Identification of Web Service Defects as an Optimization Problem

By

Wenhao Zhang

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
(Software Engineering)
in The University of Michigan
2017

Master's Thesis Committee:

Assistant Professor Marouane Kessentini, Chair
Assistant Professor Kiumi Akingbehin
Associate Professor Zhiwei Xu

DEDICATION

To the people I love.

ACKNOWLEDGEMENTS

It is with a great joy that I reserve these few lines of gratitude and deep appreciation to all those who directly or indirectly contributed to the completion of this work:

I express my greatest gratitude to Dr. Marouane Kessentini, who dedicated all his wonderful time to collaborate, support and lead me to the end of this piece of work. His advices, dedication, availability, relevant comments, corrections and committeemen led to the success of this work.

I also express my greatest thanks to SBSE members especially Mohamed Wiem who supported me with valuable feedback and always kindly encouraged me to succeed this project and Hanzhang Wang who helped me a lot in this field.

I thank all the professors of the CIS master degree who have used their valuable time to transmit the knowledge that help in putting this work together me.

Finally, I wish to express my deep gratitude and thank my family who has consistently expressed its unconditional support and encouragement.

All those who contributed in one way or another, to make this work, can be found here, the crowning of their efforts.

Thank you.

TABLE OF CONTENTS

Dedication ii

Acknowledgements iii

List of Figures v

List of Tables vi

Abstract vii

Chapter 1: Introduction 1

Chapter 2: Background 5

2.1 Web Service Interface Defects 5

2.2 Problem Statement 8

Chapter 3: Bi-Level Identification of Web Services Design Defects 11

3.1 Approach Overview 11

3.2 Solution Approach 14

Chapter 4: Validation 17

4.2 Experimental Setting 18

4.3 Parameters Tuning 20

4.4 Results 20

Chapter 5: Related work 28

Chapter 6: Conclusion and Future work 30

References 31

LIST OF FIGURES

Figure 1: God object Web service (*GOWS*) example **9**

Figure 2: Bi-level Web service defects detection overview **12**

Figure 3: Solution representation at the upper level **13**

Figure 4: The proposed interface design modularization tool **14**

Figure 5: Median precision value over 30 runs on all the 10 Web service categories using the different detection techniques with a 95% confidence level (α < 5%) **23**

Figure 6: Median recall value over 30 runs on all the 10 Web service categories using the different detection techniques with a 95% confidence level (α < 5%) **24**

Figure 7: The impact of the number of Web service defect examples on the quality of the results (Precision on the *Financial* Web services). **25**

Figure 8: The relevance of detected Web service defects evaluated by the subjects **26**

Figure 9: The usefulness of detected Web service defects evaluated by the subjects **27**

LIST OF TABLES

Table 1: List of metrics **6**

Table 2: Used Web services in our experiments **18**

Table 3: Median precision and recall results based on 30 runs **22**

ABSTRACT

Successful Web services must evolve to remain relevant (e.g. requirements update, bugs fix, etc.), but this process of evolution increases complexity and can cause the Web service interface design to decay and lead to significantly reduced usability and popularity of the services. Maintaining a high level of design quality is extremely expensive due to monetary and time pressures that force programmers to neglect improving the quality of their interfaces. A more fundamental reason is that there is little support to automatically identify design defects at the Web service interface level and reduce the high calibration effort to determine manually the threshold value for each quality metric to identify design defects. In this work, we propose to treat the generation of interface design defects detection rules as a bi-level optimization problem.

To this end, the upper level problem generates a set of detection rules, as combination of quality metrics, which maximizes the coverage of a base of defects examples extracted from several Web services and artificial defects generated by the lower level. The lower level maximizes the number of generated artificial defects that cannot be detected by the rules produced by the upper level. The statistical analysis of our experiments over 30 runs on a benchmark of 415 Web services shows that 8 types of Web service 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 and the survey performed by potential users and programmers also shows the relevance of the detected defects.

Key words: Web service interface, design defects, quality of services.

Chapter 1: Introduction

Web services have been emerging in recent years to become one of the most popular techniques for building service-based systems (SBSs) [1]. The implementation of these systems is highly relying on the operations selected from the interface of the employed Web services that are provided by several companies such as Fedex, eBay, Google, FedEx and PayPal [5]. Like any software project, the evolution of Web services may increase the complexity of the Web service interface design. However, maintaining a high level of Web service design quality is extremely expensive due to monetary and time pressures that force programmers to neglect improving the quality of their interfaces that may leads to significantly reduced usability and popularity of the services. Thus, investigating quality of Web services is becoming more and more important. An example of well-known interface design defect is *God object Web service* (GOWS) [8] which implements a multitude of operations related to different business and technical abstractions in a single service leading to low cohesion of its methods and unavailability to end users because it is overloaded.

Unlike the area of object oriented design, there has been recently few studies focusing on the study of bad design practices for web services interface [3,4,5,8]. 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 [3]. These difficulties explain a large portion of the high false-positive rates reported in existing research [5]. Recently, a heuristic-based approach based on genetic programming [8] 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 [6]. 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 thesis can be summarized as follows:

1. This work introduces a novel formulation of the Web services design defects detection as a bi-level problem.
2. It also 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 [5,8] and the survey performed by potential users and programmers also shows the relevance of detected defects.

The remainder of this thesis is as follows: Chapter 2 presents the relevant background, a motivating example for the presented work and an overview of the related work; Chapter 3 describes the search algorithm; an evaluation of the algorithm is explained and its results are discussed in Chapter 4. Finally, concluding remarks and future work are provided in Chapter 5

Chapter 2: Background

We first detail some required background information to understand the problem addressed in this work, then we present a motivating example to illustrate the limitations of existing studies. Finally, we present an overview of existing work.

2.1 Web Service Interface Defects

Web service interface 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 [1,3]. 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 [2]. 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 [1,2,3]. 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 [4,5,8], cover different maintainability factors, due to the availability of defect examples

and to compare the performance of our detection technique to existing studies [5,8]. However, the proposed approach in this thesis 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 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

2.2 Problem Statement

In the following, we introduce some issues and challenges related to the detection of the Web service defects. Overall, there is no general consensus on how to decide if a particular 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 1. Although we can measure the size of an interface, an appropriate threshold value is not trivial to define. An interface considered large in a given 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.

To address the above-mentioned challenges, we propose to consider the Web service defects detection problem as a bi-level optimization problem.

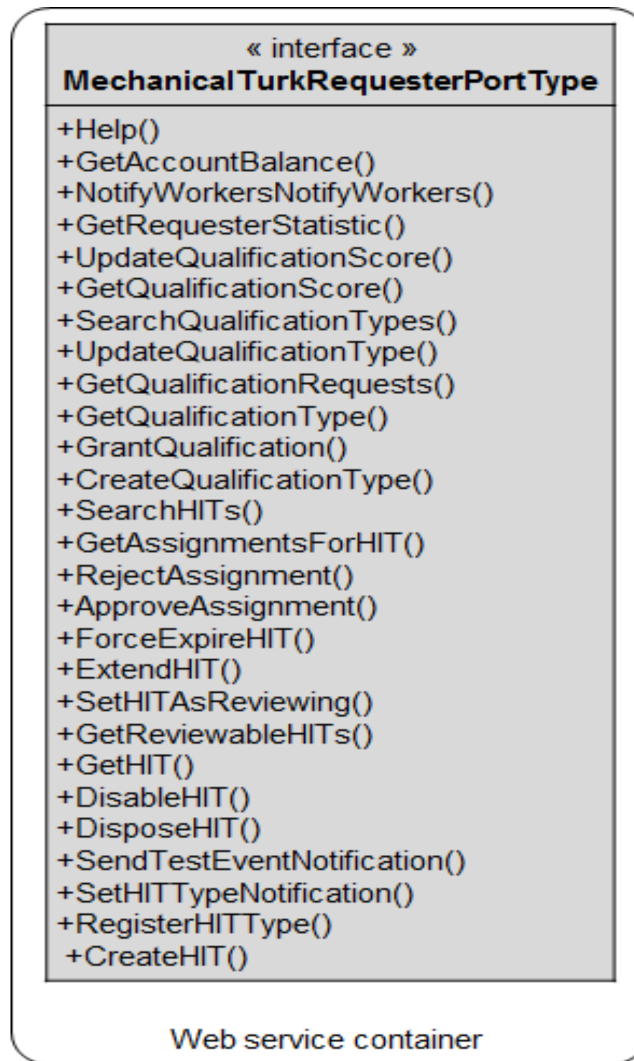

Figure1: God object Web service (*GOWS*) example

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 [6]. 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

Chapter 3: Bi-Level Identification Of Web Services Design Defects

In this chapter, we present an overview of our approach and then we provide the details of our problem formulation and the solution approach.

3.1 Approach Overview

As described in Figure 2, 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.

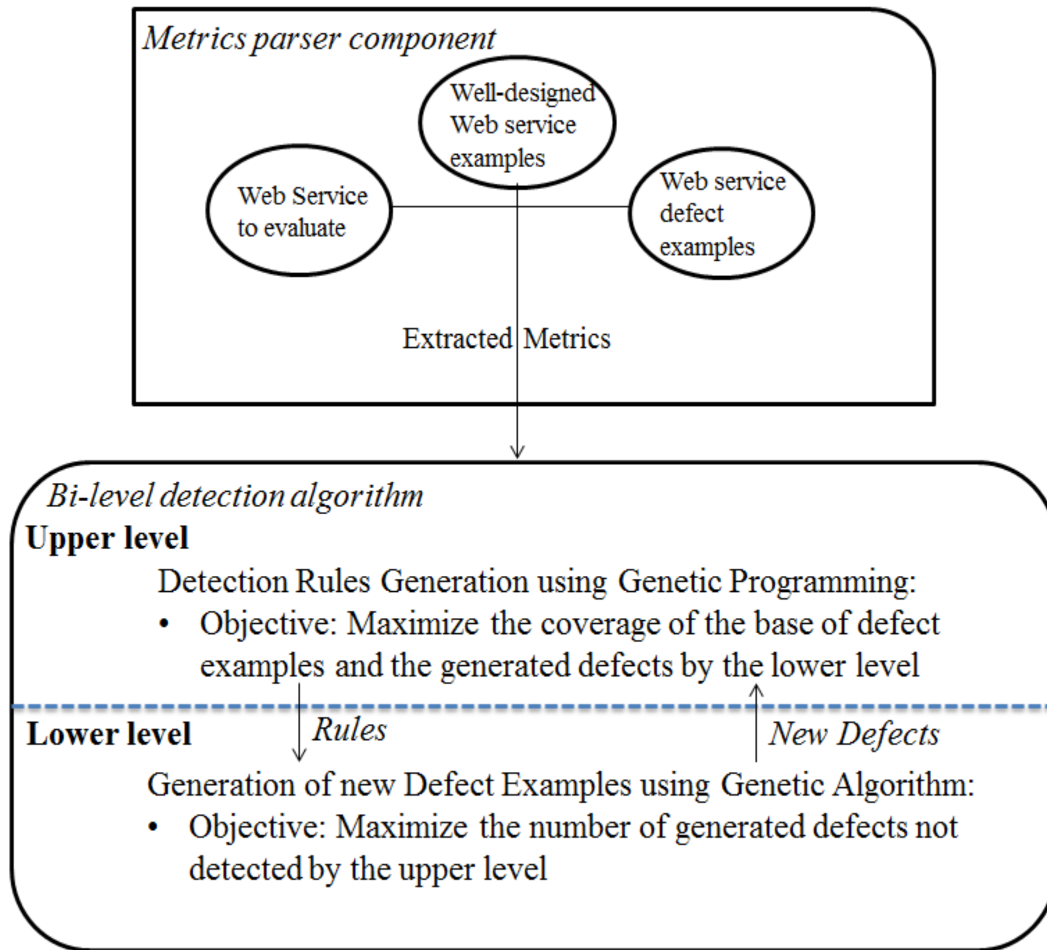

Figure 2: Bi-level Web service defects detection overview

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 3 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*

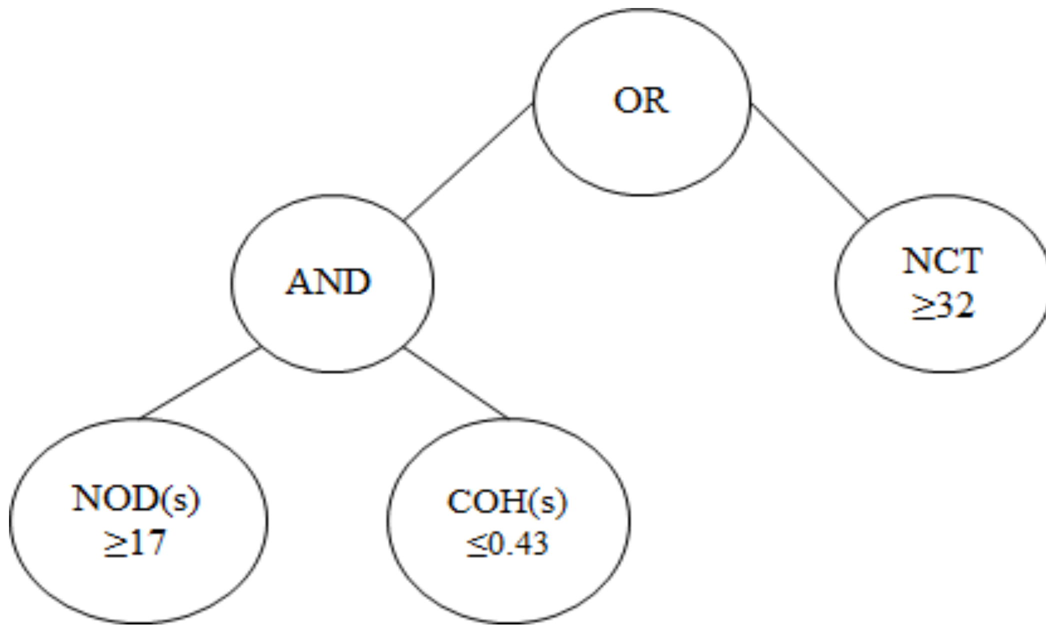

Figure 3: Solution representation at the upper level

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 4, the generated structure of defects are 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 4: 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 Solution Approach

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:

$$Maximize\ f_{upper} = \frac{\frac{Precision(SR, WSDefectExamples) + Recall(SR, WSDefectExamples)}{2} + \frac{\#detectedArtificialWSDefects}{\#artificialWSDefects}}{2}$$

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.

Chapter 4: Validation

To validate the ability of our interactive interface modularization framework to generate a good design quality, we conducted a set of experiments based on real-world web services. 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.

4.1 Research Questions and Evaluation Metrics

In order to evaluate the fesability 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?

4.2 Experimental Setting

To evaluate the performance of our approach, we used an existing benchmark [5,8] that includes a set of Web services from different categories as described in Table 2.

Table 2: Used Web services in our experiments

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 Section 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 to evaluate the accuracy of our approach and to

compare it with existing techniques [5,58]. 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 considere[8] without the use of the lower level.

To answer RQ3, we compared our approach with the SODA-W approach of Palma et al. [5]. 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 2.

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 form 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.

4.3 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 (α = 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.

4.4 Results

The results for the first research question RQ1 are presented in Table 3. 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. Figures 5 and 6 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) [8] where only defect examples are used (without the consideration of the lower-level algorithm), and an existing detection tool (SODA-W) [5] not based on computational search. Figures 5 and 6 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 [8]

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 3: 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

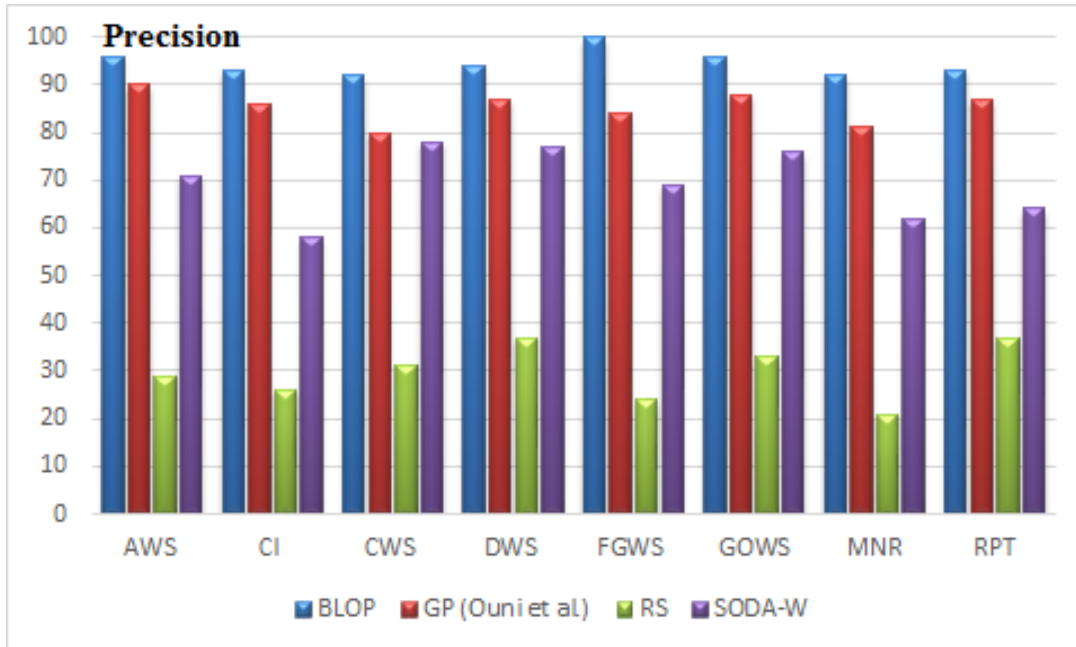

Figure 5: Median precision value over 30 runs on all the 10 Web service categories using the different detection techniques with a 95% confidence level (α < 5%)

While SODA-W shows promising results with an average precision of 71% and recall of 83% (Figures 5 and 6), 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.

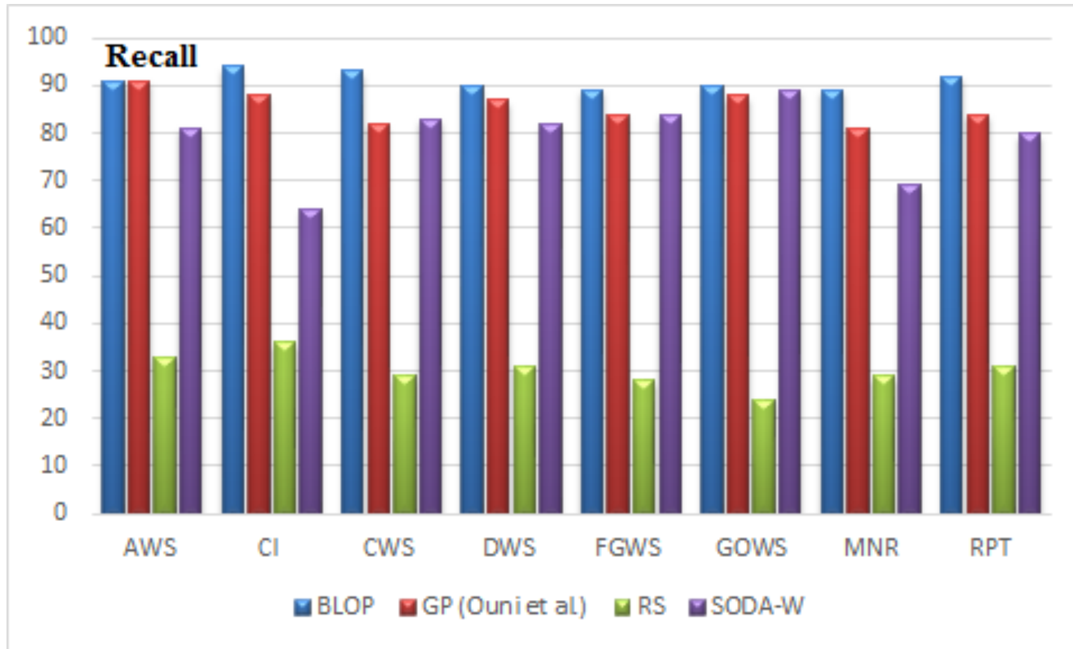

Figure 6: Median recall value over 30 runs on all the 10 Web service categories using the different detection techniques with a 95% confidence level (α < 5%)

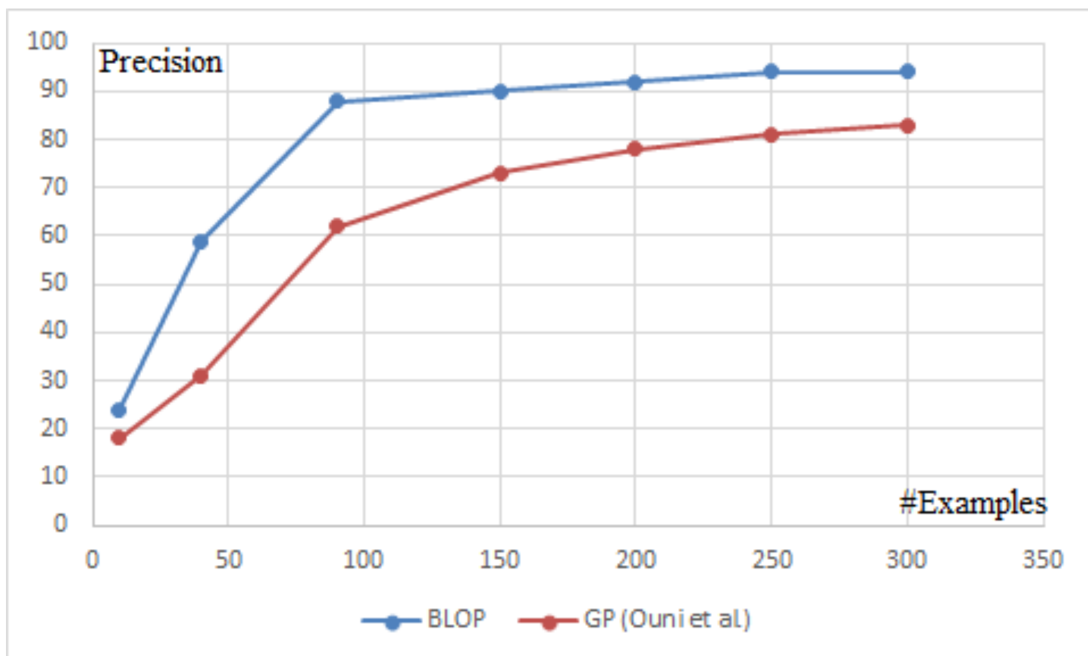

Figure 7: The impact of the number of Web service defect examples on the quality of the results (Precision on the *Financial* Web services).

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 7 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. [8] (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 [8] and an approach not based on heuristic search [5] (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.

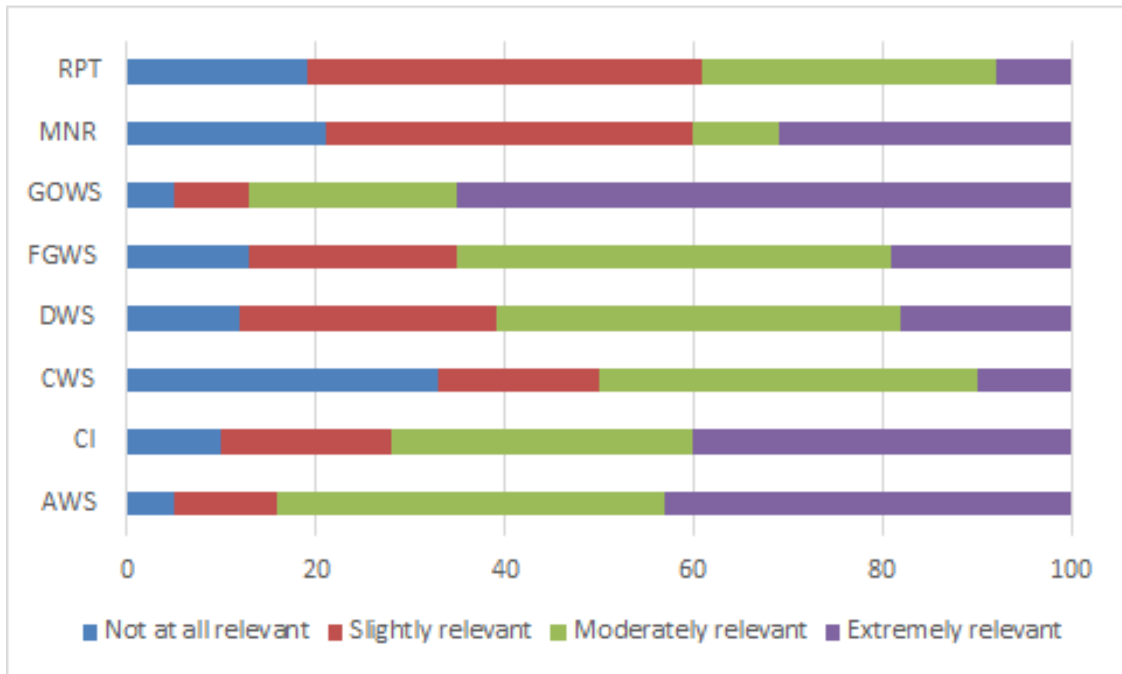

Figure 8: The relevance of detected Web service defects evaluated by the subjects.

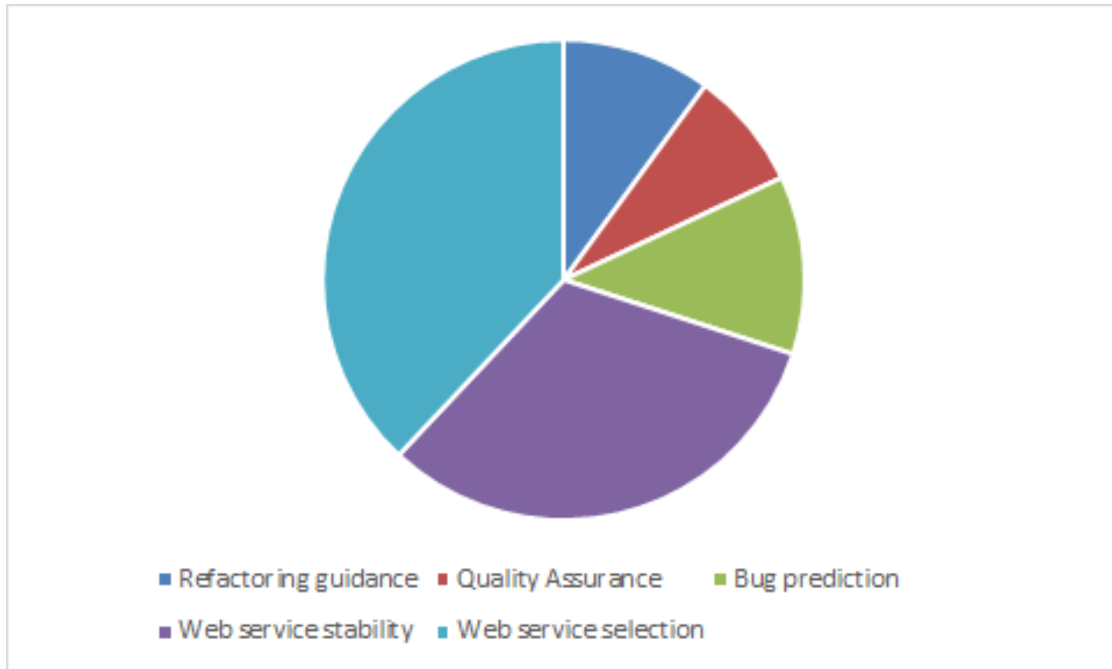

Figure 9: The usefulness of detected Web service defects evaluated by the subjects

Figure 8 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 9 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.

Chapter 5: Related Work

Detecting and specifying antipatterns in SOA and Web services is a relatively new area. The first book in the literature was written by Dudney et al. [1] and provides informal definitions of a set of Web service antipatterns. More recently, Rotem-Gal-Oz described the symptoms of a range of SOA antipatterns [2]. Furthermore, Kr´al et al. [3] listed seven "popular" SOA antipatterns that violate accepted SOA principles. In addition, a number of research works have addressed the detection of such antipatterns. Recently, Moha et al. [4] have proposed a rule-based approach called SODA for SCA systems (Service Component Architecture). Later, Palma et al. [5] extended this work for Web service antipatterns in SODA-W. The proposed approach relies on declarative rule specification using a domain-specific language (DSL) to specify/identify the key symptoms that characterize an antipattern using a set of WSDL metrics. In another study, Rodriguez et al. [11] [12] and Mateos et al. [13] provided a set of guidelines for service providers to avoid bad practices while writing WSDLs. Based on some heuristics, the authors detected eight bad practices in the writing of WSDL for Web services. In other work [14], the authors presented a repository of 45 general antipatterns in SOA. The goal of this work is a comprehensive review of these antipatterns that will help developers to work with clear understanding of patterns in phases of software development and so avoid many potential problems. Mateos et al. [15] have proposed an interesting approach towards generating WSDL documents with less antipatterns using text mining techniques.

Recently, Ouni et al. [6] [8] proposed a search-based approach based on standard GP to find regularities, from examples of Web service antipatterns, to be translated into detection rules. However, the proposed approach can deal only with Web service interface metrics and cannot consider all Web service antipattern symptoms. Similar to [5], the latter did not consider the deviation from common design practices which leads to several false positives.

Chapter 6: Conclusion And Future Work

In this thesis, 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 [8] and a non-search-based approach [5].

As part of our future work, we are planning to extend the current work by proposing a bi-level approach for the correction of Web service defects. Furthermore, owe will propose several new measures that can be used to rank the detected Web service defects by our rules. Finally, we will extend our experiments by considering a larger set of subjects, defects and Web services.

References

1. Dudney B., Krozak J., K., Asbury S., and Osborne D., J2EE Antipatterns. John Wiley; Sons, Inc., 1st edition (2003).

2. Rotem-Gal-Oz A., Bruno, E., and Dahan, U., SOA Patterns. Manning Publications, 38-62 (2012).

3. Kral J. and Zemlicka M., “Popular SOA Antipatterns,” in Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009. IEEE, pp. 271–276 (2009).

4. Moha N., Palma F., Nayrolles M., Conseil B. J., Gueheneuc Y.-G., Baudry B., and Jezequel J.-M., “Specification and detection of soa antipatterns,” in Service-Oriented Computing. Springer, pp. 1–16 (2012).

5. Palma F., Moha N., Tremblay G., and Gueheneuc Y.-G., “Specification and detection of soa antipatterns in web services,” in Software Architecture. Springer, pp. 58–73 (2014).

6. Bard J.. Practical Bilevel Optimization: Algorithms and Applications. The Netherlands: Kluwer, Vol 30 (1998).

7. Giri B. K., Hakanen J., Miettinen K., and Chakraborti N., “Genetic programming through bi-objective genetic algorithms with a study of a simulated moving bed process involving multiple objectives,” Applied Soft Computing, vol. 13, no. 5, pp. 2613–2623 (2013).

8. Ouni A., Kessentini M., and Inoue K., “Search-based web service antipatterns detection,” in IEEE Transactions on Services Computing, to appear. IEEE, pp. 1-21 (2016).

9. Frakes W. B. and Baeza-Yates R., Eds., Information Retrieval: Data Structures and Algorithms. Upper Saddle River, NJ, USA: Prentice-Hall, Inc. (1992).

10. Rodriguez J. M., Crasso M., Mateos C., and Zunino A., “Best practices for describing, consuming, and discovering web services: a comprehensive toolset,” Software: Practice and Experience, vol. 43, no. 6, pp. 613–639 (2013).

11. Rodriguez J. M., Crasso M., Zunino A., and Campo M., "Automatically detecting opportunities for web service descriptions improvement," in Software Services for e-World. Springer, pp. 139–150 (2010).

12. Ankur S., Pekka M., Anton F., Kalyanmoy D., Multi-objective Stackelberg game between a regulating authority and a mining company: A case study in environmental economics. IEEE Congress on Evolutionary Computation, Cancun, Mexico, 478–485 (2013).

13. Mateos C., Zunino A., and Coscia J. L. O., "Avoiding WSDL Bad Practices in Code-First Web Services," SADIO Electronic Journal of Informatics and Operational Research, vol. 11, no. 1, pp. 31–48 (2012).

14. Torkamani M. A. and Bagheri H., "A Systematic Method for Identification of Anti-patterns in Service Oriented System Development," International Journal of Electrical and Computer Engineering, vol. 4, no. 1, pp. 16–23 (2014).

15. Mateos C., Rodriguez J. M., and Zunino A., "A tool to improve code-first web services discoverability through text mining techniques," Software: Practice and Experience, vol. 45, no. 7, pp. 925–948 (2015).