AmazonS3	Amazon	I7	16	0,97	0,89	0,75
AmazonSNSPortType	Amazon	I8	13	0,97	0,96	0,84
ElasticLoadBalancingPortType	Amazon	I9	13	0,97	0,93	0,72
MessageQueue	Amazon	I10	13	0,98	0,98	0,81
AmazonEC2PortType	Amazon	I11	87	0,98	0,97	0,93
KeywordService	Yahoo	I12	34	0,93	0,84	0,91
AdGroupService	Yahoo	I13	28	0,94	0,84	0,65
UserManagementService	Yahoo	I14	28	0,97	0,96	0,91
TargetingService	Yahoo	I15	23	0,96	0,74	0,74
AccountService	Yahoo	I16	20	0,98	0,92	0,88
AdService	Yahoo	I17	20	0,89	0,79	0,88
CompaignService	Yahoo	I18	19	0,91	0,83	0,91
BasicReportService	Yahoo	I19	12	0,99	0,91	0,92
TargetingConverterService	Yahoo	I20	12	0,8	0,84	0,53
ExcludedWordsService	Yahoo	I21	10	0,81	0,72	0,54
GeographicalDictionaryService	Yahoo	I22	10	0,99	0,79	0,65

To answer RQ1, we assess the design improvement that a candidate remodularization suggested by SIM will bring to the service comparing to *Greedy* [5]. Historically, software engineers have conceived metric suites as valuable tools to estimate the quality of their software artifacts [6], [125], [127]. Our evaluation is based on *Cohesion*, *Coupling*, and *Modularity* metrics. For *Cohesion*, we use the complement of the average of three widely used lack of cohesion metrics:

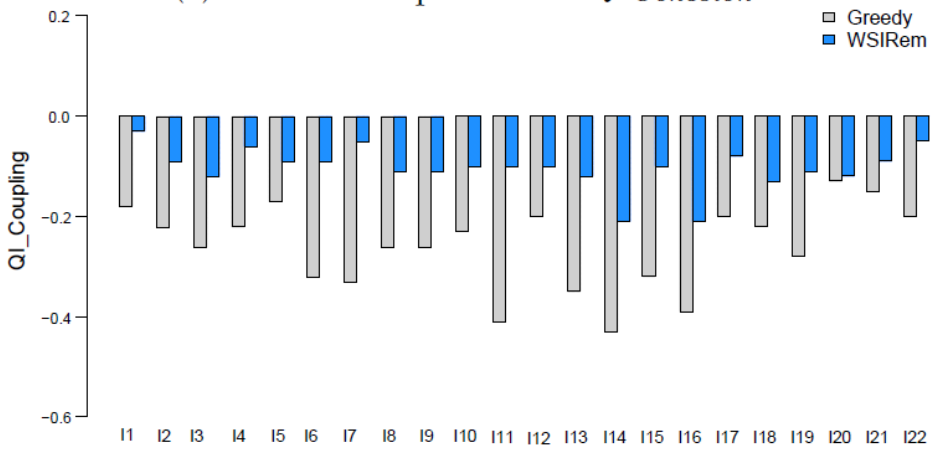
lack of sequential cohesion (LoC_{seq}), lack of communicational cohesion (LoC_{com}), and lack of semantic cohesion (LoC_{sem}). *Coupling* refers to the average coupling values between all possible pairs of interfaces. Finally, *Modularity* evaluates the balance between coupling and cohesion by combining them into a single measurement. The aim is to reward increased cohesion with a higher Modularity score and to punish increased coupling with a lower Modularity score. It has been proved that the higher the value of Modularity, the better the quality of the modularization [128]. The Modularity metric is computed as the average of the overall cohesion and coupling.

For each of these three metrics, we report the quality improvement (QI), i.e., the difference value before and after remodularization, $QI_{Cohesion}$, $QI_{Coupling}$, and $QI_{Modularity}$.

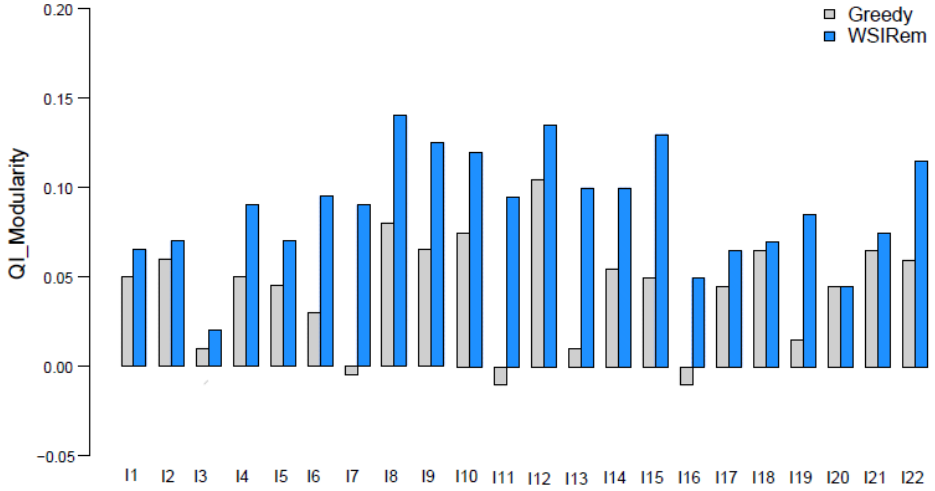
Figure 49 reports the results achieved by both *WSIRem* and *Greedy* in terms of cohesion, coupling and modularity. We expected an increase of cohesion (desired effect) due to the split of different operations exposed in the original interface. However, we also expected an increase of coupling (side effect), since splitting an interface into several interfaces typically results in an increment of the total dependencies between these interfaces. For these reasons coupling and cohesion should be measured together to make a proper judgment on the complexity and quality of the identified interfaces, using our Modularity metric.



(a) Cohesion Improvement $QI_{Cohesion}$.



(b) Coupling Improvement $QI_{Coupling}$.



(c) Modularity Improvement $QI_{Modularity}$.

Figure 49 Quality improvements achieved by WSIRem and Greedy in terms of cohesion, coupling and modularity.

Looking at Figure 49a, we can see that, for almost all the interfaces, the cohesion is greatly improved by both approaches. In particular, the improvement achieved by *Greedy* is better than *WSIRem*. However, Figure 49b shows the achieved cohesion improvement increased the coupling results. Indeed, this is natural as the original service has a single interface (thus its Coupling = 0). Consequently, any interface partitioning will result in some connections between interfaces due to the semantic similarity that is unlikely to be equals to zero and due to some shared (primitive) data types in messages. As reported in Figure 49b, *WRIRem* is able to remarkably reduce the coupling for all the services. Improvement of cohesion usually comes at the expense of increase in coupling and vice versa.

We assume that a candidate remodularization is a good design solution if the improvement of cohesion is significantly greater than the deterioration of coupling. This balance is captured by the Modularity metric as reported in Figure 49c. For the 22 services, interesting modularity improvements was achieved by WSIRem with an average of 0.08, while Greedy approach turns out to be less effective with an average of 0.04. Furthermore, we noticed that Greedy produced three modularizations for *AmazonS3* (I7), *AutoScalingPorType* (I11) and *AccountService* (I16) with deteriorated modularity due to the high coupling resulted in the new interfaces.

To answer RQ2, we asked a group of independent developers to manually decompose each of the studied interfaces in order to identify groups of operations that represent distinct core abstractions. The abstractions identified by the developers were considered as the ground truth, allowing the calculation of the precision and recall of our approach.

We compute the precision and recall scores as follows: where TP (*True Positive*) corresponds to an interface identified by the independent developer and also by the proposed approach; FP (*False Positive*) corresponds an interface identified by the proposed developer, but

not by the independent expert; FN (*False Negative*) corresponds to an interface identified by the independent developer, but not by the proposed approach.

Note that we computed TP, FP and FN at a fine-grained level, meaning that the interface identified by the proposed approach and by the independent developer should match with a Jaccard similarity of at least 80% in terms of their operations.

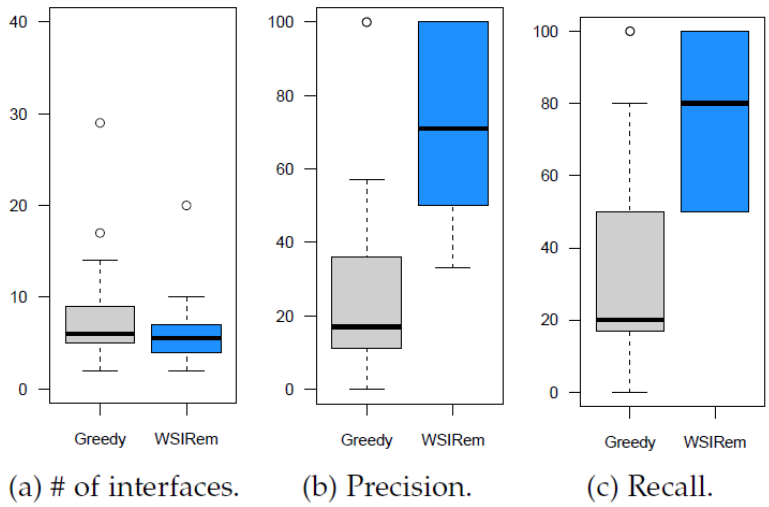
Our evaluation involved 14 independent volunteer participants including 6 industrial developers and 8 graduate students in Software Engineering (3 MSc and 5 PhD candidates). We first gathered information about the participant's background. All participants are familiar with service-oriented development and SOAP Web services with an experience ranging from 4 to 9 years. The participants were unaware of the techniques *WSIRem* and *Greedy* neither the particular research questions, in order to guarantee that there will be no bias in their judgment.

Table 16 and Figure 50 report the results for RQ2 in terms of number of generated interfaces, precision and recall of each of *WSIRem* and *Greedy*. As it can be observed from the table and figure, for the 22 services, WSIRem had an average interface split (i.e., (modularization size) of 6.04, an average precision of 73% and an average recall of 77% comparing the manual modularizations performed by developers. We consider that these values of precision and recall are high since a deviation between the proposed solution and the manual one may not be an indication of some wrong recommendations but it could be just another possible good solution. In fact, there is no single good remodularization solutions but multiple ones. On the other hand, we noticed that *Greedy* tends to produce more split interfaces with an average of 8.22, but smaller interfaces. Indeed, smaller interfaces tend to have higher cohesion. This resulted in low precision and recall with an average of 27% and 33%, respectively. We noticed that, *Greedy* generated in many cases, several interfaces with only one operation. Such finegrained interfaces will make the

service more complex and severely limit its reusability as core abstractions will be split into several small and scattered interfaces.

Table 16 Comparison results of WSIRem and Greedy in terms of (a) number of generated interfaces, (b) precision and (c) recall.

Interface	WSIRem			Greedy		
	#interfaces	Precision	Recall	#interfaces	Precision	Recall
AutoScalingPortType	4	50%	67%	6	17%	33%
MechanicalTurkRequesterPortType	7	100%	100%	17	0%	0%
AmazonFPSPorttype	10	80%	89%	11	27%	30%
AmazonRDSv2PortType	6	67%	80%	5	20%	20%
AmazonVPCPortType	6	100%	100%	6	0%	0%
AmazonFWSInboundPortType	6	67%	67%	5	40%	33%
AmazonS3	4	75%	60%	5	17%	20%
AmazonSNSPortType	5	40%	50%	6	17%	20%
ElasticLoadBalancingPortType	4	50%	67%	4	0%	0%
MessageQueue	4	50%	50%	6	50%	60%
AmazonEC2PortType	20	82%	90%	29	14%	18%
KeywordService	8	78%	88%	9	11%	14%
AdGroupService	9	100%	100%	9	100%	100%
UserManagementService	7	100%	100%	14	36%	71%
TargetingService	6	67%	80%	8	13%	20%
AccountService	6	100%	100%	14	7%	17%
AdService	5	60%	50%	3	25%	17%
CompaignService	3	67%	50%	7	14%	25%
BasicReportService	5	100%	100%	7	57%	80%
TargetingConverterService	2	100%	100%	2	100%	100%
ExcludedWordsService	3	33%	50%	4	33%	50%
GeographicalDictionaryService	3	33%	50%	4	0%	0%
Average	**6,04**	**73%**	**77%**	**8,22**	**27%**	**33%**



(a) # of interfaces. (b) Precision. (c) Recall.

Figure 50 Boxplots for the comparison results of WSIRem and Greedy in terms of (a) numer of generated interfaces, (b) precision and (c) recall.

We noticed that all fourteen participants were able to identify more independent, and sometimes completely disconnected interfaces from the original interface. These interfaces are usually the best candidates for split since they present core abstractions and do not bare strong dependency from the rest of the original interface. Another interesting observation was that *WSIRem* successfully identified remodularization solutions with 100% of precision and recall in 7 out of the 22 services while Greedy succeeded to do so only twice. On the other hand, Greedy turns out to completely fail in identifying correct remodularization in 4 cases out of 22 with 0% of precision and recall. Finally, we identify a main drawback of the *Greedy* approach from our perspective, that driving Web service interface modularization with only cohesion metrics would not be enough, and coupling, size of interfaces is as important as cohesion for good service interface design.

To answer RQ3, we asked our fourteen participants involved in RQ2 to evaluate the usefulness of three remodularization solutions, for each of the 22 cases: (*i*) the remodularization provided by WSIRem, (*ii*) the remodularization provided by Greedy, and (*iii*) a random

remodularization. The random remodularization option is considered as a 'sanity check' to make sure whether participants have seriously answered this study, as a random remodularization does not make sense.

To this end, we used a survey hosted in eSurveyPro[17], an online Web application. Specifically, for each modularization solution, we provide a high-level description of each service interfaces before and after remodularization using UML classes. Then, the participants were asked to answer the following question for each remodularization solution: "*Does the new modularization improve the understandability of the service?*" Possible answers follow a five-point Likert scale [53] to express their level of agreement: 1: *Strongly disagree*, 2: *Disagree*, 3: *Neutral,* 4: *Agree*, 5: *Fully agree*. Note that the Web application used for our survey allowed our participants to save and complete the study in multiple rounds within a maximum of 7 days available to respond. At the end of the 7 days we collected the 14 complete questionnaires.

Figure 51 and Table 17 report the results achieved by our study for the developer's assessment. Looking at Table 17, we can see that for all the studied services, the participants rated the *WSIRem* remodulations with an average score of 3.81, an average of 2.59 for the *Greedy* approach, while an average of 1.51 was recorded was recorded for the random remodularizations. This provides evidence that the remodularization solutions suggested by *WSIRem* are more adjusted to developers needs than those of *Greedy*. Moreover, on top of the 22 cases, participants identified two services, *GeographicalDictionaryService* and *AutoScalingPortType* where the original interface is relatively understandable even without remodularization, but they suggested that an early remodularization would be interesting to avoid potential difficulties in future service releases with additional operations.

[17] http://www.esurveyspro.com

Table 17 Developer's evaluation of the interface remodularizations for WSIRem, Greedy, and random modularization for each service.

Interface	WSIRem	Greedy	Random
AmazonEC2PortType	3,71	1,71	1,36
MechanicalTurkRequesterPortType	3,93	2,43	1,64
AmazonFPSPorttype	3,86	2,43	1,57
AmazonRDSv2PortType	3,79	2,50	1,57
AmazonVPCPortType	3,79	2,50	1,57
AmazonFWSInboundPortType	4,00	2,29	1,29
AmazonS3	4,00	2,21	1,43
AmazonSNSPortType	3,50	2,14	1,57
ElasticLoadBalancingPortType	3,71	2,21	1,43
MessageQueue	3,93	2,36	1,50
AutoScalingPortType	3,79	2,57	1,43
KeywordService	3,79	2,50	1,71
AdGroupService	3,71	3,71	1,36
UserManagementService	3,79	2,64	1,64
TargetingService	4,00	2,57	1,57
AccountService	3,71	2,71	1,64
AdService	3,79	3,00	1,43
CompaignService	3,93	2,93	1,43
BasicReportService	3,86	2,64	1,50
TargetingConverterService	4,00	4,00	1,79
ExcludedWordsService	3,64	2,36	1,21
GeographicalDictionaryService	3,57	2,50	1,54
Average	**3,81**	**2,59**	**1,51**

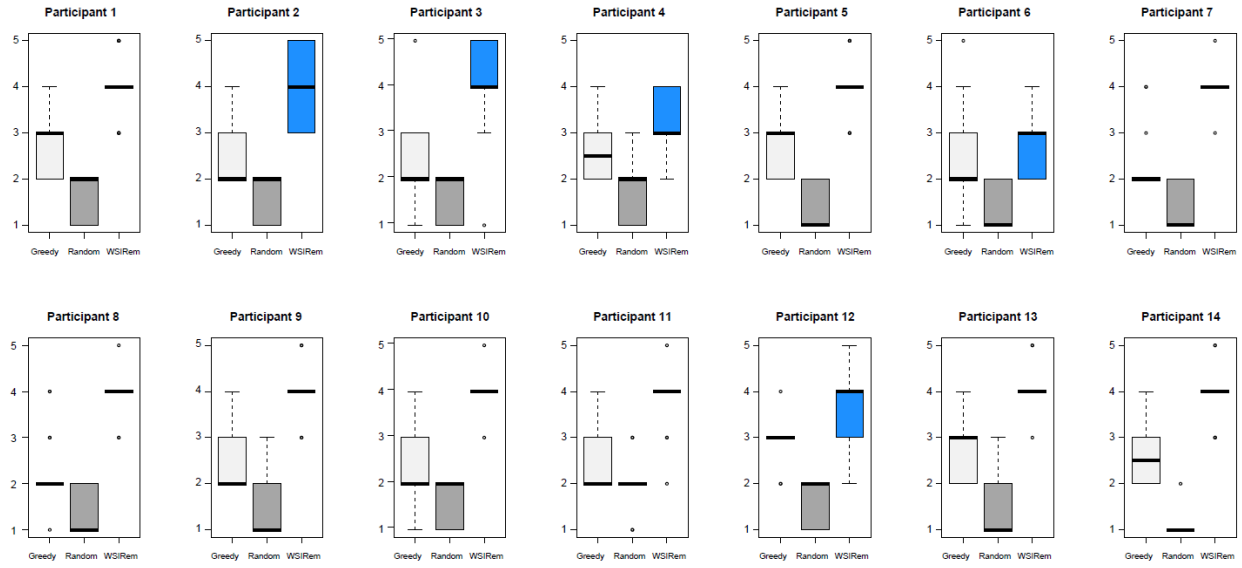


Figure 51 Developer's evaluation of the interface remodularizations for WSIRem, Greedy, and random modularization.

It is worth to note that during the evaluation, we discovered some common operations provided by different services provided by Amazon. For example, we found that *AmazonVPCPortType* and *AmazonEC2PortType* have several common operations including *CreateVpc()*, *DescribeVpcs()*, *DeleteVpc()*, *DeleteVpnConnection()*, *CreateVpnGateway()* and *DeleteVpnGateway()*. More interestingly, some generated interfaces from both *AmazonVPCPortType* and *AmazonEC2PortType* expose exactly the same operations. Although this redundancy can be related to some business constraints, best design practice in SOC suggests that common core abstraction can be implemented in separate service, making them easier to maintain, evolve and reuse.

An interesting point here was that the participants confirmed that the interfaces suggested by *WSIRem* tend to be more appropriately sized and describe distinct abstractions with less overlap. We asked one of the participants to comment on his decision for the generated Amazon EC2 interfaces, his answer was: "*This new interface structure is more understandable to me, as it was*

previously very difficult to follow and understand a bench of 87 operations exposed in a single interface. I strongly recommend the original provider to restructure his service, to allow the service to be reused more effectively".

Moreover, we noticed that in most of the cases, Greedy approach tend to split core abstractions into many interfaces. For instance, in the Amazon EC2 interface, operations related to image management was dispersed through many other interfaces: operations *RegisterImage()* and *DescribeImages()* are assigned to a new interface, *DescribeImageAttribute()* is in another interface, *CreateImage()* is in another interface, *ResetImageAttribute()*, *DeregisterImage()* and *ModifyImageAttribute()* are in another interface along with other unrelated operations [5].We asked another participant comment on this remodularization, his answer was: "*Such scattered abstractions will result in several connections between interfaces for no benefit as a large number of suggested interfaces are not representing core abstractions*". On the other hand, most of the identified interfaces expose operations related to different core abstractions. For instance, for the same Amazon EC2 service, a suggested interface by Greedy contains *DetachVolume()*, *AttachVolume()* and *DescribeInstanceAttribute()*. Results show that this design is unlikely to be desirable for developers. Moreover, the obtained results suggest that coupling is as important metric as cohesion to drive Web service interface remodularization.

6.4 **History-based Service Interface Remodularization Using Many-Objective Optimization**

6.4.1 **Many-Objective Search-Based Problem**

Recently many-objective optimization has attracted great attention in Evolutionary Multi-objective Optimization (EMO) which is one of the most active research areas in evolutionary

computation [129]. By definition, a many-objective problem is a multi-objective problem with a number of objectives greater than three. Mathematically, it could be formulated as follows:

$$\begin{cases} Min\ f(x) = [f_1(x), f_2(x), \ldots, f_M(x)]^T, & M > 3 \\ g_j(x) \geq 0 & j = 1, \ldots, P; \\ h_k(x) = 0 & k = 1, \ldots, Q; \\ x_i^L \leq x_i \leq x_i^U & i = 1, \ldots, n. \end{cases}$$

where M is the number of objective functions and is strictly greater than 3, P is the number of inequality constraints, Q is the number of equality constraints, x_i^L and x_i^U correspond to the lower and upper bounds of the decision variable x_i (i.e., ith component of x). A solution x satisfying the $(P + Q)$ constraints is said to be feasible, and the set of all feasible solutions defines the feasible search space denoted by Ω.

In this formulation, all the considered objectives are to be minimized, since maximization can be easily turned to minimization based on the duality principle. Over the two past decades, several Multi-Objective Evolutionary Algorithms (MOEAs) have been proposed with the hope to work with any number of objectives M. Unfortunately, it has been demonstrated that most MOEAs are ineffective in handling such type of problems. For example, NSGA-II [40], which is one of the most popular MOEAs, compares solutions based on their non-domination ranks. Solutions with best ranks are emphasized in order to converge to the Pareto front. When M > 3, only the first rank may be assigned to every solution as almost all population individuals become non-dominated with each other [130], [131]. Without a variety of ranks, NSGA-II cannot keep the adequate search pressure in high dimensional objective spaces.

In this work, our problem requires a search for a solution which balances multiple objectives and constraints to achieve near optimal or optimal results. This search can be fastidious and requires a labor-intensive human activity. Search-based many-objective techniques have provided new ways, based on heuristics, transforming many-objective problems from human-

based search to machine-based search techniques. Thus, the use of heuristics can guide the automated search and avoid the tedious human-in-the-loop manual activities. However, even in software engineering field, many currently existing techniques lack scalability to meet the demands of high dimensional solutions. According to a recent survey by Harman [132], most software engineering problems are naturally multi-objective. However, they are mostly handled from a mono-objective perspective. For software engineering problems, multi-objective optimization techniques have been proposed in a few works [15], [65], [67]. To the best of our knowledge no on has applied many-objective techniques to solve Web service remodularization problem.

We investigate, in this work, the applicability of many-objective techniques for the Web service remodularization problem where five objectives are considered to find the most suitable remodularization suggestions for developers.

6.4.2 **Approach Overview**

Figure 52 shows our approach overview to the History-based Web service interface remodularization problem. It targets to explore a large search space and find a set of optimal remodularization solutions, by grouping together all collections of operations that have high cohesion and history preference into separate interfaces. The approach takes two inputs: an interface WSDL file/URL of the Web service to be improved and the source code of a series of client application releases. First, parsing the WSDL sources though tree walking up the XML hierarchy to extract the Web service structure data (e.g., operation, message, and input/output). Then, Code analysis module extracts all of the Web service operations being used in the first release and its release date, as well as the added operations in the other releases and their release dates by scanning all versions of input source code files. The extracted information from first two

steps will be used in an optimization process based on the non-dominated sorting genetic algorithm (NSGA-III) to generate remodularization solutions. During the execution of the NSGA-III, it generates new interface design, evaluates and select them based on 1) a structural analysis to calculate both sequential and communicational operations similarity, 2) a semantic analysis to calculate semantic relationships between operations, 3) a history-based analysis to calculate the history-preference score.

An optimal modularization solution should find the best trade-off between the following objectives (i) maximizing history-based similarity, (ii) maximizing cohesion, (iii) minimizing coupling, (iv) maximizing the number of interfaces, and (v) maximizing the number of operations per interface.

As output, the result should be a set of interfaces; each interface is a new distinct design of the same operations/functionalities of the input Web service.

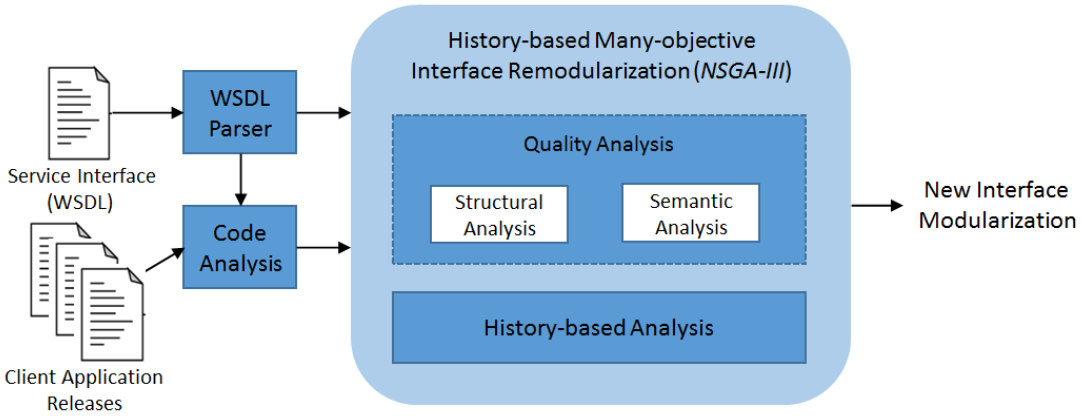


Figure 52 Solution encoding

To manipulate instances of this kind, at the beginning of NSGA-III, it starts by (i) creating a random size of new empty interfaces, and (ii) for each interface, selects randomly two operations from all the operations then remove it from the queue. (iii) Assign the rest of the operations one by one into the interfaces randomly. A modularization solution should include all the operations

in different interfaces, and there is no interfaces have less than two operations. Then, the algorithm starts to find the best modularization solution set that provides the best trade-off between our four objective functions.

In order to measure how Web service remodularization interfaces are suitable to the client application, we introduce two metrics, Operation Similarity and Invocation Time Scale to validate the correlation between the remodularization solution and evolution of the client application. A good Web interface design should respect the invocation usage of the client application and the change history of the client application so that the client developers fell comfort to understand or convenient to use the new interfaces. In this work we include the metrics listed in Chapter 6.3.2 (page 139), as well as the following metrcis:

A. Operation Similarity (OS)

Operation Similarity is a metric defined by us to quantify the similarity between interface and history of operation invocations of the client application. It's important that the interface remodularization should respect the client usage and their change history. Because of client developers understand the code and changes most of the time, if the design of Web service is similar, they should spend less time to understand and fell easy to maintain the usage of the Web service. In Figure 42, Access control and log management are added in the first release together, so it's cohesive and easier to understand to group them together as in one interface for the developers.

Definition (Operation Similarity (OS)). The operations of one interface should contain more changes that are made one release to be easier to understand or use by the developers. Formally, to one interface si, the OS is defined as follows:

$$OS(si) = \max_{\forall(d_i \in HOI)} \left(\frac{|d_i \cap si|}{|d_i|}\right)$$

Where HOI refers to the history of operation invocations, d_i represents the groups of Web service operations that are being introduced separetly during the evolution of the application. Initially, d_0 represents the invocated operations of first client release, d_i where $i > 0$ represents the changes introduced considering the previous release. If there is no new operations invoked in one release regarding to the Web serivce, there is no d_i generated for that version. For example, in Figure 42, at 4th release, we generate $d_1 = (DeleteBucket, ListBuck)$.

B. Invocation Time Scale (ITS)

Invocation time scale quantifies the time gap between the operations in one interface regarding the evolution timeline. The motivation of defining this metric is to make the most use of release history and make up the limitation of OS. During the evolution of software, the changes that are made in longer time gap tends to represent different funtionailties. Therefore, we decide to calculte the operations time gap within one interfaces, if the operations were introduced in client application during a large time scale, it means this interface tends to be less cohesive in the view of client developers. Also, the operation similairty could have its weakness, because if we put opertions of different d_i together in one interface, the score is 1 which is the best possbile score, but obviouly, this is not respecting the modular design best practices. By introducing this metrcis, we can guide the algorithm search and solve this limitation.

Definition (Invocation Time Scale (ITS)). Invocation Time Scale is the degree at which operations of one interface shares same introduced time. Formally, to one interface si, the ITS is defined as follows:

$$ITS(si) = \frac{T_{Last} - T_{First}}{T_{Max}}$$

where T_{Last} and T_{First} refer to the release time of the first and last version of client applications that have introduced at lease one opertion of si at the time. T_{Max} represents the maxium time

difference, in most cases is the time scale from first input release date to the last one. For example, in Figure 42, "BuckManage" interface only contains operations that are being introduced during Version 4, so $T_{Last} = T_{First} = T_{V4}$, T_{Max}= 488 (days) and $ITS(BuckManage)$ = 0 (days).

6.4.3 **NSGA-III and Problem Adaptation**

A. NSGA-III, Many-Objective Optimization Algorithm

NSGA-III is a recent many-objective algorithm proposed by [46]. The basic framework remains similar to the original NSGA-II algorithm with significant changes in its selection mechanism. Figure 53 shows the pseudo-code of the NSGA-III procedure for a particular generation *t*. First, the parent population P_t (of size *N*) is randomly initialized in the specified domain, and then the binary tournament selection, an offspring population Q_t is created by applying crossover and mutation operators to P_t. Thereafter, both populations are combined and sorted according to their domination level and the best *N* members are selected from the combined population to form the parent population for the next generation. The fundamental difference between NSGA-II and NSGA-III is the niche preservation operation: Unlike NSGA-II, NSGA-III starts with a set of reference points Z^r. The set of uniformly distributed reference points is generated using the method of [133] which is well-detailed and described in [134].

After the non-dominated sorting, all acceptable front members and the last front F_l that could not be completely accepted are saved in a set S_t. Members in S_t/F_l have selected right away for the next generation. However, the remaining members are selected from F_l such that the desired diversity is maintained in the population. Original NSGA-II uses the crowding distance measure for selecting a well-distributed set of points, however, in NSGA-III the supplied reference points (Z^r) are used to select these remaining members. To accomplish this, objective values and reference points are first normalized so that they have an identical range. Thereafter, orthogonal distance

between a member in S_t and each of the reference lines (joining the ideal point, i.e., the vector composed of 7 zero and a reference point) is computed. The member is then associated with the reference point having the smallest orthogonal distance. Next, the niche count ρ for each reference point, defined as the number of members in S_l/F_l that are associated with the reference point, is computed for further processing. The reference point having the minimum niche count is identified and the member from the last front F_l that is associated with it is included in the final population. The niche count of the identified reference point is increased by one and the procedure is repeated to fill up population P_{t+1}.

It is meaningless that a reference point may have one or more population members associated with it or need not have any population member associated with it. Let us denote this niche count as ρ_j for the j-th reference point. We now devise a new niche-preserving operation as follows. First, we identify the reference point set $J_{\min} = \{j: \text{argmin}_j (\rho_j)\}$ having minimum ρ_j. In case of multiple such reference points, one ($j^* \in J_{\min}$) is chosen at random. If $\rho_{j^*} = 0$ (meaning that there is no associated P_{t+1} member to the reference point j^*), two scenarios can occur. First, there exist one or more members in front F_l that are already associated with the reference point j^*. In this case, the one having the shortest perpendicular distance from the reference line is added to P_{t+1}. The count ρ_{j^*} is then increased by one. Second, the front F_l doesn't have any member associated with the reference point j^*. In this case, the reference point is excluded from further consideration for the current generation. In the event of $\rho_{j^*} \geq 1$ (meaning that already one member associated with the reference point exists), a randomly chosen member, if exists, from front F_l that is associated with the reference point F_l is added to P_{t+1}. If such a member exists, the count ρ_{j^*} is incremented by one. After ρ_j counts are updated, the procedure is repeated for a total of K times to increase the population size of P_{t+1} to N.

NSGA-III procedure at generation *t*

Input: H structured reference points Z^s, parent population P_t

Output: P_{t+1}

Begin
$S_t \leftarrow \emptyset$, $i \leftarrow 1$;
$Q_t \leftarrow$ Variation (P_t);
$R_t \leftarrow P_t \cup Q_t$;
(F_1, F_2, ...) $\leftarrow$ Non-dominationed_Sort (R_t);
Repeat
  $S_t \leftarrow S_t \cup F_i$; $i \leftarrow i$+1;
Until $|S_t| \geq N$;
$F_l \leftarrow F_i$; /*Last front to be included*/
If $|S_t| = N$ **then**
  $P_{t+1} \leftarrow S_t$;
Else
  $P_{t+1} \leftarrow \bigcup_{j=1}^{l-1} F_j$;
  /*Number of points to be chosen from F_l*/
  $K \leftarrow N - |P_{t+1}|$;
  /*Normalize objectives and create reference set Z^r*/
  Normalize (F^M; S_t; Z^r; Z^s);
  /*Associate each member s of S_t with a reference point*/
  /*$\pi(s)$: closest reference point*/
  /*$d(s)$: distance between s and $\pi(s)$*/
  [$\pi(s)$, $d(s)$] $\leftarrow$ Associate (S_t, Z^r);
  /*Compute niche count of reference point $j \in Z^r$ */
  $\rho_j \leftarrow \sum_{s \in S_t / F_l} ((\pi(s) = j) \ ? \ 1 : 0)$;
  /*Choose K members one at a time from F_l to construct P_{t+1}*/
  Niching (K, ρ_j, $\pi(s)$, $d(s)$, Z^r, F_l, P_{t+1});
End If
End

Figure 53 Pseudo-code of NSGA-III main procedure

B. Fitness Functions

The quality of each candidate modularization solution is defined by a fitness function that evaluates multiple objective and constraint dimensions. Each objective dimension refers to a specific value that should be either minimized or maximized for a solution to be considered "better" or "worse" than another solution. In our approach, we optimize the following five fitness functions:

1) **History-based similarity (HS)**. The history-based similarity measures the overall similarity of a candidate interface $\mathcal{M}$ to the developer preference based on the application release history.

This fitness function is composed by the operation similarity score and invocation time scale of each interface in a remodularization $\mathcal{M}$. It's computed as follows:

$$HS(\mathcal{M}) = \frac{\sum_{\forall si \in \mathcal{M}} \frac{OS(si) + (1 - ITS(si))}{2}}{|\mathcal{M}|}$$

Where $OS(si)$ and $ITS(si)$ are the score of operation similarity and invocation time scale score given by the equation 2 and 3, and $|\mathcal{M}|$ is the total number of interfaces in modularization $\mathcal{M}$.

2) **Cohesion**. The cohesion fitness function is a measure of the overall cohesion of a candidate interface modularization. This fitness function corresponds to the average cohesion score of each interface in a Modularization $\mathcal{M}$ and is computed as follows:

$$Cohesion(\mathcal{M}) = 1 - \frac{\sum_{\forall si \in \mathcal{M}} LoC(si)}{|\mathcal{M}|}$$

where $LoC(si_i)$ denotes the total interface lack of cohesion given by equation 10, and $|\mathcal{M}|$ is the total number of interfaces in the modularization $\mathcal{M}$.

3) **Coupling**. The coupling fitness function measures the overall coupling between operations among all interfaces in a modularization $\mathcal{M}$. This fitness function corresponds to the average coupling score between all possible pairs interfaces in the modularization $\mathcal{M}$ in a service and is calculated as follows:

$$Coupling(\mathcal{M}) = \frac{\sum_{\substack{\forall (si_i, si_j) \in \mathcal{M} \\ si_i \neq si_j}} Cpl(si_i, si_j)}{\frac{|\mathcal{M}| \times (|\mathcal{M}| - 1)}{2}}$$

where $Cpl(si_i, si_j)$ denotes the coupling between the interfaces si_i and si_j given by equation 11, and $|\mathcal{M}|$ is the total number of interfaces in the modularization $\mathcal{M}$.

Typically, coupling among service interfaces should be minimized as this indicates that each interface represents separate functionality aspects.

4) **Number of interfaces (NI)**. Number of interfaces fitness function refers to the total number of interfaces in the modularization $\mathcal{M}$.

$$NI(\mathcal{M}) = |\mathcal{M}|$$

The number of interfaces should be maximized in order to avoid having all operations in a single large interface.

5) **Average number of operations per interface (AOI).** The average number of operations per interface in a modularization $\mathcal{M}$:

$$AOI(\mathcal{M}) = \frac{\sum_{\forall si \in \mathcal{M}} Size(si)}{|\mathcal{M}|}$$

Where $Size(si)$ returns the number of operations in the interface si. This fitness functiong ought to be maximized to avoid having too many interfaces and over-splitting the Web services. aim at appropriately, equal-sized interfaces.

One can notice that the first three objective functions are conflicting by nature making service interface remodularization more challenging to find the best balance between coupling and cohesion. On the other hand, looking at the last two fitness function, decreasing the average number of operations per interface (AOI) might result in a large number of interfaces (NI), leading to several scattered core abstractions. By introducing these two conflicting fitness functions, the solution set's diversity is increased regarding interface numbers. In another word, our service interface remodularization approach provides solutions with more interface size choices to the developers while providing the design that is cohesive and easier to understand or reuse by them.

C. Evolutionary Operators

Population-based search algorithms require evolutionary operators to improve the fitness functions of the solutions in the population at each iteration. Evolutionary operators such as crossover and mutation aim to promote the search towards to the optimal solutions, in our case, to the best remodularizations. The evolutionary operators are used to creating new solutions based on the existing one/ones.

The *crossover* operator is using more than one solution to create the new and different solutions, e.g., re-combining solutions into ones. In our adaptation, we use a single, random cut-point crossover to construct offspring solutions. The cut-point selects and splits at random two-parent solutions. Then to perform crossover action, swap the first or second part of the solutions, so that two new child solutions are created based on the existing two. An example of crossover is depicted in Figure 54.

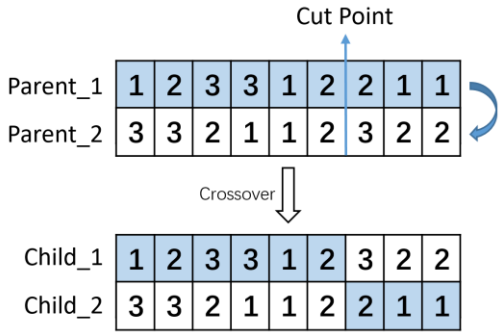


Figure 54 An example of crossover.

The *mutation* operator is used to introduce minor random changes into the parent solution. This operator promotes the algorithm into the location of the search space that would not be reachable through recombination alone and avoids the convergence of the population towards a few elite solutions. With Web service interface remodularization, we define two types of mutation operator to guide the search. The first mutation operator that picks at random one or more positions from their integer array and replaces them by other ones randomly. The second mutation operator

picks two random positions with different integer then swap them. The first operator is same as moving one or more operations to a new interface, while the second operator is equal to swap two operations in the different interfaces. The examples of these two operators are in Figure 55. These two mutation operators are randomly () selected to perform during the mutation stage.

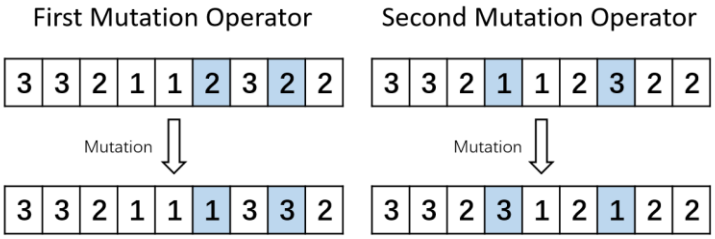


Figure 55 The examples of mutation.

Note that, to be valid, crossover and mutation operators should ensure that (*i*) each operation is assigned to a unique interface, and (*ii*) each interface should contain more than one operation ($minSize = 2$). If any child solution is failed to pass the validation, a new crossover/mutation operator should be applied to the parent solutions. In addition, when applying crossover and mutation operators we ensure the validity of the solution using a repair function that eliminates the redundancy when assigning operations to the interfaces. Thus, we ensure that an operation is not assigned to two interfaces at the same time after applying the change operators.

6.5 **Improving Web Services Design Quality Using Heuristic Search and Machine Learning**

The evolution of Web services may have a negative impact on the design quality of the interface by concatenating many non-cohesive operations that are semantically unrelated. The Web services interface design becomes unnecessarily complex for users to find relevant operations to be used in their services-based systems. An example of well-known interface design defect is

the God object Web service (GOWS) [10], [12]. GOWS implements many operations related to different business and technical abstractions in a single service interface leading to low cohesion of its operations and high unavailability to end users because it is overloaded. Indeed, the modularization process of how operations should be exposed through a service interface can have an impact on the performance, popularity and reusability of the service and it is not a trivial task.

Recently, several studies provided solutions to improve the design of Web service interfaces for the users/subscribers [1], [2], [5], [12], [35], [69]. However, most of these studies addressed the problem of the detection of design defects of Web services interface based on declarative rule specification and not the correction step to fix these design defects. In these existing techniques, Web services modularization solutions are evaluated based on the use of quality metrics. However, the evaluation of the design quality is subjective and difficult to formalize using quality metrics with the appropriate threshold values due to several reasons.

Several challenges could be discussed around the modularization of Web services interface. First, there is no consensus about the definition of Web services design defects [10], [35], [135], [136] also called antipatterns, due to the various user behaviors and contexts. Thus, it is difficult to formalize the definitions of these design violations in terms of quality metrics then use them to evaluate the quality of a Web service modularization solution. Second, existing studies do not include the user in the loop to analyze the suggested modularization solutions and give their feed-back during the design improvement process. Third, the computational complexity of some Web services quality metrics is expensive thus the defined fitness function to evaluate proposed Web services design changes can be expensive. Fourth, deciding on how to decompose/modularize an interface is subjective and difficult to automate since it is required to integrate the feedback of users during the modularization process Finally, quality metrics can just evaluate the structural

improvements of the design after applying the suggested interface changes but it is difficult to evaluate the semantic coherence of the design without an interactive user interpretation.

We propose, in this work, a Genetic Algorithm (GA)-based learning algorithm [44] for Web services interface modularization based on Artificial Neural Networks (ANN) [45]. The proposed approach is based on the important feedback of the user to guide the search for relevant Web services modularization solutions using predictive models. To the best of our knowledge, the use of predictive models has not been used to improve the quality of Web services design. In the proposed approach, we are modeling the user's design preferences using ANN as a predictive model to approximate the fitness function for the evaluation of the Web services modularization solutions. The user is asked to evaluate manually Web services interface modularization solutions suggested by a Genetic Algorithm (GA) for few iterations then these examples are used as a training set for the ANNs to evaluate the solutions of the GA in the next iterations.

We evaluated our approach on a set of 82 real-world Web services, extracted from an existing benchmark [5], [12]. Statistical analysis of our experiments shows that our interactive approach performed significantly better than the state-of-the-art modularization techniques [5], [69] in terms of design improvements and fixing design defects. The primary contributions of this work can be summarized as follows:

This contribution introduces a novel way to modularize and improve the design quality of Web services using interactive predictive modeling optimization. The proposed technique supports the adaptation of interface design solutions based on the user without the need to use specific design quality metrics. To the best of our knowledge, we propose the first approach to interactively generate a modularized Web services interface using predictive modeling techniques.

The section reports the results of an empirical study on an implementation of our approach. The obtained results provide evidence to support the claim that our proposal is more efficient, on average, than existing Web services modularization techniques based on 82 real-world services.

6.5.1 **Approach**

As described in Figure 56, our approach takes as input the Web services interface to modularize, list of possible operators (decompose a port Type or merge port Types or move operations) and the number of user's interactions during the search process. It generates as output the best sequence of design changes/operators that improves the quality of the Web service interface. Our approach is composed of two main components: the interactive component (IGA) and the learning module (LGA).

The algorithm starts first by executing the IGA component where the designer evaluates the modularization solutions manually generated by a genetic algorithm (GA) [44] for a number of iterations. The user evaluates the feasibility and the efficiency/quality of the suggested suggestions one by one since each modularization solution is a sequence of change operator (decompose or merge or move). Thus, the user classifies all the suggested design changes (modules) as good or not one by one based on his preferences and gives the different port Types values between 0 and 1.

After executing the IGA component for a number of iterations, all the evaluated solutions by the user are considered as training set for the second component LGA of the algorithm. The LGA component executes an Artificial Neural Network (ANN) to generate a predictive model to approximate the evaluation of the interface modularization solutions in the next iteration of the GA. Thus, our approach does not require the definition of a fitness function. Alternatively, the LGA incorporates many components to approximate the unknown target function f . Those

components are the training set, the learning algorithm and the predictive model. For each new sequence of refactoring X_{k+1}, the goal of learning is to maximize the accuracy of the evaluation y_{k+1}. We applied the ANN as being among the most reliable predictive models, especially, in the case of noisy and incomplete data. Its architecture is chosen to be a multilayered architecture in which all neurons are fully connected; weights of connections have been, randomly, set at the beginning of the training. Regarding the activation function, the sigmoid function is applied [45] as being adequate in the case of continuous data. The network is composed of three layers: the first layer is composed of p input neurons. Each neuron is assigned the value x_{kt}. The hidden layer is composed of a set of hidden neurons. The learning algorithm is an iterative algorithm that allows the training of the network. Its performance is controlled by two parameters. The first parameter is the momentum factor that tries to avoid local minima by stabilizing weights. The second factor is the learning rate which is responsible of the rapidity of the adjustment of weights.

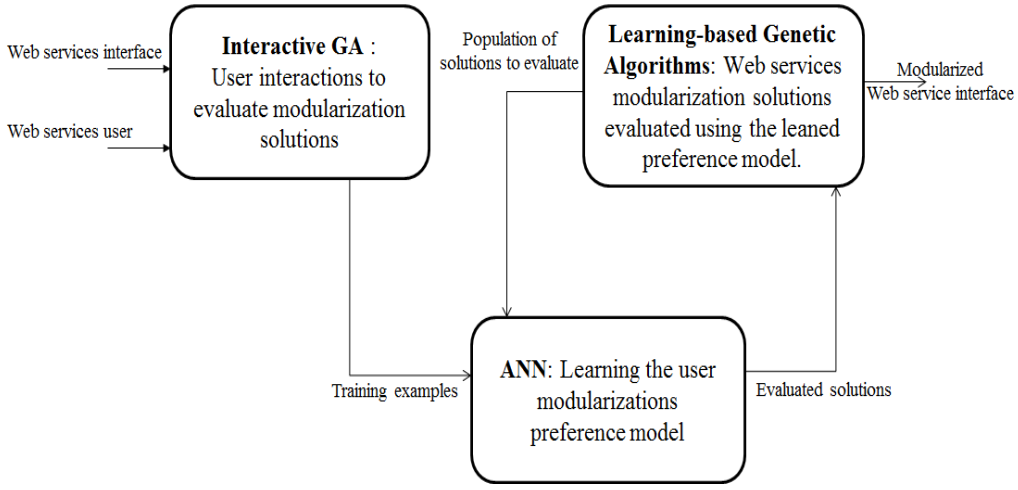


Figure 56 Approach overview

6.5.2 **Problem Adaptation**

A. Training Set and Data Normalization

Before the learning process, the data used in the training set should be normalized. In our case, we choose to apply the Min-max technique since it is among the most accurate techniques. We used the following data representation to the GA-based learning problem using ANN for software refactoring. Let us denote by E the training set of the ANN. It is composed of a set of couples that represent the refactoring sequence and its evaluation.

$$E = \{(X_1, y_1), (X_2, y_2), (X_3, y_3), ..., (X_k, y_k), ..., (X_n, y_n)\}, k \in [1..n]$$

X_k is an interface refactoring sequence represented as $X_k = [x_{k1}, x_{k2}, ..., x_{kt}, ..., x_{kp}], t \in [1..p]$.

y_k is the evaluation associated to the k[th] refactoring sequence in the range $y_k \in [0..1]$.

Let's denote by O the matrix that includes numerical values related to the set of refactorings and by Y the vector that contains numerical values representing X_k's evaluations. O is composed of n lines and p columns where n is equal to the number of refactoring sequences and p is equal to the number of solutions.

$$O = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1p} \\ x_{21} & x_{22} & \dots & x_{2p} \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ x_{n1} & x_{n2} & \dots & x_{np} \end{bmatrix} \qquad Y = \begin{bmatrix} y1 \\ y2 \\ . \\ . \\ . \\ y_n \end{bmatrix}$$

B. Change operators

In each search algorithm, the variation operators play the key role of moving within the search space with the aim of driving the search towards optimal solutions. For the crossover, we use the one-point crossover operator. It starts by selecting and splitting at random two parent solutions. Then, this operator creates two child solutions by putting, for the first child, the first part of the first parent with the second part of the second parent, and vice-versa for the second child. It is important to note that in multi-objective optimization, it is better to create children that are close to their parents to have a more efficient search process. For mutation, we use the bit-string mutation

operator that picks probabilistically one or more modularization operations from its or their associated sequence and replaces them by other ones from the initial list of possible refactorings. When applying the change operators, different pre- and post-conditions are checked to ensure the applicability of the newly generated solutions such as removing redundant operations or conflicts between operations such as assigning the same operation to two different port types.

6.5.3 **Validation**

1) Experimental Setup

To evaluate the ability of our Web services modularization framework to generate a good design quality, we conducted a set of experiments based on 82 real-world web services as described in Table 18. the obtained results are subsequently statistically analyzed with the aim of comparing our proposal with a variety of existing fully-automated approaches. In this section, we first present our research questions and then describe and discuss the obtained results.

A. Research Questions

We defined three research questions that address the applicability, performance in comparison to existing fully-automated Web services modularization approaches [5], [69] and the usefulness of our approach. The three research questions are as follows:

- RQ1: To what extent can our approach recommend relevant Web services design improvements?
- RQ2: How does our interactive formulation perform compared to fully-automated Web services restructuring techniques?
- RQ3: Can our approach be useful for the users of Web services (the developers of service-based systems)?

To answer these research questions, we considered the best interface design restructuring solutions recommended by our approach. To answer RQ1, it is important to validate the proposed modularization solutions on the different Web services highlighted in Table 18. We asked a group of developers, as detailed in the next section, to manually modularize the design of the different interfaces considered in our experiments. Then, we calculated precision and recall scores to compare between the generated design and the expected one:

$$PR_{precision} = \frac{\text{suggested portTypes} \cap \text{expected portTypes}}{\text{suggested portTypes}} \in [0,1]$$

$$RC_{recall} = \frac{\text{suggested portTypes} \cap \text{expected portTypes}}{\text{expected portTypes}} \in [0,1]$$

When calculating the precision and recall, we consider a two port types are similar if they contain the same operations. We divided the participants in groups to make sure that they do not use our tool on the Web services that they are asked to manually modularize.

Another metric that we considered for the quantitative evaluation is the percentage of fixed design antipatterns (*NF*) by the proposed modularization solution. The detection of design antipatterns after applying a modularization solution is performed using the detection rules of our previous work [3]. Formally, *NF* is defined as

$$NF = \frac{\#\,\text{fixed desing antipatterns}}{\#\,\text{design antipatterns}} \in [0,1]$$

For the qualitative validation, we asked groups of potential users of our Web services modularization tool to evaluate, manually, whether the suggested interface design modularizations are feasible and efficient at improving the quality of Web services interface design. We define the metric Manual Correctness (*MC*) to mean the number of meaningful Web services interface refactorings divided by the total number of recommended refactorings by our tool. *MC* is given by the following equation:

$$MC = \frac{\#\,\text{correct modularization operations}}{\#\,\text{proposed modularization operations}}$$

To answer RQ2, we compared our approach to two other existing fully-automated Web services decomposition techniques [5], [69]. Ouni et al. [69] proposed an approach to decompose Web services using graph partitioning to improve cohesion. Similarly, Athanasopoulos et al. [5] used a greedy algorithm to decompose the interface based on cohesion as well. All these existing techniques are fully-automated and do not provide any interaction with the developers to update their solutions towards a desired design. We also compared the running time *T* of the proposed algorithm comparing to fully automated techniques. Thus, we used the metrics *PR, RC, T* and *NF* to perform the comparisons.

To answer RQ3, we used a post-study questionnaire that collects the opinions of Web service developers on our tool as described in the next section. Thus, we asked these participants to use both our tool and the automated framework proposed by Ouni et al. [3] on different sets of Web services. The participants were asked to make changes, when appropriate, to the final solution of the automated approach of Ouni et al. [3]. Thus, we can check whether the interactive component of the proposed interactive approach makes a real contribution, or whether the same effect can be attained by just fixing the output of the automated remodularization approaches. We measured the time spent by the developers on using our interactive approach and the automated techniques. Then, we compared between the outcomes of the survey questions for both interactive and fully automate techniques.

Table 18 Web service statistics

Web Service Provider	#services	#operations (min, max)
FedEx	19	(13, 36)
Amazon	16	(16, 93)
Yahoo	18	(11, 41)
Ebay	12	(13, 37)
Microsoft	17	(11, 59)

We extracted a set of 82 well-known Web services from an existing benchmark [3], [5] as detailed in Table 18. All studied services are widely used in different contexts and provided by Amazon, FedEx, Ebay, Microsoft and Yahoo, five major Web service providers. We selected these Web services for our validation because they range from medium to large-sized interfaces, which have been actively developed and changed over several years. Our study involved 36 participants from the University of Michigan to use and evaluate our tool. Participants include 27 master students in Software Engineering and 9 Ph.D. students in Software Engineering. All the participants are volunteers and familiar with Web services and refactoring in general. The experience of these participants on programming ranged from 3 to 17 years. 19 out of the 36 participants are currently active programmers as well in software industry with a minimum experience of 3 years. Participants were first asked to fill out a pre-study questionnaire containing nine questions. The questionnaire helped to collect background information such as their role within the company, their programming experience, their familiarity with Web services. As part of the Software Quality Assurance graduate course, all the participants attended two lectures about Web services design quality, modularization and passed five tests to evaluate their performance to evaluate and suggest interface design modularization solutions.

As described in Table 19, we formed 6 groups. Each of the 6 groups is composed by 6 participants. summarizes the survey organization including the list of Web services and the algorithms evaluated by each of the groups. The groups were formed based on the pre-study

questionnaire and the tests result to make sure that all the groups have almost the same average skills. Consequently, each group of participants who accepted to participate in the study received a questionnaire, a manuscript guide to help them to fill the questionnaire, the tools and results to evaluate the Web services design. Since the application of remodularization solutions is a subjective process, it is normal that not all the developers have the same opinion. In our case, we considered the majority of votes to determine if suggested solutions are correct or not.

Table 19 Survey organization

Groups	Web Services
Group 1	FedEx
Group 2	Amazon
Group 3	Yahoo
Group 4	Ebay
Group 5	Microsoft, Ebay
Group 6	FedEx, Yahoo

We executed three different scenarios. In the first scenario, we asked every participant to manually modularize a set of Web services. As an outcome of the first scenario, we calculated the differences between the recommended modularizations and the expected ones (manually suggested by the users/developers). The evaluate the fixed Web services design antipatterns, we focus on the ones defined. In the second scenario, we asked the users to manually evaluate the recommended solution by our algorithm. We performed a cross-validation between the groups to avoid the computation of the *MC* metric being biased by the developer's feedback. In the third scenario, we collected their opinions of the participants based on a post-study questionnaire that will be detailed before in this section. The participants were asked to justify their evaluation of the solutions and these justifications are reviewed by the organizers of the study.

Parameter setting influences significantly the performance of a search algorithm. For this reason, for each algorithm and for each Web service, we perform a set of experiments using several

population sizes: 20, 30 and 50. We limited the interaction with the user in our approach to a maximum of 30. The stopping criterion was set to 1000 evaluations for all algorithms to ensure fairness of comparison. The other parameters' values were fixed by trial and error and are as follows: (1) crossover probability = 0.5; mutation probability = 0.2 where the probability of gene modification is 0.1. Each algorithm is executed 30 times with each configuration and then the comparison between the configurations is done using the Wilcoxon test. To achieve significant results, for each couple (algorithm, Web service), we use the trial and error method to obtain a good parameter configuration.

Since metaheuristic algorithms are stochastic optimizers, they can provide different results for the same problem instance from one run to another. For this reason, our experimental study is based on 30 independent simulation runs for each problem instance of the automated approaches and the obtained results are statistically analyzed by using the Wilcoxon rank sum test with a 95% confidence level ($\alpha = 5\%$). The latter tests the null hypothesis, H0, that the obtained results of two algorithms are samples from continuous distributions with equal medians, against the alternative that they are not, H1. The p-value of the Wilcoxon test corresponds to the probability of rejecting the null hypothesis H0 while it is true (type I error). A p-value that is less than or equal to α (≤ 0.05) means that we accept H1 and we reject H0. However, a p-value that is strictly greater than α (> 0.05) means the opposite. In fact, for each problem instance, we compute the p-value obtained by comparing existing studies results with our approach ones. In this way, we determine whether the performance difference between our technique and one of the other approaches is statistically significant or just a random result. The results presented were found to be statistically significant on 30 independent runs of the fully-automated approaches using the Wilcoxon rank sum test with a 95% confidence level ($\alpha < 5\%$) as detailed in the next sub-section.

B. Experiment Results

Results for RQ1. As described in Figure 57 and Figure 58, we found that a considerable number of proposed port types, with an average of more than 81% in terms of precision and recall on all the 82 Web services, were already suggested manually (expected refactorings) by the users (software development team). The achieved recall scores are slightly higher, in average, than the precision ones since we found that some of the port types suggested manually by developers do not exactly match the solutions provided by our approach. In addition, we found that the slight deviation with the expected port types is not related to incorrect ones but to the fact that different possible modularization solutions could be optimal.

We evaluated the ability of our approach to fix several types of interface design antipatterns and to improve the quality. Figure 59 depicts the percentage of fixed code smells (*NF*). It is higher than 82% on all the Web services, which is an acceptable score since users may not be interested to fix all the antipatterns in the interface. We reported the results of our empirical qualitative evaluation in Figure 60 (*MC*). As reported in Figure 60, most of the Web services modularization solutions recommended by our interactive approach were correct and approved by developers. On average, for the different Web services, 88% of the created port types and applied changes to the initial design are considered as correct, improve the quality, and are found to be useful by the software developers of our experiments. Thus, we found that the slight deviation with the expected design is not related to incorrect changes but to the fact that the developers have different scenarios/contexts in using the different operations.

To summarize and answer RQ1, the experimentation results con-firm that our interactive approach helps the participants to re-structure their Web service interface design efficiently by finding the relevant portTypes and improve the quality of all the 22 Web services.

Results for RQ2. Figure 57**,** Figure 58**,** Figure 59, and Figure 60 confirm the average superior performance of our interactive learning GA approach compared to the two existing fully automated Web service modularization techniques [3], [5]. Figure 60 shows that our approach provides significantly higher manual correctness results (*MC*) than all other approaches having *MC* scores respectively between 41% and 62%, on average as MC scores on the different Web services. The same observation is valid for the precision and recall as described in Figure 57 and Figure 58. The outperformance of our technique in terms of percentage of fixed defects, as described in Figure 59, can be explained by the fact that the main goal of existing studies is not to mainly fix these defects (not considered in the fitness function by the work of Ouni et al. [3]).

In conclusion, our interactive approach provides better results, on average, than all existing fully-automated Web services modularization techniques (answer to RQ2).

Results for RQ3. To further analyze the obtained results, we have also asked the participants to take a post-study questionnaire after completing the different validation and tasks using our interactive approach and the two techniques considered in our experiments. The post-study questionnaires collected the opinions of the participants about their experience in using our approach compared to fully-automated tools. The post-study questionnaire asked participants to rate their agreement on a Likert scale from 1 (complete disagreement) to 5 (complete agreement) with the following statements: (1) The interactive interface modularization recommendations using our predictive modeling approach are a desirable feature to improve the quality of Web services interface. (2) The interactive manner of recommending modularization solutions by our GA learning approach is a useful and flexible way to consider the user perspective compared to fully-automated tools.

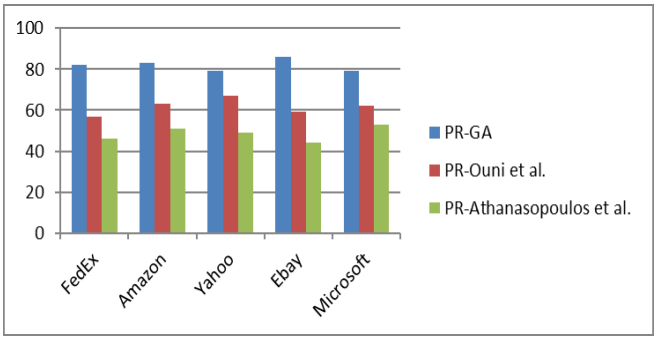


Figure 57 Median precision (PR)

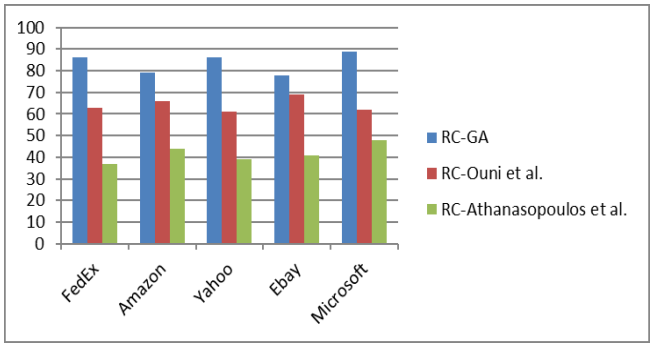


Figure 58 Median recall (RC) value

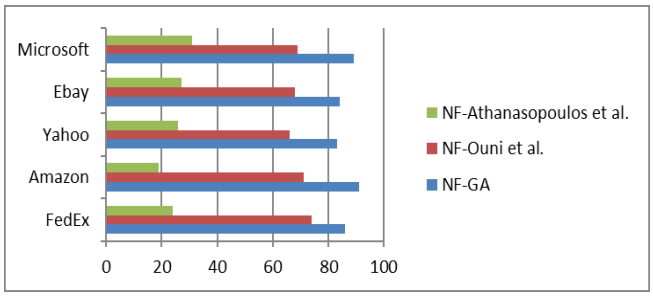


Figure 59 Median number of fixed Web service defects (NF) value

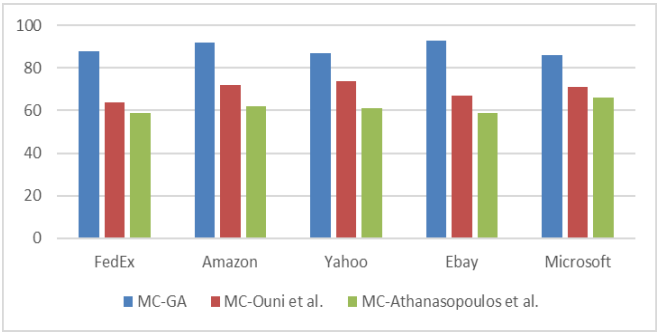


Figure 60 Median manual correctness (MC) value

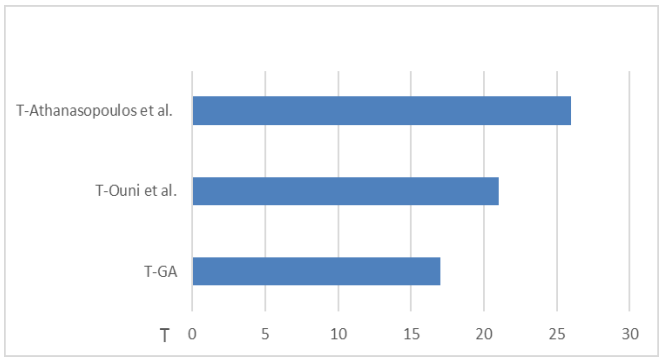


Figure 61 Median execution time (T), including user interaction

The agreement of the participants was 4.6 and 4.2 for the first and second statements respectively. This confirms the usefulness of our approach for the users of our experiments. The remaining questions of the post-study questionnaire were about the benefits and the limitations (possible improvements) of our interactive approach.

We summarize in the following the feedback of the users. Most of the participants mention that our approach is much faster and easy to use compared to the manual restructuring of the interface since they spent a long time with manual changes to create port types and move operations. Thus, the developers liked the functionality of our tool that helps them to modify a port type based on the recommendations. Some participants believe that it will be very helpful to extend the tool by adding a new feature to decompose multiple services into interfaces based on the dependency between them. Another possibly suggested improvement is to consider the users invocation data to restructure the interface.

In our evaluation, we considered measuring the time spent by the different developers to use our tool and automated Web services modularization techniques [3], [5]. We allowed the user to fix the solutions proposed by the automated tools to reach an acceptable design. Figure 61 shows the average results of the execution time of the different tools per Web service including the interaction time. The developers found that automated techniques generate solutions that require a

lot of effort to inspect and manually adjust the proposed design. All developers expressed a high interest in the idea of the interactive tool that can incorporate their preferences by evaluating manually very few solutions.

6.6 **Improving Web Services Design Quality Using Dimensionality Reduction Techniques**

6.6.1 **Introduction**

In this work, we start from the hypothesis that there may be correlations among any two or more objectives (e.g. quality metrics) that are used to evaluate Web service modularization solutions. Our approach, based on the PCA-NSGA- II methodology [40], [137], aims at finding the best and reduced set of objectives that represents the quality metrics of interest to the domain expert. A regular multi-objective NSGA-II algorithm [40] with an initial set of exhaustive metrics is executed for a number of iterations then a PCA component analyzes the correlation between the different objectives using the execution traces. The number of objectives maybe reduced during the next iterations based on the PCA results. The process is repeated several times until a maximum number of iterations is reached to generate a set of non-dominated Web services modularization solutions.

6.6.2 **Approach**

The general structure of the proposed approach is described in Figure 62. The approach takes as inputs a set of quality metrics, several Web services refactoring types, and a Web service to refactor. The first component consists of a regular execution of NSGA-II during a number of iterations. During this phase, NSGA-II [40] will try to find the non-dominated solutions balancing

the initial set containing all the objectives such as improving the quality metrics of the service and minimizing the number of refactorings in the proposed solutions.

After a number of iterations, the second component of the algorithm is executed to analyze the execution traces of the first component (solutions and their evaluations), using PCA [138], to check the correlation between the different objectives. When a correlation between two or more objectives is detected, only one of them is selected for future iterations of the first component. Then, the first component is executed again with the new objective set.

The whole process of these two components continue until a maximum num- ber of iterations is reached. A set of non-dominated refacotoring solutions are proposed to the users with the reduced objectives set to select the best Web service refactorings sequence based on his or her preferences.

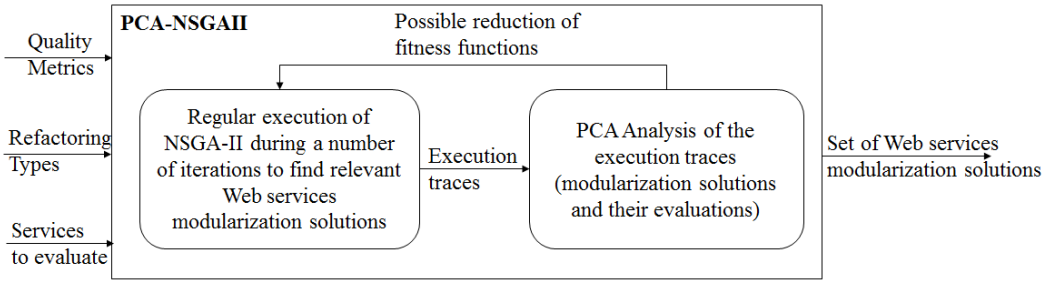


Figure 62 The proposed approach

6.6.3 **NSGA-II Adaptation**

Objective space dimensionality reduction approaches assume that given a multi-objective problem with M objectives, there is a subset of the objectives that are correlated. To the best of our knowledge, very few methodologies have been developed for multi-objective evolutionary algorithms towards the reduction of the number of objectives [46].

Saxena et al. proposed two dimensionality reduction methodologies based on Principal Component Analysis (PCA). Their methodology considers both linear and nonlinear solutions [46]. The authors demonstrated that the methodology have some vulnerabilities in finding Pareto-

optimal front in a 10-objective problem. In [137] a more robust objectives selection approach was proposed to improve the performance of both non-linear and linear dimensionality reduction. Not only these methodologies can be utilized before and after execution of the MOEA, but the computation of the PCA is straightforward for the multi-objective optimization problem. In this work, we apply the linear PCA dimensionality reduction technique to the multi/many-objective software refactoring problem using NSGA-II. In the remainder of this work, PCA refers to linear PCA unless specified otherwise.

PCA is posed as an eigenvalue-eigenvector problem: the data is recorded over a population of individuals of size N generated and used in the NSGA-II algorithm. This data consists of measurement of all the objective function used in the NSGA-II, and represented as a matrix $\mathbf{F} = \left(\mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_M\right)^T$. A column $\mathbf{f}_i = \left[f_{i1}, f_{i2}, \ldots, f_{iN}\right]^T$ is the vector representing values for the $i-$th objective over the N individuals, and each entry f_{ij} of $\mathbf{f}_i$ is the value of the i-th objective for the j-th individual in the population. In this notation, $(^T)$ is the matrix transpose operator, and M is the number of objectives.

PCA is performed using the correlation or covariance matrix of the standardized dataset $\mathbf{X} = \left(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_M\right)^T$. This means each entry $x_{ij} = (f_{ij} - \mu_i)/\sigma_i$, where μ_i and σ_i are the sample mean and standard deviation of $\mathbf{f}_i$, respectively. Consequently, every row of $\mathbf{X}$ centered at zero, and has unit standard deviation. The correlation matrix is given by equation 1, and algorithm 3.2 gives an high-level view of the objective reduction procedure.

$$\mathbf{R} = \frac{1}{M}\mathbf{X}\mathbf{X}^T$$

ALGORITHM 1: Objective Reduction High-level view

Input: $\mathbf{V}, \mathbf{\Lambda}, N_v$
Output: $\mathcal{F}_s$
$\mathcal{F}^+ = \emptyset, \mathcal{F}^- = \emptyset, \mathcal{F}_e = \emptyset;$
foreach $V_i, 1 \leq i \leq N_v$ **do**
    partition-objectives$(\mathcal{F}^+, \mathcal{F}^-)$;
    set-selected-objectives$(\mathcal{F}_e, \mathcal{F}^+, \mathcal{F}^-)$;
end
$\mathcal{F}_s = \emptyset;$
foreach $f_i \in \mathcal{F}_e$ **do**
    $\mathcal{S}_i =$ *compute-identically-correlated-subset*(f_i);
    set-selected-objectives$(\mathcal{F}_s, \mathcal{S}_i)$;
end

A. Refactoring solution representation

A solution consists of a sequence of n interface change operations assigned to a set of port types. A port type could contain one or many operations but an operation could be assigned to only one port type. A vector-based representation is used to cluster the different operations of the original interface, taken as input from the WSDL file description, into appropriate interfaces, i.e., port types. Figure 63 describes an example of 5 operations assigned to two port types. As output, a vector representation is automatically translated by our tool into a graphical interface of the modularized Web service.

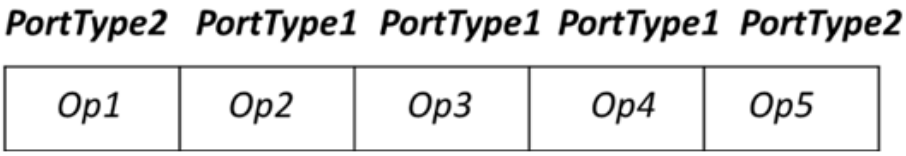


Figure 63 Solution representation example

B. Change operators

In each search algorithm, the variation operators play the key role of moving within the search space with the aim of driving the search towards optimal solutions. For the crossover, we use the one-point crossover operator. It starts by selecting and splitting at random two parent

solutions. Then, this operator creates two child solutions by putting, for the first child, the first part of the first parent with the second part of the second parent, and vice versa for the second child. It is important to note that in multi-objective optimization, it is better to create children that are close to their parents to have a more efficient search process. For mutation, we use the bit-string mutation operator that picks probabilistically one or more refactoring operations from its or their associated sequence and replaces them by other ones from the initial list of possible refactorings.

When applying the change operators, different pre- and post-conditions are checked to ensure the applicability of the newly generated solutions such as removing redundant operations or conflicts between operations such as assigning the same opera-tion to two different port types.

C. Objective functions and solution evaluation

We used the 11 quality attributes that are defined in Table 1 along with the number of refactorings in the solutions (solution size) as fitness functions of our algorithm. The initial iterations of NSGAII-PCA will use all the 12 fitness functions as input then the algorithm will reduce the number of objectives by mining the execution traces (the solutions and their evaluations).

6.6.4 **Validation**

1) Experimental Setup

We designed our experiments to address the following research questions:

- RQ1.: To what extent can the proposed dimensionality reduction approach recommends useful Web service refactorings?
- RQ2.: To what extent does the proposed dimensionality reduction approach reduce the number of objectives while recommending useful refactorings?

- RQ3.: How does the proposed dimensionality reduction approach perform compared to other existing Web services modularization techniques not based on computational search [1, 11]?

To answer **RQ1.**, we considered both automatic and manual validations to evaluate the usefulness of the proposed Web service refactorings. For the automatic validation we compared the proposed Web service refactorings with the expected ones. The expected refactorings are suggested by users (e.g. subjects of our study) to fix existing Web service design defects as detailed later.

$$RE_{recall} = \frac{|\ suggestedWebservicerefactorings \cap expectedWebservicerefactorings\ |}{|\ expectedWebservicerefactorings\ |} \in [0,1]$$

$$RE_{precision} = \frac{|\ suggestedWebservicerefactorings \cap expectedWebservicerefactorings\ |}{|\ suggestedrefactorings\ |} \in [0,1]$$

For the manual validation, we asked groups of potential users of our tool to manually evaluate whether the suggested refactorings are feasible and efficient at improving the services quality and achieving their maintainability objectives. We define the metric Manual Correctness (*MC*) that corresponds to the number of meaningful refactorings divided by the total number of suggested refactorings. MC is given by the following equation:

$$MC_{manualcorrectness} = \frac{|\ relevantWebservicerefactorings\ |}{|\ suggestedWebservicerefactorings\ |} \in [0,1]$$

We have also evaluated the ability of our approach to fix design defects, detailed in Section 2, using the measure *NF* that corresponds to the number of fixed defects divided by the total number of defects. The defects are detected using a set of rules defined in our previous work [3].

To answer **RQ2**, we compared the number of objectives (*NOB*), precision, recall and manual correctness of our approach to a regular multi-objective algorithm (NSGAII) using the same fitness functions adaptation.

To answer **RQ3**, We compared our results with a recent state-of-the art approaches by [5], [69]. Athanasopoulos et al. proposed a Web service refactoring approach based on a greedy algorithm to refactor and split Web service interfaces based on different cohesion measures. Ouni et al. proposed a graph decomposition approach for Web services remodularization using coupling and cohesion metrics.

To answer all the above research questions, we conducted our experiment on a benchmark of 22 real-world services provided by Amazon[18] and Yahoo[19]. We selected services with interfaces exposing at least 10 operations. We chose these Web services because their WSDL interfaces are publicly available, and they were previously studied in the literature [5], [57]. Table 20 presents our used benchmark.

Our evaluation involved 14 independent volunteer participants including 6 industrial developers and 8 graduate students. In particular, 3 senior developers from *Browser Kings*[20], 3 developers from *Accunet Web Services*[21], 3 MSc and 5 PhD candidates in Software Engineering. We first gathered information about the participant's background. All participants are familiar with service-oriented development and SOAP Web services with an experience ranging from 4 to 9 years. The participants were unaware of the techniques to be evaluated neither the particular research questions, in order to guarantee that there will be no bias in their judgment.

[18] http://aws.amazon.com/

[19] developer.searchmarketing.yahoo.com/docs/V6/reference/

[20] http://www.browserkings.com

[21] http://www.accunet.us

Table 20 Amazon and Yahoo benchmark overview

Service interface	Provider
AutoScalingPortType	Amazon
MechanicalTurkRequesterPortType	Amazon
AmazonFPSPorttype	Amazon
AmazonRDSv2PortType	Amazon
AmazonVPCPortType	Amazon
AmazonFWSInboundPortType	Amazon
AmazonS3	Amazon
AmazonSNSPortType	Amazon
ElasticLoadBalancingPortType	Amazon
MessageQueue	Amazon
AmazonEC2PortType	Amazon
KeywordService	Yahoo
AdGroupService	Yahoo
UserManagementService	Yahoo
TargetingService	Yahoo
AccountService	Yahoo
AdService	Yahoo
CompaignService	Yahoo
BasicReportService	Yahoo
TargetingConverterService	Yahoo
ExcludedWordsService	Yahoo
GeographicalDictionaryService	Yahoo

We performed a set of experiments using several population sizes: 30, 40 and 50. The stopping criterion was set to 100,000 fitness evaluations. Each algorithm was executed 30 times with each configuration and then comparison between the configurations was performed based on precision and recall using the Wilcoxon test with a 95% confidence level ($\alpha = 5\%$). The other parameters setting were fixed by trial and error and are as follows: (1) crossover probability = 0.4; mutation probability = 0.7 where the probability of gene modification is 0.1.

2) Experiment results

We reported the results of our empirical qualitative evaluation in Figure 64 (MC). As reported in Figure 64, most of the Web services modularization solutions recommended by our approach were correct and approved by developers. On average, for the different Web services,

78% of the created port types and applied changes to the initial design are considered as correct, improve the quality, and are found to be useful by the software developers of our experiments. The highest MC score is 84% and was achieved for the Web service GeographicalDictionary, while the lowest score was 67% for AmazonVPCPortType. Thus, this finding indicates that the results are independent of the size of the Web services and the number of recommended changes to the initial design.

Since the manual correctness MC metric just evaluates the correctness and not the relevance of the recommended solutions, we also compared the proposed modularization changes with some expected ones defined manually by the different groups for the different Web services. Figure 65 and Figure 66 summarize our findings. We found that a considerable number of proposed port types, with an average of more than 76% in terms of precision and recall, were already created by the users manually (expected port types). The recall scores are higher than precision ones since we found that the port types sug-gested manually by developers could be further decomposed, if necessary. This was confirmed by the qualitative evaluation (MC). In addition, we found that the slight deviation with the expected design is not related to incorrect changes but to the fact that the developers have different scenarios/contexts in using the different operations.

We evaluated also the ability of our approach to fix several types of design defects and to improve the service interface design quality as described in Figure 67 that depicts the percentage of fixed defects (NF). It is higher than 77% on all the 22 Web services, which is an acceptable score since developers may reject or modify some design changes that fix some de-fects because they do not consider some of them as very important (their goal is not to fix all design defects in the Web service interface) or because they wanted to focus on improving the cohesion and

minimize coupling. Some Web service interfaces, such as AmazonFWSInboundPortType, have a higher percentage of fixed code smells with an average of more than 83%.

To summarize and answer RQ1, the experimentation results confirm that our approach helps the participants to restructure their Web service interface design efficiently by finding the relevant portTypes and improve the quality of all the 22 Web services.

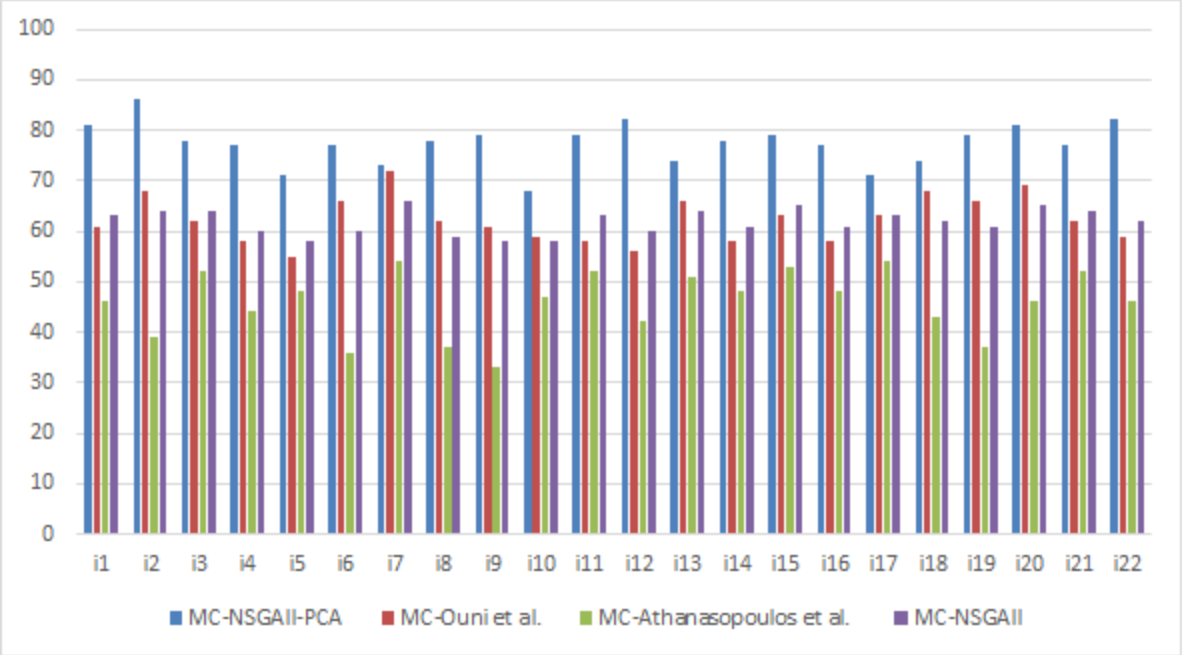


Figure 64 Median manual correctness value

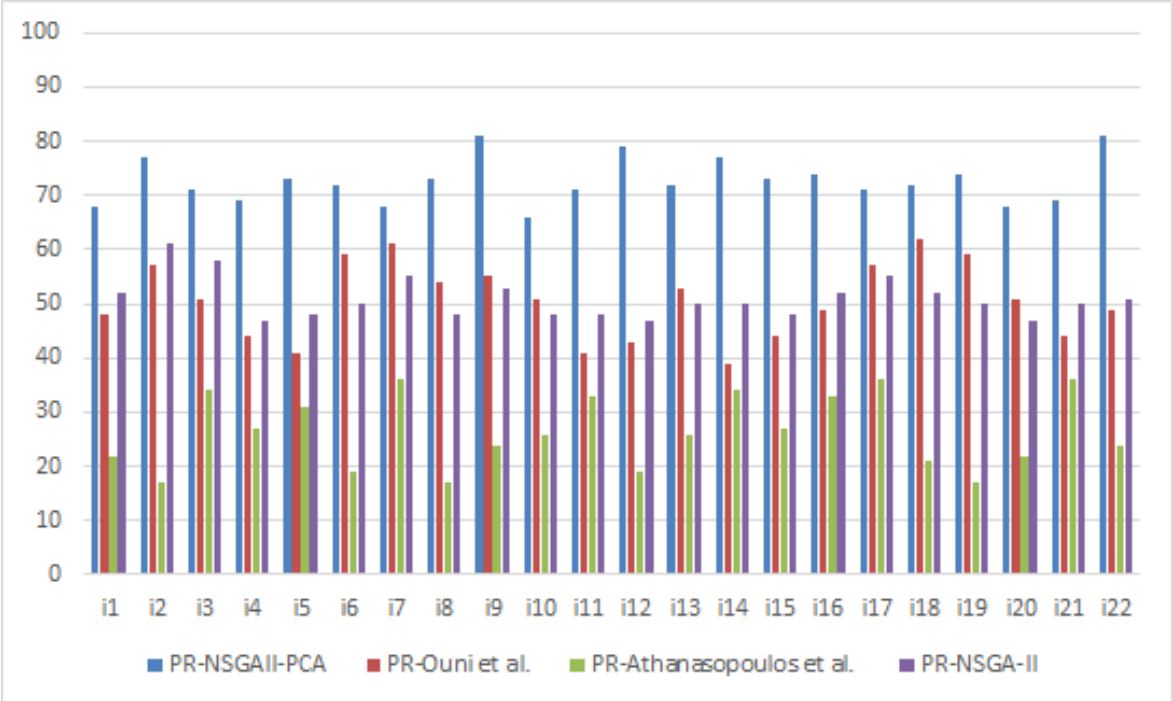


Figure 65 Median precision value over 30 runs

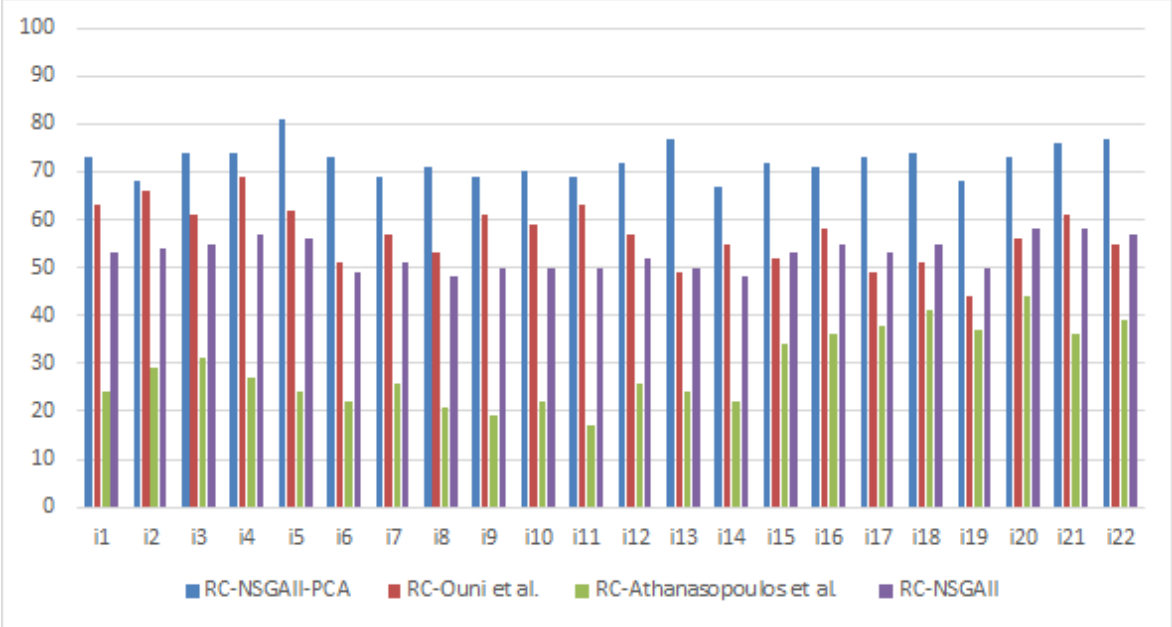


Figure 66 Median recall value over 30 runs

Results for RQ2. Figure 68shows that our approach significantly reduced the number of objectives when executed on all the systems. The number of objectives were reduced to only four

in several services. The reduced objectives may show the importance of coupling and cohesion when identifying refactoring recommendations since they were identified in all the 22 services after the reduction of objectives.The number of changes was also selected for all the services after the reduction step. Combined with the results of RQ1, it is clear that the proposed NSGAII-PCA formulation successfully reduced the number of objectives while generating useful Web services refactoring recommendations.

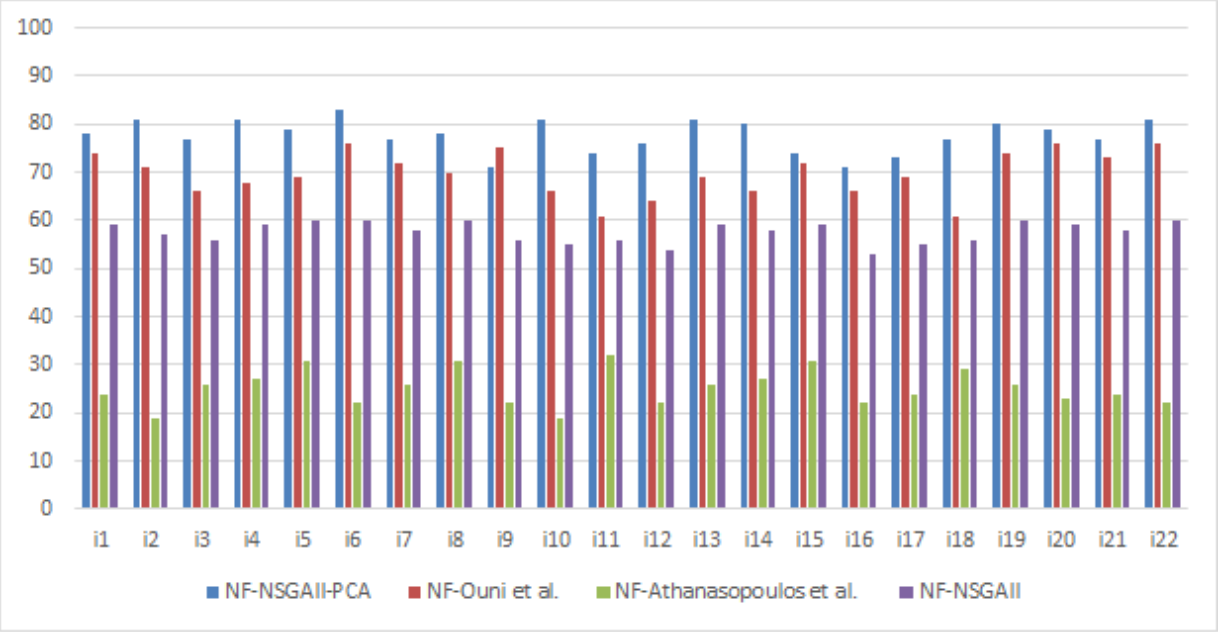


Figure 67 Median number of fixed design defects value

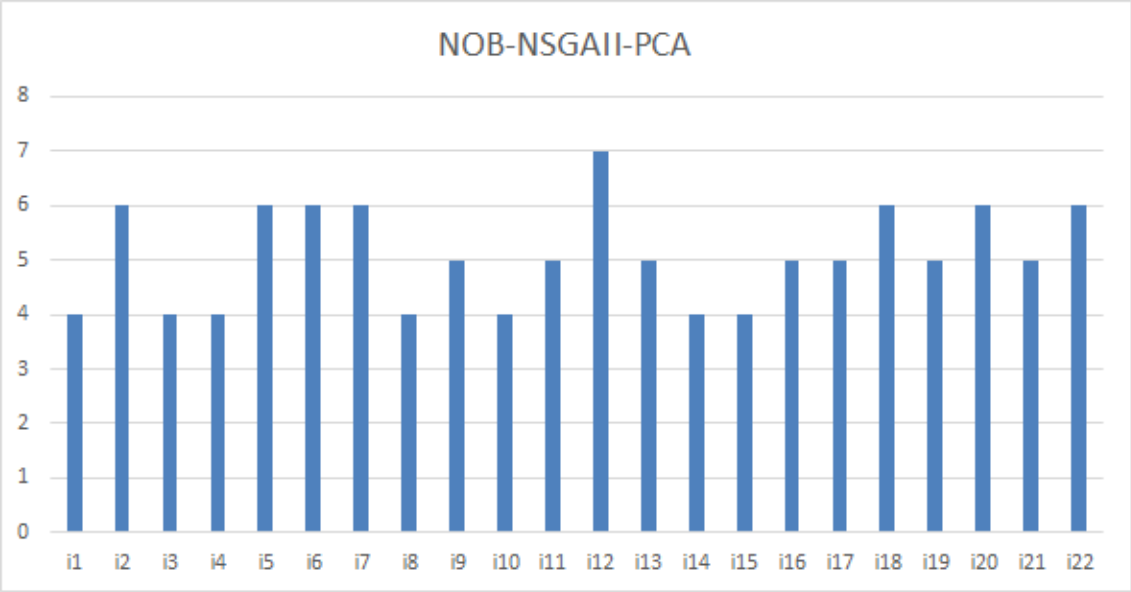


Figure 68 Median number of objectives value over 30 runs

Results for RQ3. Figure 64, Figure 65, Figure 66, and Figure 67 confirm the average superior performance of our approach compared to the two existing fully automated Web service modularization techniques [1, 11] and also the multi-objective approach combining all the metrics together without the use of the PCA component. Figure 64 shows that our approach provides significantly higher manual correctness results (MC) than all other approaches having MC scores respectively between 48% and 64%, on average as MC scores on the different Web services. The same observation is valid for the precision and recall as described in Figure 65, Figure 66. The outperformance of our technique in terms of percentage of fixed defects, as described in Figure 67, can be explained by the fact that the main goal of existing studies is not to mainly fix these defects. Existing work are mainly limited to the coupling and cohesion metrisc which may not be sufficient to guide the modularization of Web services. In conclusion, our approach provides better results, on average, than all existing fully-automated Web services modularization techniques (answer to RQ3).

3) Threats to Validity

Threats to Validity. We identify, in the following, several factors that may affect the validity of our study. A possible threat to construct validity can be related to the set of ground truth to calculate precision and recall with refactorings performed manually by developers. A possible threat to construct validity can be related to the set of ground truth to calculate precision and recall with refactorings performed manually by developers. An external threat can be related to the studied services. Although we used 22 real-world Web services provided by Amazon and Yahoo, from different application domains and ranging from 10 to 87 operations, we cannot generalize our results to other services and other technologies, e.g., REST services. As part of our future work, we plan to test our approach with an extended benchmark of Web services.

An internal threats to validity can be related to the knowledge and expertise of the human evaluators. Inadequate knowledge could lead to limited ability to assess the quality of an interface. We mitigate this threat by selecting participants having from 4 to 9 years experience with service-oriented development and familiar with SOAP Web services. Moreover, to avoid bias in the experiment none of the authors have been involved in this evaluation. In addition, we randomized the ordering in which the MOWSIR , Athanasopoulos et al. and random refactorings were shown to participants, to mitigate any sort of learning or fatigue effect.

6.6.5 **Conclusion**

In this work, we proposed a dimensionality reduction approach for multi-objective Web services remodularization that adjusts the number of considered objectives during the search for near optimal solutions. The execution traces of the multi-objective algorithm are analyzed using a PCA component to find potential correlation between the objectives (e.g. quality metrics). To evaluate the effectiveness of our tool, we conducted a human study on a set of users who evaluated the tool and compared it with the state-of-the-art Web services modularization techniques. Our

evaluation results provide strong evidence that our technique successfully reduced the initial set of large number of objectives/quality metrics. The results also show that our approach outperforms several of existing Web services modularization techniques, not based on heuristic search.

6.7 **Interactive Design of Web Services Interface Refactoring**

6.7.1 **Introduction**

The decision on how to decompose/modularize an interface is subjective and difficult to automate since it is required to integrate the feedback of users during the modularization process. In addition, the history of interactions between the users and the current Web service interface could be important to understand the dependency between the operations and generate a well-designed interface [1]. However, these aspects related to the users' feedback, when improving the quality of services interface, were not considered by existing studies.

In this work, we propose a recommendation approach that dynamically adapts and interactively suggests a possible modularization, also called refactoring [27], of the Web services interface to developers and takes their feedback into consideration. Our approach uses an interactive multi-criteria decision-making algorithm, based on interactive non-dominated sorting genetic algorithm (NSGA-II) [105],to find a set of good design interface modularization solutions that provide a trade-off between (1) improving several interface design quality metrics (e.g. coupling, cohesion, number of portTypes and number of antipatterns), (2) maximizing the satisfaction of the interaction constraints learnt from the user feedback during the execution of the algorithm, while (3) minimizing the deviation from the initial design. To find a trade-off between these different conflicting objectives, there is no single possible modularization solution but a set of optimal, i.e., non-dominated, solutions, so-called Pareto front [40]. The challenge at this step is how to choose one solution from this front to present to the Web service's user or developer? The

traditional approach is to seek a 'knee point' [40] from the front that presents the maximum trade-off between the different objectives. However, this may ignore the preferences of the user. To address this issue, we propose to analyze and explore the Pareto front of possible remodularization solutions interactively and implicitly with the developer.

Our algorithm starts by finding the most frequently-occurring remodularization operations among the set of non-dominated solutions. Based on this analysis, a complete interface remodularization solution is chosen from the front that best matches the most frequently-occurring operations, i.e., the solution that best represents the entire front. The recommended modularization operations are then ranked and suggested to the developer one by one. The developer can approve, modify or reject each suggested modularization such as moving operations between port types, or merging/splitting port types. Each action by the developer participates to guide the search process towards a desired solution. For example, if the user rejects to apply a modularization operation, the search process will subsequently avoid reconsidering it when creating new solutions. NSGA-II will continue to execute in the new modified context to repair and evolve the set of good remodularization solutions based on the feedback received from the Web services developer.

We evaluated our approach on a set of 22 real-world Web services, provided by Amazon and Yahoo. Statistical analysis of our experiments shows that our dynamic interactive Web services interface modularization approach performed significantly better than the state-of-the-art modularization techniques [5], [69]. The primary contributions of this work can be summarized as follows:

The work introduces a novel interactive way to modularize and improve the quality of Web services using interactive dynamic multi-objective optimization. The proposed technique supports the adaptation of interface design solutions based on the user feedback while improving several

quality attributes while minimizing the deviation from the initial design. To the best of our knowledge, we propose the first approach to interactively generate a modularized Web services interface.

The work reports the results of an empirical study on an implementation of our approach. The obtained results provide evidence to support the claim that our proposal is more efficient, on average, than existing Web services modularization techniques based on a benchmark of 22 real-world services. The work also evaluates the relevance and usefulness of the suggested interface design improvements for Web service users.

6.7.2 **Approach**

A. Approach Overview

The goal of our approach is to propose a new dynamic interactive way for users to refactor their Web services interface de-sign. The general structure of our approach is sketched in Figure 69. Our technique comprises two main components. The first component consists of an offline phase. It is executed first in the background when the developer uploads the WSDL file to analyze. During this phase, the multi-objective algorithm, NSGA-II [40], is executed for several iterations to find the non-dominated solutions balancing the three following objectives:

- Objective 1 maximizes the interface design quality, which corresponds to minimize the number of design antipatterns and improve design quality metrics (coupling and cohesion),
- Objective 2 maximizes the satisfaction of the constraints learnt from the user interaction,
- Objective 3 minimizes the number of introduced changes to modify the Web service design and port types.

The output of this first step of the offline phase is a set of Web services remodularization solutions that optimize the above three objectives. As explained in Algorithms 1 and 2, the second

step of the offline phase explores this Pareto front in an intelligent manner using our algorithm to rank recommended changes based on the common features between the non-dominated solutions. In our adaptation, we assume true the hypothesis that the most frequently occurring remodularization operations in the non-dominated solutions are the most relevant ones for developers and can fix several antipattern types. Thus, the output of this second step of the offline phase is a set of ranked solutions based on this frequency score.

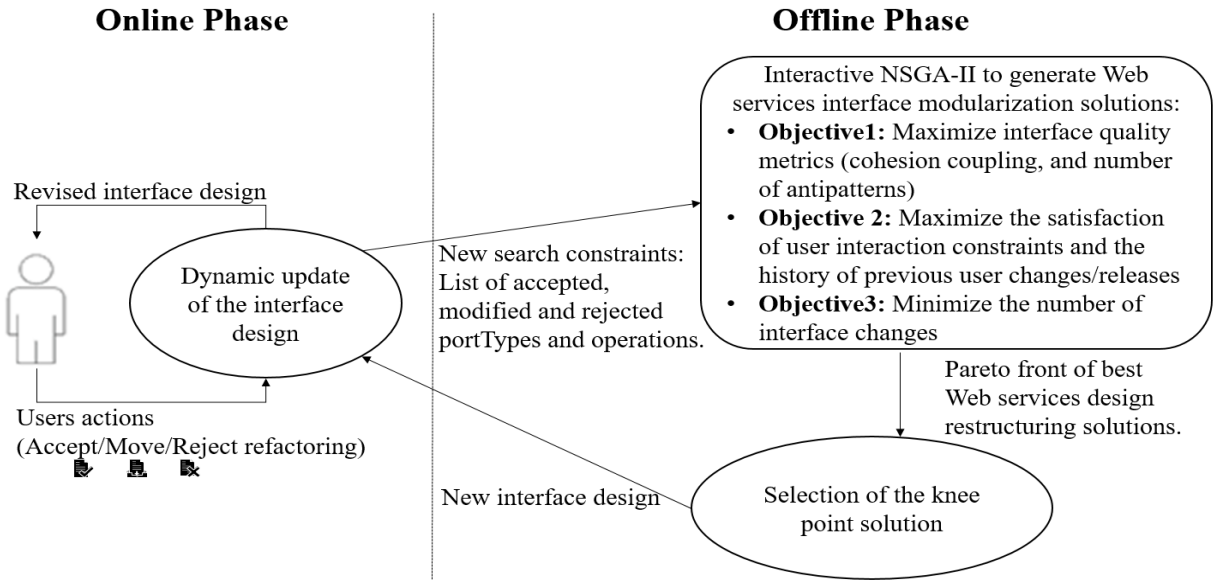


Figure 69 Approach overview

The second component of our approach is an online phase to manage the interaction with the user. It dynamically updates the list of interaction constraints based on the feedback of the developer. This feedback can be to accept/apply or modify or reject some of the suggested design changes. Thus, the goal is to guide, implicitly, the exploration of the search space of possible Web services modularization solutions. Since the interactions constraints are updated dynamically, our interactive algorithm allows the implicit move between non-dominated solutions of the Pareto front. The list of constraints that could be learnt will be discussed in the next section. For example, when a user accepts a port type then the operations of that port type should stay together in the

next interactions of the algorithm, but new operations could be moved to that port type. Another interaction option for the user is to specify desired values of the different metrics then the multi-objective algorithm will try to restructure the design of the interface to reach these desired values. The interaction algorithm will be explained later in more details.

After several interactions, users may have modified or rejected a high number of suggested design changes or have introduced several new changes manually. Whenever the users stop the Web service design modularization session by closing the suggestions window, the first component of our approach is executed again on the background to update the last set of non-dominated modularization solutions by continuing the execution of NSGA-II based on the three objectives defined in the first component as described in Algorithm 1 and the new constraints summarizing the feedback of the user. In fact, we consider the rejected port types or operations by the developer as constraints to avoid generating solutions containing similar port types in the next iterations to avoid putting together again the operations of that rejected port types in the next iterations of the algorithm. This may lead to reducing the search space and thus a fast convergence to better interface modularization solutions. Of course, the next iterations of NSGA-II take as input the updated version of the interface after the interactions with users. The whole process continues until the developers decide that there is no necessity to restructure the Web service anymore. The outcome of the proposed approach that consists of the modularization of the Web service interface should have an impact on the implementation of the operations as well. In fact, the operations that are grouped together into one sub-interface may give an indication that they should be implemented within the same module. Thus, the proposed interface modularization could help the services developer to improve the cohesion and coupling of their implementation of services operation.

B. Interactive NSGA-II

Most real world optimization problems encountered in practice involve multiple criteria to be considered simultaneously. These criteria, also called objectives, are often conflicting. Usually, there is no single solution that is optimal with respect to all these objectives at the same time, but rather many different designs exist which are incomparable per se. Consequently, contrary to Single-objective Optimization Problems (SOPs) where we look for the solution presenting the best performance, the resolution of a multi-objective optimization (MOP) yields a set of compromise solutions presenting the optimal trade-offs between the different objectives. When plotted in the objective space, the set of compromise solutions is called the Pareto front. The resolution of a MOP yields a set of trade-off solutions, called Pareto optimal solutions or non-dominated solutions, and the image of this set in the objective space is called the Pareto front. Hence, the resolution of a MOP consists in approximating the whole Pareto front.

In this work, we adapted one of the widely used multi-objective search algorithms called NSGA-II and integrated our interactive component to it. NSGA-II is a powerful search method stimulated by natural selection that is inspired from the theory of Darwin. Hence, the basic idea of NSGA-II is to make a population of candidate solutions evolve toward the near-optimal solution in order to solve a multi-objective optimization problem. NSGA-II is designed to find a set of optimal solutions, called non-dominated solutions, also Pareto set. A non-dominated solution is the one which provides a suitable compromise between all objectives without degrading any of them. As described in Algorithm 1, the first step in NSGA-II is to create randomly a population P_0 of individuals encoded using a specific representation. Then, a child population Q_0 is generated from the population of parents P_0 using genetic operators such as crossover and mutation. Both populations are merged into an initial population R_0 of size N. As a consequence, NSGA-II starts

by generating an initial population based on a specific representation that will be discussed later, using the exhaustive list of interface operations given as input as mentioned in the previous section. Thus, this population stands for a set of possible solutions represented as sequences of portTypes (including the operations) which are selected and combined. After a number of iterations, the best solution (interface design modularization) will be presented to the user to get his feedback then the algorithm will continue to execute taking into consideration the new learnt interaction constraints.

To summarize, the main NSGA-II loop goal is to make a population of candidate solutions evolve toward the best clustering of interface operations into portTypes, i.e., the sequence that minimizes the coupling, number of antipatterns, number of portTypes and number of interface changes, and maximizes the cohesion and the satisfaction of the interaction constraints. During each iteration t, an offspring population Q_t is generated from a parent population P_t using genetic operators (selection, crossover and mutation). Then, Q_t and P_t are assembled to create a global population R_t. Then, each solution S_i in the population R_t is evaluated using our three fitness functions. We describe in the next sections, the different steps of adaption of the interactive NSGA-II algorithm to our problem.

C. Fitness Function

The generated solutions are evaluated using three fitness functions as detailed in the following.

Objective 1: Maximize the Web services design quality metrics. This fitness function is defined as the average of three measures. The first measure is the number of design antipatterns that can be detected using the rules defined in our previous work in Chapter 3. The second measure is the cohesion that corresponds to the degree to which the operations exposed in a service interface

conceptually belong together. We used, in this work, the definition of cohesion described before which is based on communicational and textual similarities between the operations within the same port type based on cosine similarity and call-graphs. The third measure is coupling within a service measures the relationships between implementation elements belonging to the same service . Service interface coupling is a measure of how strongly a service interface is connected to or relies on other service interfaces. We used the existing definition of coupling based on the similarity between the operations within the same port type and the number of calls to other operations in different port types. The reason of not treating quality objectives separately are related to redcuing the execution time and the number of non-dominated solutions (especially for an interactive approach), and also the performance of NSGA-II when the number of objectives becomes high.

$$f_1 = \frac{\#antipatterns\_After\_Modularization}{\#antipatterns\_Before\_Modularization} + Cohesion - Coupling$$

Objective 2: Maximize the interaction-based function. This function maximizes the satisfaction of the constraints learnt from the interaction with user or minimizes the distance with the desired metrics, if specified by the user as described in Figure 69. In case that the user did not specify these desired values then we just ignore this component of the fitness function. Furthermore, the user has four other types of interaction, as described in Figures 5 and 6, that correspond to accept a portType, reject a portType, move operation(s) and delete operation(s). Each of these user actions will generate a set of constraints for the exploration of the search space. When a port type is accepted, the list of operations in that port type should stay together in the next iterations but new operations could be added to the port type. When a port type is rejected by the user, a constraint is generated to avoid regrouping together again these operations into the same port type. The application of a move operation action will generate a constraint to keep the moved operation in the targeted port type in the next iterations. When an operation is deleted, a constraint

will be generated to avoid putting again that operation in the source port type in the next iterations. Formally, the second fitness function to minimize is defined as follows:

$$Rank(S_i) = \sum_{k=1} Rank(R_{k,i}) + (RO \cap AppliedRefactoringsList) \\ -(RO \cap RejectedRefactoringsList) + 0.5 * (RO \cap ModifiedRefactoringsList)$$

$$f_2 = \frac{1}{\sum_{i=1}^{k} |MDesired_i - M_i|} + \frac{\#Statisfied\_Histroy\_Constraints}{\#Constraints}$$

This second fitness function is composed by two components. The first component is to minimize the distance between the desired metrics value specified by the user (e.g. coupling, cohesion, number of portTypes, etc.) and the actual values of the solution to evaluate. The second component is to maximize the number of satisfied interaction constraints over the total number of learnt constraints.

Objective 3: Minimize the number of changes comparing to the initial design. The designer may have some preferences regarding the degree of the deviation with the initial design of the interface. Thus, we formally define the fitness function as the following:

$$f_3 = \#designChanges$$

The number of design changes is calculated based on the number of differences between the two vector representations of the initial design and the generated one, i.e. the number of operations of the new design assigned to different port types compared to the initial design.

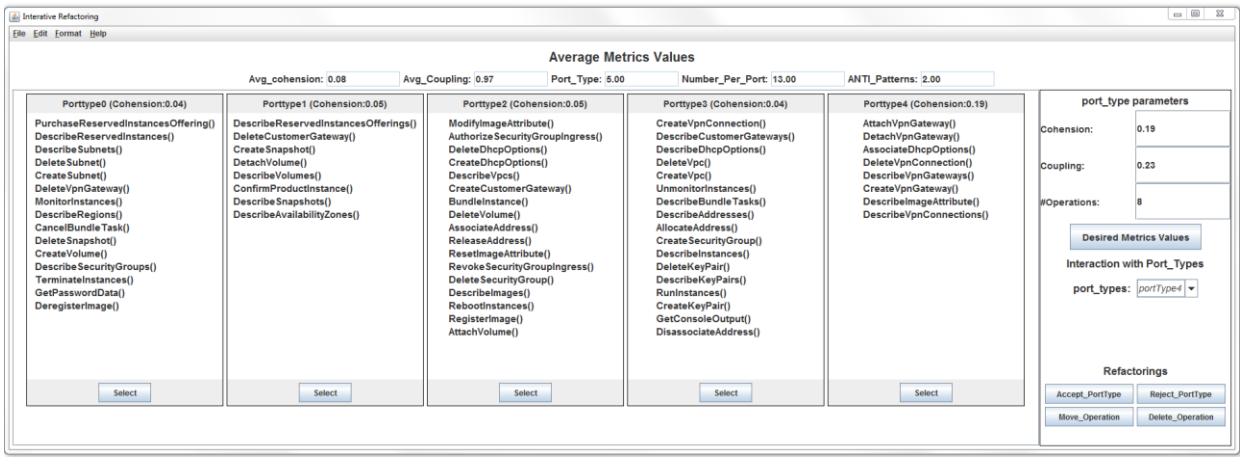


Figure 70 The proposed Web services design modularization tool

D. Interactive Recommendations

The first step of the interactive component is executed as described in Algorithm, to investigate if there are some common patterns among the generated non-dominated refactoring solutions. The algorithm checks if the optimal refactoring solutions have some common features such as similar refactoring operations among most or all the solutions, and a specific common order/sequence in which to apply the refactorings. Such information will be used to rank the suggested refactorings for developers using the following formula:

$$Rank(R_{x,y}) = \frac{\sum_{j=0}^{n} \sum_{i=0}^{size(Sj)} [Ri,j{=}Rx,y]}{MAX(\sum_{j=0}^{n} \sum_{i=0}^{size(Sj)} [Ri,j{=}Rx,y])} \in [0..1]$$

Algorithm. The ranking procedure to manage the interactions with the developer (online phase)

Input
RNS: Ranked Non-dominated SolutionSet
Output
M: Map of refactorings along with their occurrences.
Begin
Applied-Refactorings ← Ø;
Rejected-Refactorings ← Ø;
For *i=1* **to** |*RNS*| **do**
  ref[i] ← 0;
End for
/* Main loop to suggest refactorings one by one to the user*/
While |Rejected-Refactorings|< α **do**
/* Select index of the the solution with highest rank*/
  index ← Max-Rank(*RNS*);

```
  d ← User-Decision(RNS_index,ref[index]);
/* If the user has applied or modified the operation*/
  If (d = True) then
      Applied-Refactorings ← Applied-Refactorings ∪ RNS_index,ref[index];
/* If the user has rejected the operation*/
  else
      Rejected-Refactorings ← Rejected-Refactorings ∪ RNS_index,ref[index];
  End if
  ref[index] ← ref[index] + 1;
/* Update solutions indexes */
  For i=1 to |RNS| do
    Update-Rank(RNS_i, Applied-Refactorings,Rejected-Refactorings)
End While
End
```

where Rx,y is the refactoring operation number x (index in the solution vector) of solution number y, and n is the number of solutions in the front. Si is the solution of index i. All the solutions of the Pareto front are ranked based on the score of this measure applied to every solution.

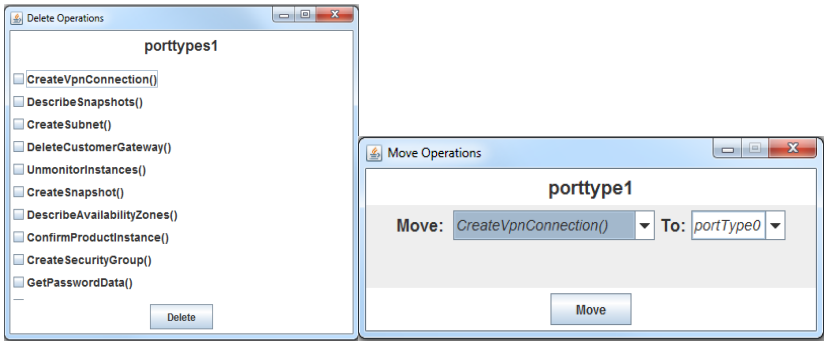


Figure 71 User interactions

Once all Pareto front solutions are ranked, the second step of the interactive process is executed as described in Figure 70. The refactorings of the best solution, in terms of ranking, are recommended to the developer based on their order in the vector. Then, the ranking score of the solutions is updated automatically after every feedback (interaction) with the developer. Our interactive algorithm proposes three levels of interaction as described in Figure 70. The developer can check the ranked list of refactorings and then apply, modify or reject the refactoring. If the developer prefers to modify the refactoring, then our algorithm can help them during the modification process as described in Figure 71. In fact, our tool proposes to the developer a set of

recommendations to modify the refactoring based on the history of changes applied in the past and the semantic similarity between the port types and operations. For example, if the developer wants to modify a move operation refactoring then, having specified the source port type to move, our interactive algorithm automatically suggests a list of possible target port types ranked based on the history of changes and semantic similarity. This is an interesting feature of our approach since developers often know which operation to move, but find it hard to determine a suitable target port type. The same observation is valid for the remaining refactoring types. Another action that the developers can select is to reject/delete a refactoring from the list. After every action selected by the developer, the ranking is updated based on the feedback using the following formula:

Where Si is the solution to be ranked, the first component consists of the sum of the ranks of its operations as explained previously and the second component will take the value of 1 if the recommended refactoring operation was applied by the developer, or -1 if the refactoring operation was rejected or 0.5 if it was partially modified by the developer. We selected 0.5 as a threshold since most of the operations have very few parameters (up-to two parameters) that could be modified. The recommended refactorings will be adjusted based on the updated ranking score.

It is important to note that we calculate the ranking score for each non-dominated solution using our ranking measure and then the solution with the highest score is presented refactoring by refactoring to the developer. In fact, refactorings tend to be dependent on one another, thus it is important to ensure the coherence of the recommended solution. After several modified or rejected refactorings, the generated Pareto front of refactoring solutions by NSGA-II needs to be updated since the original interface was modified. Thus, the ranking of the solutions will change after every interaction. If many refactorings are rejected, the NSGA-II algorithm will continue to execute while taking into consideration all the feedback from developers as constraints to satisfy during

the search. The rejected refactorings should not be considered as part of the newly generated solutions and the new Web service interfaces after refactoring will be considered in the input of the next iteration of the NSGA-II.

In a non-interactive Web services refactoring approach, the set of refactorings, suggested by the best-chosen solution, needs to be fully executed to reach the solution's promised results. Thus, any changes applied to the set of refactorings such as changing or skipping some of them could deteriorate the resulting design quality. In this context, the goal of this work is to cope with the above-mentioned limitation by granting to the developer's the possibility to customize the set of suggested refactorings either by accepting, modifying or rejecting them. The novelty of this work is the approach's ability to consider the developer's interaction, in terms of introduced customization to the existing solution, by conducting a local search to locate a new solution in the Pareto Front that is nearest to the newly introduced changes. We believe that our approach may narrow the gap that exists between automated and manual Web services refactoring techniques. It allows the developer to select the refactorings that best match his/her design preferences.

6.7.3 **Validation**

1) Experimental Setup

We defined three research questions that address the applicability, performance in comparison to existing fully-automated Web services modularization approaches [5], [69], and the usefulness of our interactive multi-objective approach. The three research questions are as follows:

- RQ1: To what extent can our approach recommend relevant Web services design improvements?
- RQ2: How does our interactive formulation perform compared to fully-automated Web services restructuring techniques [5], [69]?

- RQ3: Can our approach be useful for the users of Web services (the developers of service-based systems)?

To answer these research questions, we considered the best interface design restructuring solutions recommended by our approach after interactions with the developers as described in the previous section. To answer RQ1, it is important to validate the proposed modularization solutions on the different Web services highlighted in Table 22. We asked a group of developers, as detailed in the next section, to manually modularize the design of the different interfaces considered in our experiments. Then, we calculated precision and recall scores to compare between the generated design and the expected one:

$$\mathrm{RC}recall = \frac{\mathrm{suggested\_portTypes} \cap \mathrm{expected\_portTypes}}{\mathrm{expected\_portTypes}} \in [0, 1]$$

$$\mathrm{PR}precision = \frac{\mathrm{suggested\_portTypes} \cap \mathrm{expected\_portTypes}}{\mathrm{suggested\_portTypes}} \in [0, 1]$$

When calculating the precision and recall, we consider a two port types are similar if they contain the same operations. We divided the participants in groups to make sure that they do not use our tool on the Web services that they are asked to manually modularize.

Another metric that we considered for the quantitative evaluation is the percentage of fixed design antipatterns (*NF*) by the proposed modularization solution. The detection of design antipatterns after applying a modularization solution. Formally, *NF* is defined as:

$$NF = \frac{\#\mathrm{fixeddesingantipatterns}}{\#\mathrm{designantipatterns}} \in [0, 1]$$

For the qualitative validation, we asked groups of potential users of our Web services refactoring tool to evaluate, manually, whether the suggested interface design refactorings are feasible and efficient at improving the quality of Web services interface design. We define the metric Manual Correctness (*MC*) to mean the number of meaningful refactorings divided by the

total number of recommended refactorings by our tool. The *MC* metric is computed after the user interaction is completed. In fact, the number of correct refactorings includes the number of design refactorings applied by developers when using our tool, since they can either apply, modify or reject a refactoring recommendation (e.g. created port type). *MC* is given by the following equation:

$$MC = \frac{\#\text{coherentappliedmodularizationoperations}}{\#\text{proposedmodularizationoperations}}$$

To avoid the computation of the *MC* metric being biased by the developer's feedback, we asked the developers to manually evaluate the correctness of the recommended refactorings on the Web services that they did not refactor using our tool.

We considered also some other useful metrics to answer RQ1 that count the percentage of Web service refactorings that were accepted (*NAC*) or rejected (*NRE*) or applied with some modifications (*NMO*). Formally, these metrics are defined as:

$$NRE = \frac{\#\text{rejectedmodularizationoperations}}{\#\text{recommendedmodularizationoperations}} \in [0,1]$$

$$NMO = \frac{\#\text{modifiedmodularizationoperations}}{\#\text{recommendedmodularizationoperations}} \in [0,1]$$

$$NAC = \frac{\#\text{acceptedmodularizationoperations}}{\#\text{recommendedmodularizationoperations}} \in [0,1]$$

To answer RQ2, we compared our approach to two other existing fully-automated Web services decomposition techniques [5], [69]. Ouni et al. [69] proposed an approach to decompose Web services using graph partitioning to improve cohesion. Similarly, Athanasopoulos et al. [5] used a greedy algorithm to decompose the interface based on cohesion as well. All these existing techniques are fully-automated and do not provide any interaction with the developers to update their solutions towards a desired design. Thus, we used the metrics *PR, RC,* and *NF* to perform the comparisons.

To answer RQ3, we used a post-study questionnaire that collects the opinions of Web service developers on our tool as described in the next section. Thus, we asked these participants to use both our interactive tool and the automated framework proposed by Ouni et al. [5] on different sets of Web services. The participants were asked to make changes, when appropriate, to the final solution of the automated approach of Ouni et al. [5]. Thus, we can check whether the "online phase" of the proposed interactive approach makes a real contribution, or whether the same effect can be attained by just fixing the output of the automated remodularization approaches. Then, we compared between the outcomes of the survey questions for both interactive and fully automate techniques.

Table 21 Studied Web service interfaces

Service interface	Provider	#operations
AutoScalingPortType	Amazon	13
MechanicalTurkRequesterPortType	Amazon	27
AmazonFPSPorttype	Amazon	27
AmazonRDSv2PortType	Amazon	23
AmazonVPCPortType	Amazon	21
AmazonFWSInboundPortType	Amazon	18
AmazonS3	Amazon	16
AmazonSNSPortType	Amazon	13
ElasticLoadBalancingPortType	Amazon	13
MessageQueue	Amazon	13
AmazonEC2PortType	Amazon	87
KeywordService	Yahoo	34
AdGroupService	Yahoo	28
UserManagementService	Yahoo	28
TargetingService	Yahoo	23
AccountService	Yahoo	20
AdService	Yahoo	20
CompaignService	Yahoo	19
BasicReportService	Yahoo	12
TargetingConverterService	Yahoo	12
ExcludedWordsService	Yahoo	10

We used a benchmark of 22 well-known Web services as detailed in Table 21. All studied services are widely used in different contexts and provided by Amazon and Yahoo, two major Web service providers. We selected these Web services for our validation because they range from medium to large-sized projects, which have been actively developed and changed over several years. Our study involved 24 participants from the University of Michigan to use and evaluate our tool. Participants include 16 master students in Software Engineering and 8 Ph.D. students in Software Engineering. All the participants are volunteers and familiar with Web services and refactoring in general. The experience of these participants on programming ranged from 2 to 19 years. 11 out of the 24 participants are currently active programmers as well in software industry with a minimum experience of 2 years. Participants were first asked to fill out a pre-study questionnaire containing twelve questions. The questionnaire helped to collect background information such as their role within the company, their programming experience, their familiarity with Web services. In addition, all the participants attended one lecture about Web services design quality, modularization and passed five tests to evaluate their performance to evaluate and suggest interface design modularization solutions.

As described in Table 22, we formed 4 groups. Each of the four groups is composed by 6 participants. Table 22 summarizes the survey organization including the list of Web services and the algorithms evaluated by each of the groups. The groups were formed based on the pre-study questionnaire and the tests result to make sure that all the groups have almost the same average skills. Consequently, each group of participants who accepted to participate in the study received a questionnaire, a manuscript guide to help them to fill the questionnaire, the tools and results to evaluate the Web services design. Since the application of remodularization solutions is a subjective process, it is normal that not all the developers have the same opinion. In our case, we

considered the majority of votes to determine if suggested solutions are correct or not. We performed a cross-validation between the groups to avoid the evaluation will be biased by the developer's feedback. Thus, the subjects within the same group evaluated only the desiring obtained with the feedback of individual of other groups.

Table 22 Survey organization

Groups	Web Services	Algorithms / Approaches
Group 1	i1-i5	Interactive approach Ouni et al. [4] Athanasopoulos et al. [5]
Group 2	i6-i10	
Group 3	i11-i16	
Group 4	i17-i22	

We executed three different scenarios. In the first scenario, we asked every participant to manually modularize a set of Web services. As an outcome of the first scenario, we calculated the differences between the recommended modularizations and the expected ones (manually suggested by the users/developers). To evaluate the fixed Web services design antipatterns, we focus on the ones defined. In the second scenario, we asked the users to manually evaluate the last recommended solution by our algorithm after the interaction with the user. We performed a cross-validation between the groups to avoid the computation of the *MC* metric being biased by the developer's feedback. In the third scenario, we collected their opinions of the participants based on a post-study questionnaire that will be detailed before in this section. The participants were asked to justify their evaluation of the solutions and these justifications are reviewed by the organizers of the study.

Parameter setting influences significantly the performance of a search algorithm. For this reason, for each algorithm and for each Web service, we perform a set of experiments using several population sizes: 20, 30, 50, 100 and 200. The stopping criterion was set to 50,000 evaluations for all algorithms to ensure fairness of comparison. The other parameters' values were fixed by trial

and error and are as follows: (1) crossover probability = 0.6; mutation probability = 0.3 where the probability of gene modification is 0.2; stopping criterion = 50,000 evaluations. Each algorithm is executed 30 times with each configuration and then the comparison between the configurations is done using the Wilcoxon test. To achieve significant results, for each couple (algorithm, Web service), we use the trial and error method to obtain a good parameter configuration.

Since metaheuristic algorithms are stochastic optimizers, they can provide different results for the same problem instance from one run to another. For this reason, our experimental study is based on 30 independent simulation runs for each problem instance and the obtained results are statistically analyzed by using the Wilcoxon rank sum test with a 95% confidence level ($\alpha = 5\%$). The latter tests the null hypothesis, H0, that the obtained results of two algorithms are samples from continuous distributions with equal medians, against the alternative that they are not, H1. The p-value of the Wilcoxon test corresponds to the probability of rejecting the null hypothesis H0 while it is true (type I error). A p-value that is less than or equal to α (≤ 0.05) means that we accept H1 and we reject H0. However, a p-value that is strictly greater than α (> 0.05) means the opposite. In fact, for each problem instance, we compute the p-value obtained by comparing existing studies. results with our approach ones. In this way, we determine whether the performance difference between our technique and one of the other approaches is statistically significant or just a random result. The results presented were found to be statistically significant on 30 independent runs using the Wilcoxon rank sum test with a 95% confidence level ($\alpha < 5\%$) as detailed in the next section.

The Wilcoxon rank sum test verifies whether the results are statistically different or not; however, it does not give any idea about the difference in magnitude. To this end, we used the Vargha-Delaney A measure which is a non-parametric effect size measure. In our context, given the different performance metrics (such as *PR, RC, NF, MC*, etc.), the A statistic measures the

probability that running an algorithm B1 (interactive NSGA-II) yields better performance than running another algorithm B2 (such as [69].). If the two algorithms are equivalent, then A = 0.5. In our experiments, we have found the following results: a) On small Web services our approach is better than all the other algorithms based on all the performance metrics with an A effect size higher than 0.91; and b) On large Web services, our approach is better than all the other algorithms with an A effect size higher than 0.84.

2) Results and Discussions

Results for RQ1. As described in Figure 73 and Figure 74, we found that a considerable number of proposed port types, with an average of more than 80% in terms of precision and recall on all the 22 Web services, were already suggested manually (expected refactorings) by the users (software development team). The achieved recall scores are slightly higher, in average, than the precision ones since we found that some of the port types suggested manually by developers do not exactly match the solutions provided by our approach. In addition, we found that the slight deviation with the expected port types is not related to incorrect ones but to the fact that different possible modularization solutions could be optimal.

We evaluated the ability of our approach to fix several types of interface design antipatterns and to improve the quality. Figure 75 depicts the percentage of fixed code smells (*NF*). It is higher than 79% on all the 22 Web services, which is an acceptable score since users may not be interested to fix all the antipatterns in the interface. Some Web services, such as *AmazonSNSPortType*, has a higher percentage of antipatterns with an average of more than 86%. This can be explained by the fact that this Web service interface includes a lower number of antipatterns than others.

We reported the results of our empirical qualitative evaluation in Figure 72 (*MC*). As reported in Figure 72, most of the Web services modularization solutions recommended by our

interactive approach were correct and approved by developers. On average, for the different Web services, 89% of the created port types and applied changes to the initial design are considered as correct, improve the quality, and are found to be useful by the software developers of our experiments. The highest MC score is 94% and was achieved for the Web service *GeographicalDictionary,* while the lowest score was 79% for *AmazonVPCPortType*. Thus, this finding indicates that the results are independent of the size of the Web services and the number of recommended changes to the initial design.

Since the manual correctness *MC* metric just evaluates the correctness and not the relevance of the recommended solutions, we also compared the proposed modularization changes with some expected ones defined manually by the different groups for the different Web services. Figure 73 and Figure 74 summarize our findings. We found that a considerable number of proposed port types, with an average of more than 84% in terms of precision and recall, were already created by the users manually (expected port types). The recall scores are higher than precision ones since we found that the port types suggested manually by developers could be further decomposed, if necessary. This was confirmed by the qualitative evaluation (MC). In addition, we found that the slight deviation with the expected design is not related to incorrect changes but to the fact that the developers have different scenarios/contexts in using the different operations.

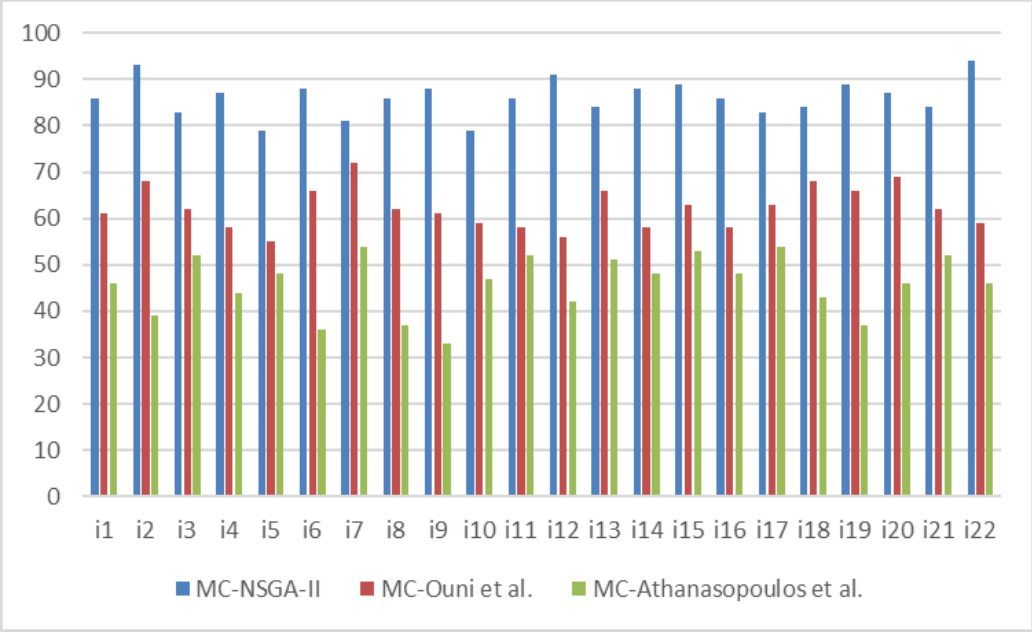


Figure 72 Median manual correctness (MC) value over 30 runs

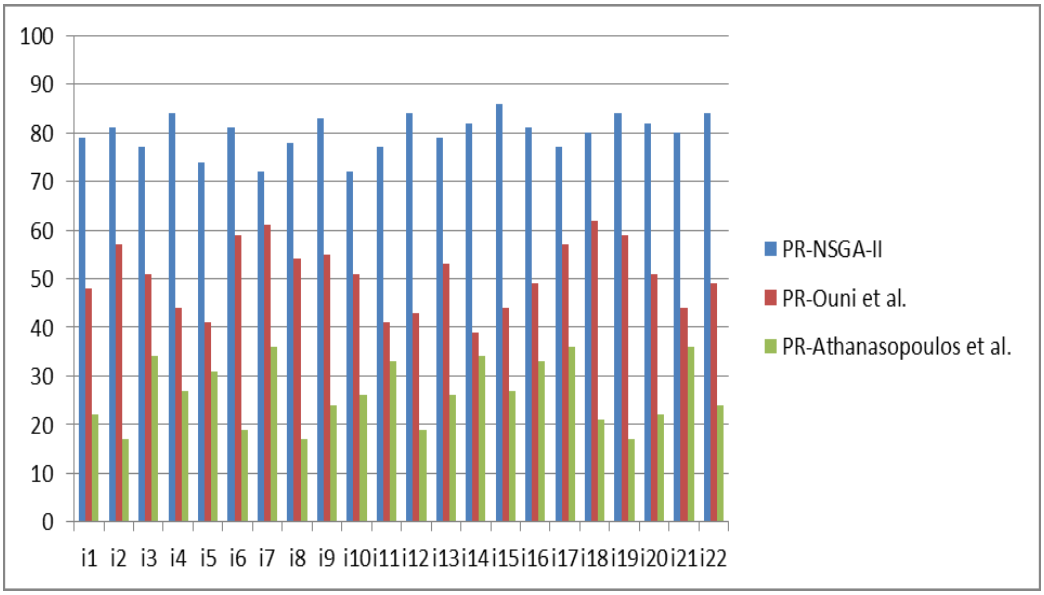


Figure 73 Median precision (PR) value over 30 runs

We evaluated also the ability of our approach to fix several types of design antipatterns and to improve the service interface design quality as described in Figure 75 that depicts the percentage of fixed antipatterns (*NF*). It is higher than 83% on all the 22 Web services, which is an acceptable score since developers may reject or modify some design changes that fix some antipatterns

because they do not consider some of them as very important (their goal is not to fix all design antipatterns in the Web service interface) or because they wanted to focus on improving the cohesion and minimize coupling. Some Web service interfaces, such as *AmazonFWSInboundPortType*, have a higher percentage of fixed code smells with an average of more than 90%. This can be explained by the fact that these Web services include a higher number of design antipatterns than others. We have also considered three other evaluation metrics *NMO* (percentage of modified portTypes), *NRE* (percentage of rejected portTypes) and *NAC* (percentage of accepted portTypes) to evaluate the efficiency of our interactive approach. We collected this data using a feature that we implemented in our tool to record all the actions performed by the developers during the remodularization sessions. Figure 76 shows that, on average, more than 81% of the recommended portTypes were accepted by the developers. In addition, an average of 9% of the recommended refactorings were modified by the developers, while 11% of the suggested refactorings were rejected by the developers. Thus, our recommendation tool successfully suggested a good set of design changes to apply.

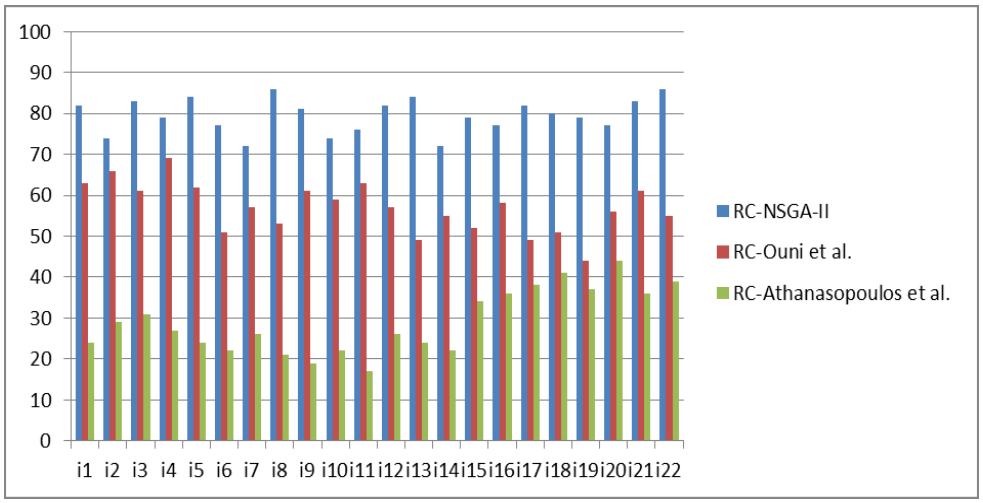


Figure 74 Median recall (RE) value over 30 runs

To summarize and answer RQ1, the experimentation results confirm that our interactive approach helps the participants to restructure their Web service interface design efficiently by finding the relevant portTypes and improve the quality of all the 22 Web services.

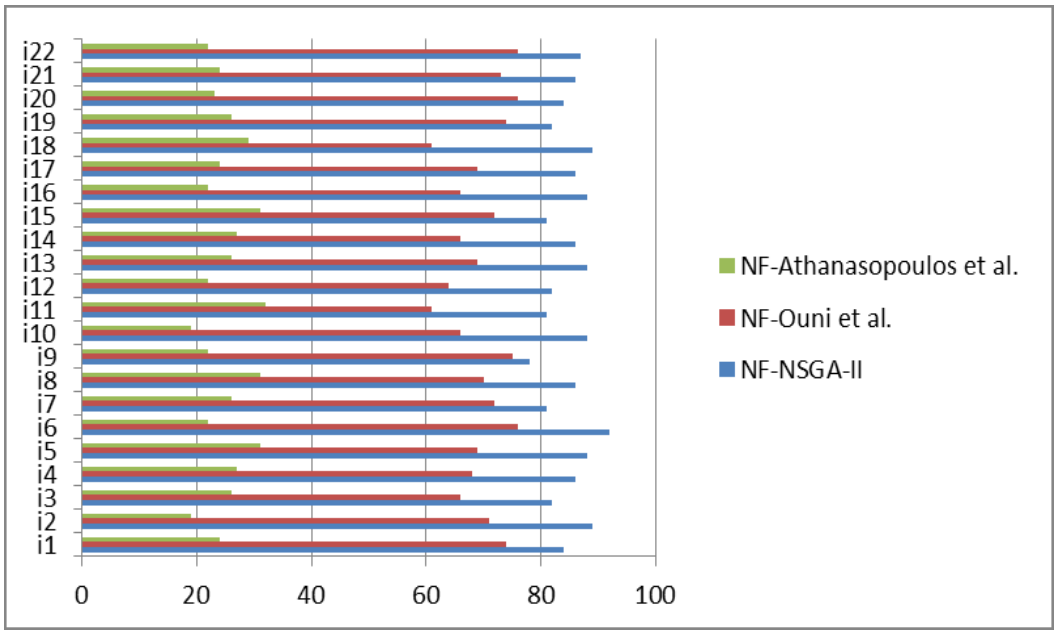


Figure 75 Median number of fixed Web service antipatterns (*NF*) value over 30 runs

Results for RQ2. Figure 72, Figure 73, Figure 74, and Figure 75 confirm the average superior performance of our interactive approach compared to the two existing fully automated Web service modularization techniques. Figure 72 shows that our approach provides significantly higher manual correctness results (*MC*) than all other approaches having *MC* scores respectively between 48% and 61%, on average as MC scores on the different Web services. The same observation is valid for the precision and recall as described in Figure 73 and Figure 74. The outperformance of our technique in terms of percentage of fixed antipatterns, as described in Figure 75, can be explained by the fact that the main goal of existing studies is not to mainly fix these antipatterns (not considered in the fitness function by the work of Ouni et al.).

Overall the superior performance of our interactive approach can be explained by several factors. First, existing studies use only structural indications (quality metrics) to evaluate the

remodularization solutions and thus a high number of changes may lead to a semantically incoherent Web services design. Our approach reduces the number of semantic incoherencies when suggesting refactorings and during the interaction with the developers. Second, the ranking component of our approach improved the quality of the suggested refactoring solutions by using an interactive approach as compared to a regular NSGA-II where the developers need to select one solution from the Pareto front that cannot be updated dynamically. Third, existing work are mainly limited to the cohesion metric which may not be sufficient to guide the modularization of Web services.

In conclusion, our interactive approach provides better results, on average, than all existing fully-automated Web services modularization techniques (answer to RQ2).

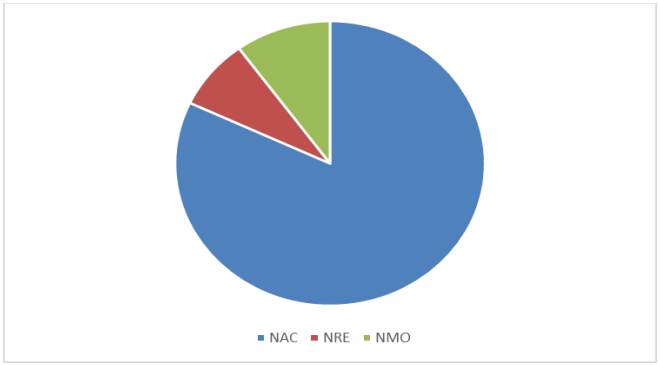


Figure 76 Median percentage of accepted (NAC), modified(NMO) and rejected(NRE) portTypes over 30 runs

Results for RQ3. To further analyze the obtained results, we have have also asked the participants to take a post-study questionnaire after completing the different validation and tasks using our interactive approach and the two techniques considered in our experiments. The post-study questionnaires collected the opinions of the participants about their experience in using our approach compared to fully-automated tools. The post-study questionnaire asked participants to

rate their agreement on a Likert scale from 1 (complete disagreement) to 5 (complete agreement) with the following statements:

The interactive dynamic interface modularization recommendations are a desirable feature to improve the quality of Web services interface.

The interactive manner of recommending modularization solutions by our approach is a useful and flexible way to consider the user perspective compared to fully-automated tools.

The agreement of the participants was 4.9 and 4.6 for the first and second statements respectively. This confirms the usefulness of our approach for the users of our experiments. The remaining questions of the post-study questionnaire were about the benefits and the limitations (possible improvements) of our interactive approach.

We summarize in the following the feedback of the users. Most of the participants mention that our interactive approach is much faster and easy to use compared to the manual restructuring of the interface since they spent a long time with manual changes to create port types and move operations. Thus, the developers liked the functionality of our tool that helps them to modify a port type based on the recommendations.

Another important feature that the participants mention is that our interactive approach allows them to take the advantages of using multi-objective optimization without the need to learn anything about optimization and exploring explicitly the Pareto front to select one “ideal” solution. The implicit exploration of the Pareto front in an interactive fashion represents an important advantage of our tool along with the dynamic update of the recommended design. The participants also suggested some possible improvements to our interactive approach. Some participants believe that it will be very helpful to extend the tool by adding a new feature to decompose multiple

services into interfaces based on the dependency between them. Another possibly suggested improvement is to consider the users invocation data to restructure the interface.

3) Threats to Validity

Conclusion validity is concerned with the statistical relationship between the treatment and the outcome. The parameter tuning of the different computational search algorithms used in our experiments creates another internal threat that we need to evaluate in our future work. The parameters' values used in our experiments are found by trial-and-error. However, it would be an interesting perspective to design an adaptive parameter tuning strategy for our approach so that parameters are updated during the execution to provide the best possible performance. In addition, our multi-objective formulation treats the different types of quality metrics such as coupling and cohesion with the same weight in terms of complexity when calculating one of the fitness functions. However, some quality metrics can be more important than others when evaluating a Web service design but we considered both coupling and cohesion as equally important. The same observation is valid for the different types of considered design antipatterns. Another threat is related to the use of our previous work to detect antipatterns which may include few false positive. However, this threat may not have a high impact on the validity of the results since the different proposed refactorings were manually validated by the participants but some of the rejected recommendations by the developer are related to the detected antipatterns.

Internal validity is concerned with the causal relationship between the treatment and the outcome. We dealt with internal threats to validity by performing 30 independent simulation runs for each problem instance. This makes it highly unlikely that the observed results were caused by anything other than the applied multi-objective approach. The second internal threat is related to the variation of correctness and speed between the different groups when using our approach and

the other tools. In fact, our approach may not be the only reason for the superior performance because the participants have different programming skills and familiarity with Web services and tools. To counteract this, we assigned the developers to different groups per their programming experience to reduce the gap between the different groups and we also adapted a counter-balanced design.

Construct validity is concerned with the relationship between theory and what is observed. The different developers involved in our experiments may have divergent opinions about the recommended modularizations in terms of correctness and readability. We considered in our experiments the majority of votes from the developers. For the selection threat, the participant diversity in terms of experience could affect the results of our study. We addressed the selection threat by giving a lecture and examples of Web services modularization already evaluated with arguments and justification.

External validity refers to the generalizability of our findings. In this study, we performed our experiments on eight different widely used Web services belonging to different domains and having different sizes. However, we cannot assert that our results can be generalized to other Web service, and to other technologies or practitioners. Future replications of this study are necessary to confirm our findings. Further empirical studies are also required to deeply evaluate the performance of the interactive NSGA-II using the same problem formulation. The first threat is the limited number of participants and evaluated Web services, which externally threatens the generalizability of our results. In addition, our study was limited to the use of specific refactoring types and types of design antipatterns. Future replications of this study are necessary to confirm our findings. Another external threat is the current applicability of the proposed tool only on

WSDL interfaces. However, the proposed approach is generic and can be adapted as part of our future work to RESTfull since REST is nowadays widely used to implement services.

6.7.4 **Conclusion**

We proposed, in this work, an interactive recommendation tool for Web services interface design modularization that dynamically adapts and suggests design changes to developers based on their feedback and three objective functions. Our interactive approach allows users to benefit from search-based tools without explicitly involving any knowledge about optimization and multi-objective optimization algorithms. In fact, the exploration of the non-dominated refactoring solutions is implicitly performed based on the interaction with the users. The feedback received from the users is used to reduce the search space and converge to better design modularization solutions.

Chapter 7 **Conclusion and Future work**

7.1 **Conclusion**

To summarize the contributions related to the detection of Web service design defects (chapter 3), we introduced multi-objective and bi-level approaches for this problem. We used interface, code-level metrics, and also introduced QoS metrics for the first time to this problem. We designed fitness functions to guide the search to cover most of the antipatterns example and the minimum of good design practices example. We validated all contributions on over 400 real-world web-services with a median precision and recall over 90% for all three contributions.

In the work around the detection of changes among service releases (chapter 4), we proposed an approach to detect changes during the evolution of Web services. Our genetic algorithm approach generates a list of detected changes in terms of composite changes, and not atomic ones. We evaluated our approach on a set of 6 popular Web services including more than 110 releases. We reported the results on the efficiency and effectiveness of our approach to detect changes of the evolution of Web services interfaces in terms of precision and recall.

For the prediction of service evolution (chapter 5), we propose to use machine learning, based on Artificial Neuronal Networks, to forecast the evolution of Web services interface design. To validate the proposed approach, we collected training data from quality metrics of previous releases from 6 Web services. The validation of our prediction approach shows that the predicted metric's value, such as the number of operations, on the different releases of the 6 Web services were similar to the expected ones with a very low deviation rate. Furthermore, most of the quality

issues of the studied Web service interfaces were accurately predicted, for the next releases, with an average precision and recall higher than 82%.

Finally, we proposed many techniques to automatically recommend better service designs (chapter 6). The proposed multi-objective and many objective search-based approaches generate new service designs based on many quality fitness functions (e.g. interface quality, history-based similarity, user preferences and so on). We introduce machine learning techniques to solve the increasing complexity of search algorithm. An interactive method is also introduced to better evolve the solutions based on user feedback. The feedback received from the users is used to reduce the search space and converge to better design modularization solutions. Finally, we evaluated our work on over 20 major services provided by Amazon and Yahoo, the statistical and survey results show high effectiveness and efficiency of our approaches.

To conclude, this dissertation introduced, using search-based and machine learning algorithms, various approaches to find, predict and correct the Web Services design problems. For the methodologies presented in this work, we have applied machine learning techniques to interpret, understand or predicted the service evolution; multi-objective evolutionary algorithms to search the solutions of defined issue; and objective reduction, bi-level and interactive methodologies to reach better convergence to desired solutions. To the best of our knowledge, this thesis is the first to apply these technologies to solve these problems in the Web Service filed and addresses the problem of refactoring Web services. Based on the validation results, we have outperformed state-of-the-art technologies using various evaluation metrics on existing benchmarks.

7.2 **Future Work**

We plan to extend the detection approach to identify business process and work flow antipatterns in SBS. We also intend to automate the process of service compositions/selection while avoiding antipatterns and improve QoS performance. Future work of Web service prediction involves the validation of our prediction technique with additional metrics, Web services and developers to conclude about the general applicability of our methodology. Furthermore, we plan to extend the prediction approach by defining new risk measures based on the predicted metrics value. In addition, we will study of the impact of predicted quality issues on the usability and popularity of Web services over time.

In our future work for services refactoring, we are planning to validate our technique with additional objectives and Web services in order to conclude about the general applicability of our methodology. Furthermore, we are planning to adapt our dimensionality reduction approach to others problems such as Web services composition. Another future research direction related to our work is to integrate the developers in the loop when reducing the number of objectives to either select which one to eliminate or revise the fitness function formulation (aggregating the objectives).

Finally, we will investigate the use intelligent-based approaches to combine search-based algorithms and machine learning to address large-scale services composition challenges. Most exsiting large software sytems use multiple layers of work flows while each layer involve the compostion of many micro services which are in general hard to evaluate, evolve, or merge with new features. We plan to abstract such system into a structured representation and treat each service and software compoent as a dimension within that structure. Then, we will define new metrics and objectives to evaluate the performance of the micro-services component.

Bibliography

[1] M. Perepletchikov, C. Ryan, and K. Frampton, "Cohesion metrics for predicting maintainability of service-oriented software," in *Proceedings - International Conference on Quality Software*, 2007, pp. 328–335.

[2] D. Romano and M. Pinzger, "Analyzing the evolution of web services using fine-grained changes," in *Proceedings - 2012 IEEE 19th International Conference on Web Services, ICWS 2012*, 2012, pp. 392–399.

[3] A. Ouni, M. Kessentini, K. Inoue, and M. O. Cinneide, "Search-Based Web Service Antipatterns Detection," *IEEE Trans. Serv. Comput.*, vol. 10, no. 4, pp. 603–617, Nov. 2015.

[4] A. Rotem-Gal-Oz, *SOA Patterns*. Manning Publications, 2012.

[5] D. Athanasopoulos, A. V. Zarras, G. Miskos, V. Issarny, and P. Vassiliadis, "Cohesion-Driven Decomposition of Service Interfaces without Access to Source Code," *IEEE Trans. Serv. Comput.*, vol. 8, no. 4, pp. 550–5532, 2015.

[6] P. Mikhail, R. Caspar, and T. Zahir, "The Impact of Service Cohesion on the Analyzability of Service-Oriented Software," *IEEE Trans. Serv. Comput.*, vol. 3, no. 2, pp. 89–103, 2010.

[7] D. Romano and M. Pinzger, "A Genetic Algorithm to Find the Adequate Granularity for Service Interfaces," in *Services (SERVICES), 2014 IEEE World Congress on*, 2014, pp. 478–485.

[8] R. Haesen, M. Snoeck, W. Lemahieu, and S. Poelmans, "On the definition of service granularity and its architectural impact," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2008, vol. 5074 LNCS, pp. 375–389.

[9] M. Perepletchikov, C. Ryan, K. Frampton, and H. Schmidt, "Formalising service-oriented design," *J. Softw.*, vol. 3, no. 2, pp. 1–14, 2008.

[10] B. Dudney, J. Krozak, K. Wittkopf, S. Asbury, and D. Osborne, *J2EE Antipatterns*. John Wiley & Sons, 2003.

[11] F. Palma, N. Moha, G. Tremblay, and Y.-G. Guéhéneuc, "Specification and Detection of SOA Antipatterns in Web Services," in *Software Architecture*, Springer, 2014, pp. 58–73.

[12] A. Ouni, R. G. Kula, M. Kessentini, and K. Inoue, "Web service antipatterns detection using genetic programming," in *Proceedings of the 2015 on Genetic and Evolutionary Computation Conference - GECCO '15*, 2015, pp. 1351–1358.

[13] J. Král and M. Žemlička, "Popular SOA antipatterns," in *Computation World: Future Computing, Service Computation, Adaptive, Content, Cognitive, Patterns, ComputationWorld 2009*, 2009, pp. 271–276.

[14] H. Masoud and S. Jalili, "A clustering-based model for class responsibility assignment problem in object-oriented analysis," *J. Syst. Softw.*, vol. 93, pp. 110–131, 2014.

[15] W. Mkaouer *et al.*, "Many-Objective Software Remodularization Using NSGA-III," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 3, pp. 1–45, 2015.

[16] U. Mansoor, M. Kessentini, M. Wimmer, and K. Deb, "Multi-view refactoring of class and

activity diagrams using a multi-objective evolutionary algorithm," *Softw. Qual. J.*, p. 1, 2015.

[17] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide, "Recommendation system for software refactoring using innovization and interactive dynamic optimization," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14*, 2014, pp. 331–336.

[18] S. Kalboussi, S. Bechikh, M. Kessentini, and L. Ben Said, "Preference-based many-objective evolutionary testing generates harder test cases for autonomous agents," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2013, vol. 8084 LNCS, pp. 245–250.

[19] A. Ouni, M. Kessentini, and H. Sahraoui, "Search-based refactoring using recorded code changes," in *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, 2013, pp. 221–230.

[20] S. Bechikh, M. Kessentini, L. Ben Said, and K. Ghédira, "Preference Incorporation in Evolutionary Multiobjective Optimization: A Survey of the State-of-the-Art," in *Advances in Computers*, vol. 98, 2015, pp. 141–207.

[21] M. Boussaa, W. Kessentini, M. Kessentini, S. Bechikh, and S. Ben Chikha, "Competitive coevolutionary code-smells detection," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2013, vol. 8084 LNCS, pp. 50–65.

[22] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni, "Design defects detection and correction by example," in *IEEE International Conference on Program Comprehension*, 2011, pp. 81–90.

[23] M. Kessentini, A. Bouchoucha, H. Sahraoui, and M. Boukadoum, "Example-based sequence diagrams to colored petri nets transformation using heuristic Search," *Model. Found. Appl.*, pp. 156–172, 2010.

[24] M. Kessentini, P. Langer, and M. Wimmer, "Searching models, modeling search: On the synergies of SBSE and MDE," in *Proceedings of the 1st International Workshop on Combining Modelling and Search-Based Software Engineering*, 2013, pp. 51–54.

[25] M. Kessentini, M. Wimmer, H. Sahraoui, and M. Boukadoum, "Generating transformation rules from examples for behavioral models," in *Proceedings of the Second International Workshop on Behaviour Modelling: Foundation and Applications*, 2010, p. 2.

[26] N. Moha *et al.*, "Specification and Detection of SOA Antipatterns," Springer, Berlin, Heidelberg, 2012, pp. 1–16.

[27] M. Fowler and K. Beck, *Refactoring : improving the design of existing code*. Addison-Wesley, 1999.

[28] H. Wang, M. Kessentini, and A. Ouni, "Interactive Refactoring of Web Service Interfaces Using Computational Search," *IEEE Trans. Serv. Comput. To Appear*.

[29] H. Wang, A. Ouni, M. Kessentini, S. Bouktif, and I. Katsuro, "A Hybrid Approach for Improving the Design Quality of Web Service Interfaces," *ACM Trans. Internet Technol. To Appear*.

[30] H. Wang, M. Kessentini, and K. Gaaloul, "A Search-based approach to History-based Web Service Interface Remodularization," *ACM Trans. Web, To Appear*.

[31] H. Wang, M. Kessentini, T. Hassouna, and A. Ouni, "On the Value of Quality of Service Attributes for Detecting Bad Design Practices," in *2017 IEEE International Conference on Web Services (ICWS)*, 2017, pp. 341–348.

[32] M. Kessentini and H. Wang, “Detecting Refactorings among Multiple Web Service Releases: A Heuristic-Based Approach,” in *2017 IEEE International Conference on Web Services (ICWS)*, 2017, pp. 365–372.
[33] M. Kessentini, H. Wang, J. T. Dea, and A. Ouni, “Improving Web Services Design Quality Using Heuristic Search and Machine Learning,” in *2017 IEEE International Conference on Web Services (ICWS)*, 2017, pp. 540–547.
[34] H. Wang and M. Kessentini, “Improving Web Services Design Quality Using Dimensionality Reduction Techniques,” in *International Conference on Service-Oriented Computing*, 2017, pp. 499–507.
[35] H. Wang, M. Kessentini, and A. Ouni, “Bi-level Identification of Web Service Defects,” in *International Conference on Service-Oriented Computing*, 2016, pp. 352–368.
[36] H. Wang, M. Kessentini, and A. Ouni, “Prediction of web services evolution,” in *International Conference on Service-Oriented Computing*, 2016, vol. 9936 LNCS, pp. 282–297.
[37] H. Wang, A. Ouni, M. Kessentini, B. Maxim, and W. I. Grosky, “Identification of Web Service Refactoring Opportunities as a Multi-objective Problem,” in *2016 IEEE International Conference on Web Services (ICWS)*, 2016, pp. 586–593.
[38] H. Wang, M. Kessentini, W. Grosky, and H. Meddeb, “On the use of time series and search based software engineering for refactoring recommendation,” in *Proceedings of the 7th International Conference on Management of computational and collective intElligence in Digital EcoSystems - MEDES ’15*, 2015, no. October, pp. 35–42.
[39] S. W. Choi, J. S. Her, and S. D. Kim, “QoS metrics for evaluating services from the perspective of service providers,” in *Proceedings - ICEBE 2007: IEEE International Conference on e-Business Engineering - Workshops: SOAIC 2007; SOSE 2007; SOKM 2007*, 2007, pp. 622–625.
[40] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, “A fast and elitist multiobjective genetic algorithm: NSGA-II,” *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, 2002.
[41] M. Crasso, J. M. Rodriguez, A. Zunino, and M. Campo, “Revising WSDL Documents: Why and How,” *Internet Comput. IEEE*, no. 5, pp. 48–56.
[42] C. Mateos, M. Crasso, A. Zunino, and J. Coscia, “Avoiding WSDL bad practices in code-first web services,” *SADIO Electron. J. Informatics*, 2012.
[43] S. Haykin, “Neural networks-A comprehensive foundation,” *New York: IEEE Press. Herrmann, M., Bauer, H.-U., & Der, R*, vol. psychology. p. pp107-116, 1994.
[44] M. Mitchell, “An introduction to genetic algorithms,” *Comput. Math. with Appl.*, vol. 32, no. 6, p. 133, 1996.
[45] T. Jayalakshmi and A. Santhakumaran, “Statistical Normalization and Backpropagation for Classification,” *Int. J. Comput. Theory Eng.*, vol. 3, no. 1, pp. 89–93, 2011.
[46] K. Deb and H. Jain, “An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, Part I: Solving problems with box constraints,” *IEEE Trans. Evol. Comput.*, vol. 18, no. 4, pp. 577–601, 2014.
[47] K. Deb and D. K. Saxena, “Searching for Pareto-optimal solutions through dimensionality reduction for certain large- dimensional multi-objective optimization problems Searching For Pareto-Optimal Solutions Through Dimensionality Reduction for Certain Large-Dimensional Multi-Object,” *2006 IEEE Congress on Evolutionary Computation*, no. March. 2006.
[48] M. V. Mäntylä and C. Lassenius, “Subjective evaluation of software evolvability using code

smells: An empirical study," in *Empirical Software Engineering*, 2006, vol. 11, no. 3, pp. 395–431.
[49] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, 1994.
[50] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, and V. G. Da Fonseca, "Performance assessment of multiobjective optimizers: An analysis and review," *IEEE Transactions on Evolutionary Computation*, vol. 7, no. 2. pp. 117–132, 2003.
[51] J. M. Rodriguez, M. Crasso, C. Mateos, and A. Zunino, "Best practices for describing, consuming, and discovering web services: A comprehensive toolset," *Softw. - Pract. Exp.*, vol. 43, no. 6, pp. 613–639, 2013.
[52] J. M. Rodriguez, M. Crasso, A. Zunino, and M. Campo, "Automatically detecting opportunities for web service descriptions improvement," in *IFIP Advances in Information and Communication Technology*, 2010, vol. 341 AICT, pp. 139–150.
[53] M. Nayrolles, F. Palma, N. Moha, and Y.-G. Guéhéneuc, "Soda: A Tool Support for the Detection of SOA Antipatterns," in *Service-Oriented Computing - ICSOC 2012 Workshops*, vol. 7759, A. Ghose, H. Zhu, Q. Yu, A. Delis, Q. Sheng, O. Perrin, J. Wang, and Y. Wang, Eds. Springer Berlin Heidelberg, 2013, pp. 451–455.
[54] N. Moha and Y. Guéhéneuc, "DECOR: A method for the specification and detection of code and design smells," *... , IEEE Trans.*, vol. 36, no. 1, pp. 20–36, 2010.
[55] M. A. Torkamani and H. Bagheri, "A Systematic Method for Identification of Anti-patterns in Service Oriented System Development," *Int. J. Electr. Comput. Eng.*, vol. 4, no. 1, p. 16, 2014.
[56] C. Mateos, M. Crasso, A. Zunino, and J. L. O. Coscia, "Detecting WSDL bad practices in code–first Web Services," *Int. J. Web Grid Serv.*, vol. 7, no. 4, pp. 357–387, 2011.
[57] M. Fokaefs, R. Mikhaiel, N. Tsantalis, E. Stroulia, and A. Lau, "An Empirical Study on Web Service Evolution," in *2011 IEEE International Conference on Web Services*, 2011, pp. 49–56.
[58] L. Aversano, M. Bruno, M. Di Penta, A. Falanga, and R. Scognamiglio, "Visualizing the Evolution ofWeb Services using Formal Concept Analysis," in *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*, 2005, pp. 57–60.
[59] Z. Xing and E. Stroulia, "UMLDiff: An Algorithm for Object-Oriented Design Differencing," *Proc. 20th IEEE/ACM Int. Conf. Autom. Softw. Eng. - ASE '05*, p. 54, 2005.
[60] A. V. Zarras, P. Vassiliadis, and I. Dinos, "Keep calm and wait for the spike! insights on the evolution of amazon services," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2016, vol. 9694, pp. 444–458.
[61] M. Klusch, P. Kapahnke, and I. Zinnikus, "SAWSDL-MX2: A machine-learning approach for integrating semantic web service matchmaking variants," in *2009 IEEE International Conference on Web Services, ICWS 2009*, 2009, pp. 335–342.
[62] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based Software Engineering: Trends, Techniques and Applications," *ACM Comput. Surv.*, vol. 45, no. 1, p. 11:1--11:61, 2012.
[63] F. Qayum and R. Heckel, "Local search-based refactoring as graph transformation," in *Proceedings - 1st International Symposium on Search Based Software Engineering, SSBSE 2009*, 2009, pp. 43–46.
[64] H. Kilic, E. Koc, and I. Cereci, "Search-Based Parallel Refactoring Using Population-Based Direct Approaches," *Search Based Softw. Eng.*, vol. 6956, pp. 271–272, 2011.

[65] M. Harman and L. Tratt, “Pareto optimal search based refactoring at the design level,” in *Proceedings of the 9th annual conference on Genetic and evolutionary computation - GECCO ’07*, 2007, p. 1106.
[66] M. Ó Cinnéide, L. Tratt, M. Harman, S. Counsell, and I. Hemati Moghadam, “Experimental assessment of software metrics using automated refactoring,” in *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement - ESEM ’12*, 2012, p. 49.
[67] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi, “The Use of Development History in Software Refactoring Using a Multi-Objective Evolutionary Algorithm,” *Proc. 15th Annu. Conf. Genet. Evol. Comput.*, no. July, pp. 1461–1468, 2013.
[68] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, “Identification and application of Extract Class refactorings in object-oriented systems,” *J. Syst. Softw.*, vol. 85, no. 10, pp. 2241–2260, 2012.
[69] A. Ouni, Z. Salem, K. Inoue, and M. Soui, “SIM: An automated approach to improve web service interface modularization,” in *Proceedings - 2016 IEEE International Conference on Web Services, ICWS 2016*, 2016, pp. 91–98.
[70] T. Erl, *SOA Design Patterns*. Prentice Hall PTR, 2009.
[71] B. K. Giri, J. Hakanen, K. Miettinen, and N. Chakraborti, “Genetic programming through bi-objective genetic algorithms with a study of a simulated moving bed process involving multiple objectives,” *Appl. Soft Comput.*, vol. 13, no. 5, pp. 2613–2623, 2013.
[72] “Experimental data,” *WSantipatterns*. [Online]. Available: https://github.com/ouniali/. [Accessed: 14-Aug-2015].
[73] W. B. (William B. Frakes and R. (Ricardo) Baeza-Yates, “Information retrieval : data structures & algorithms,” *Inf. Retr. Boston.*, p. 504, 1992.
[74] J. Krai and M. Zemlicka, “The Most Important Service-Oriented Antipatterns,” in *Software Engineering Advances, 2007. ICSEA 2007. International Conference on*, 2007, p. 29.
[75] J. F. Bard, *Practical bilevel optimization: algorithms and applications*, vol. 30. Springer Science & Business Media, 2013.
[76] B. Colson, P. Marcotte, and G. Savard, “An overview of bilevel optimization,” *Ann. Oper. Res.*, vol. 153, no. 1, pp. 235–256, 2007.
[77] D. D. Clark, S. Shenker, and L. Zhang, “Supporting real-time applications in an Integrated Services Packet Network: architecture and mechanism,” *Sigcomm’92*, pp. 14–26, 1992.
[78] S. Frølund and J. Koistinen, “Quality-of-service specification in distributed object systems,” *Distrib. Syst. Eng.*, vol. 5, no. 4, p. 179, 1998.
[79] M. A. Hiltunen, R. D. Schlichting, C. A. Ugarte, and G. T. Wong, “Survivability through customization and adaptability: The Cactus approach,” in *Proceedings - DARPA Information Survivability Conference and Exposition, DISCEX 2000*, 2000, vol. 1, pp. 294–307.
[80] R. L. Cruz, “Quality of Service Guarantees in Virtual Circuit Switched Networks,” *IEEE J. Sel. Areas Commun.*, vol. 13, no. 6, pp. 1048–1056, 1995.
[81] L. Georgiadis, R. Guérin, V. Peris, and K. N. Sivarajan, “Efficient network QoS provisioning based on per node traffic shaping,” *IEEE/ACM Trans. Netw.*, vol. 4, no. 4, pp. 482–501, 1996.
[82] J. Cardoso, A. Sheth, J. Miller, J. Arnold, and K. Kochut, “Quality of service for workflows and web service processes,” *Web Semant.*, vol. 1, no. 3, pp. 281–308, 2004.
[83] D. A. Menascé, “QoS issues in web services,” *IEEE Internet Comput.*, vol. 6, no. 6, pp. 72–

75, 2002.
[84] A. Benveniste, “Composing Web Services in an Open World: QoS Issues,” in *Quantitative Evaluation of Systems, 2008. QEST ’08. Fifth International Conference on*, 2008, p. 121.
[85] M. C. Jaeger, G. Rojec-Goldmann, and G. M??hl, “QoS aggregation for Web service composition using workflow patterns,” in *Proceedings - IEEE International Enterprise Distributed Object Computing Workshop, EDOC*, 2004, pp. 149–159.
[86] S. Ran, “A model for web services discovery with QoS,” *ACM SIGecom Exch.*, vol. 4, no. 1, pp. 1–10, 2003.
[87] G. V. Bochmann, B. Kerhervé, H. Lutfiyya, M. V. M. Salem, and H. Ye, “Introducing qos to electronic commerce applications,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2001, vol. 2040, pp. 138–147.
[88] F. Mardukhi, N. NematBakhsh, K. Zamanifar, and A. Barati, “QoS decomposition for service composition using genetic algorithm,” *Appl. Soft Comput.*, vol. 13, no. 0, p. , 2013.
[89] K. Christos, D. C. Vassilakis, E. Rouvas, and D. P. Georgiadis, “QoS-driven adaptation of BPEL scenario execution,” in *2009 IEEE International Conference on Web Services, ICWS 2009*, 2009, pp. 271–278.
[90] G. Fan, H. Yu, L. Chen, and D. Liu, “Petri net based techniques for constructing reliable service composition,” in *Journal of Systems and Software*, 2013, vol. 86, no. 4, pp. 1089–1106.
[91] R. Sindhgatta, B. Sengupta, and K. Ponnalagu, “Measuring the Quality of Service Oriented Design,” *ICSOC-ServiceWave 2009*, vol. 5900, pp. 485–499, 2009.
[92] J. L. O. Coscia, M. Crasso, C. Mateos, and A. Zunino, “Estimating Web Service interface quality through conventional object-oriented metrics,” *CLEI Electron. J.*, vol. 16, no. 1, 2013.
[93] R. Marinescu, “Detection strategies: Metrics-based rules for detecting design flaws,” in *IEEE International Conference on Software Maintenance, ICSM*, 2004, pp. 350–359.
[94] J. Král and M. Žemlička, “Crucial Service-Oriented Antipatterns,” *Int. J. Adv. Softw.*, vol. 2, no. 1, pp. 160–171, 2009.
[95] F. Palma, “Specification and Detection of SOA Antipatterns,” in *2014 IEEE International Conference on Software Maintenance and Evolution*, Springer, 2014, pp. 670–670.
[96] H. Zheng, W. Zhao, J. Yang, and A. Bouguettaya, “QoS Analysis for Web Service Composition,” *2009 IEEE Int. Conf. Serv. Comput.*, vol. 2, no. 1, pp. 235–242, 2009.
[97] J. Durillo and J. García-Nieto, “Multi-objective particle swarm optimizers: An experimental comparison,” *5th Int. Conf. EMO 2009*, pp. 495–509, 2009.
[98] M. Harman and B. F. Jones, “Search-based software engineering,” *Inf. Softw. Technol.*, vol. 43, no. 14, pp. 833–839, 2001.
[99] P. Moscato and others, “On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms,” *Caltech Concurr. Comput. program, C3P Rep.*, vol. 826, p. 1989, 1989.
[100] R. Baeza-Yates and B. Ribeiro-Neto, “Modern information retrieval,” *New York*, vol. 9, p. 513, 1999.
[101] L. Yujian and L. Bo, “A normalized Levenshtein distance metric,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 29, no. 6, pp. 1091–1095, 2007.
[102] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, “Detecting bad smells in source code using change history information,” in *2013 28th*

IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings, 2013, pp. 268–278.
[103] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Trans. Softw. Eng.*, vol. 30, no. 2, pp. 126–139, 2004.
[104] G. E. P. Box, G. M. Jenkins, and G. C. Reinsel, *Time Series Analysis: Forecasting & Control*. 1994.
[105] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. 'O Cinn'eide, "High Dimensional Search-based Software Engineering: Finding Tradeoffs Among 15 Objectives for Automating Software Refactoring Using NSGA-III," *Proc. 2014 Conf. Genet. Evol. Comput.*, no. June, pp. 1263–1270, 2014.
[106] B. Schelter, M. Winterhalder, and J. Timmer, "Handbook of Time Series Analysis," *Time*, vol. 1, p. 496, 2006.
[107] U. Raja, D. P. Hale, and J. E. Hale, "Modeling software evolution defects: a time series approach," *J. Softw. Maint. Evol. Res. Pract.*, vol. 21, no. 1, pp. 49–71, 2009.
[108] C. Couto, P. Pires, M. T. Valente, R. S. Bigonha, and N. Anquetil, "Predicting software defects with causality tests," *J. Syst. Softw.*, vol. 93, pp. 24–41, 2014.
[109] G. Canfora, M. Ceccarelli, L. Cerulo, and M. Di Penta, "Using multivariate time series and association rules to detect logical change coupling: An empirical study," in *IEEE International Conference on Software Maintenance, ICSM*, 2010.
[110] G. Antoniol, G. Casazza, M. Di Penta, and E. Merlo, "Modeling clones evolution through time series," in *IEEE International Conference on Software Maintenance, ICSM*, 2001, pp. 273–280.
[111] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Trans. Softw. Eng.*, vol. 28, no. 1, pp. 4–17, 2002.
[112] W. H. Kruskal and W. A. Wallis, "Use of Ranks in One-Criterion Variance Analysis," *J. Am. Stat. Assoc.*, vol. 47, no. 260, pp. 583–621, 1952.
[113] M. . Gardner and S. . Dorling, "Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences," *Atmos. Environ.*, vol. 32, no. 14–15, pp. 2627–2636, 1998.
[114] W. G. Cobourn, L. Dolcine, M. French, and M. C. Hubbard, "A comparison of nonlinear regression and neural network models for ground-level ozone forecasting.," *J. Air Waste Manag. Assoc.*, vol. 50, no. 11, pp. 1999–2009, 2000.
[115] E. Agirre, G. Ibarra, A. Anta, and L. J. R. Barron, "Evaluation of a multilayer perceptron based model to forecast O 3 and NO 2 hourly levels," *Environ. Model. Softw.*, vol. 21, no. 2, pp. 430–446, 2006.
[116] H. A. Simon, "Why Should Machines Learn?," in *Machine Learning: An Artificial Intelligence Approach*, 1983, pp. 25–37.
[117] L. Al Shalabi, Z. Shaaban, and B. Kasasbeh, "Data Mining: A Preprocessing Engine," *J. Comput. Sci.*, vol. 2, no. 9, pp. 735–739, 2006.
[118] M. Endrei *et al.*, "Patterns: Service-Oriented Architecture and Web Services," *Contract*, vol. 1, pp. 17–42, 2004.
[119] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*. Pearson Education India, 2005.
[120] T. H. Noor, Q. Z. Sheng, S. Zeadally, and J. Yu, "Trust management of services in cloud environments: Obstacles and solutions," *ACM Comput. Surv.*, vol. 46, no. 1, pp. 1–30, 2013.
[121] M. P. Papazoglou, "Service -oriented computing: Concepts, characteristics and directions,"

in *Proceedings - 4th International Conference on Web Information Systems Engineering, WISE 2003*, 2003, pp. 3–12.
[122] D. Budgen, "Design patterns: Magic or myth?," *IEEE Softw.*, vol. 30, no. 2, pp. 87–90, 2013.
[123] D. Webster, P. Townend, and J. Xu, "Interface refactoring in performance-constrained web services," in *Proceedings - 2012 15th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC 2012*, 2012, pp. 111–118.
[124] K. Praditwong, M. Harman, and X. Yao, "Software module clustering as a multi-objective search problem," *IEEE Trans. Softw. Eng.*, vol. 37, no. 2, pp. 264–282, 2011.
[125] D. Athanasopoulos and A. V. Zarras, "Fine-grained metrics of cohesion lack for service interfaces," in *Proceedings - 2011 IEEE 9th International Conference on Web Services, ICWS 2011*, 2011, pp. 588–595.
[126] E. R. Hruschka, R. J. G. B. Campello, A. A. Freitas, and A. C. P. L. F. de Carvalho, "A survey of evolutionary algorithms for clustering," *IEEE Trans. Syst. Man Cybern. Part C Appl. Rev.*, vol. 39, no. 2, pp. 133–155, 2009.
[127] M. Perepletchikov, C. Ryan, K. Frampton, and Z. Tari, "Coupling Metrics for Predicting Maintainability in Service-Oriented Designs," in *2007 Australian Software Engineering Conference (ASWEC'07)*, 2007, pp. 329–340.
[128] K. J. Stewart, D. P. Darcy, and S. L. Daniel, "Opportunities and challenges applying functional data analysis to the study of open source software evolution," *Stat. Sci.*, vol. 21, no. 2, pp. 167–178, 2006.
[129] K. Deb and H. Jain, "Handling many-objective problems using an improved NSGA-II procedure," in *2012 IEEE Congress on Evolutionary Computation, CEC 2012*, 2012.
[130] V. Khare, X. Yao, and K. Deb, "Performance Scaling of Multi-objective Evolutionary Algorithms," in *Evolutionary Multi-Criterion Optimization, EMO 2003*, 2003, vol. 2632, no. JULY, pp. 376–390.
[131] Z. Wang, K. Tang, and X. Yao, "Multi-objective approaches to optimal testing resource allocation in modular software systems," *IEEE Trans. Reliab.*, vol. 59, no. 3, pp. 563–575, 2010.
[132] M. Harman, "Software engineering meets evolutionary computation," *Computer (Long. Beach. Calif).*, vol. 44, no. 10, pp. 31–39, 2011.
[133] I. Das and J. E. Dennis, "Normal-Boundary Intersection: A New Method for Generating the Pareto Surface in Nonlinear Multicriteria Optimization Problems," *SIAM J. Optim.*, vol. 8, no. 3, pp. 631–657, 1998.
[134] M. Asafuddoula, T. Ray, and R. Sarker, "A decomposition based evolutionary algorithm for many objective optimization with systematic sampling and adaptive epsilon control," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2013, vol. 7811 LNCS, pp. 413–427.
[135] M. Kessentini, R. Mahaouachi, and K. Ghedira, "What you like in design use to correct bad-smells," *Softw. Qual. J.*, vol. 21, no. 4, pp. 551–571, 2013.
[136] U. Mansoor, M. Kessentini, B. R. Maxim, and K. Deb, "Multi-objective code-smells detection using good and bad design examples," *Softw. Qual. J.*, vol. 25, no. 2, pp. 529–552, 2017.
[137] D. K. Saxena, J. A. Duro, A. Tiwari, K. Deb, and Q. Zhang, "Objective reduction in many-objective optimization: Linear and nonlinear algorithms," *IEEE Trans. Evol. Comput.*, vol.

17, no. 1, pp. 77–99, 2013.
[138] J. E. Jackson, *A User's Guide to Principal Components*, vol. 43. 1991.